

Вывод дерева

Севастьянов В., 251 гр., 2021–22 уч.г.

Проверка правильности работы дерева и его представление преподавателю на проверку будут проходить удобнее, если визуализировать его вывод в консоль. Здесь представлен способ сделать это в операционной системе Windows, однако аналогичные действия могут повторить и пользователи unix-систем. Более того, в разделе «**Модернизации и альтернатива**» представлен **альтернативный алгоритм** (ссылка активная), не требующий для своего функционирования эксклюзивной для Windows библиотеки `windows.h`. Понятия *терминал* и *консоль*, а также *каретка* и *курсор* здесь будут использоваться как равнозначные.

Целью данной работы является не только визуализация вывода дерева, но и знакомство с библиотекой `windows.h` или самостоятельное изучение её аналогов для unix-систем.

Используемые наименования

Сначала рассмотрим вывод обычного дерева бинарного поиска, работающего с типом данных `int`, а точнее, с числами от 0 до 99, чтобы было проще. Пусть структура узла имеет имя `node` и поля `node *left`, `*right` — указатели на левого и правого ребёнка соответственно, `int key` — информационное поле. Указатель на корень дерева — `node *root` — это будет глобальная переменная.

Идея алгоритма

Главная идея заключается в том, что сначала резервируется место для вывода всего дерева, при этом запоминаются изначальное и конечное (после вывода) положения каретки. Этим будем заниматься функция `print()`. Внутри неё вызывается функция проверки достатка места — `isSizeOfConsoleCorrect()`, и, в случае нехватки места, программа переходит в режим ожидания до исправления ситуации. Далее вызывается рекурсивная функция `print_helper()`, которая совершает прямой (`preorder`) обход дерева и непосредственно выводит значения на экран внутри зарезервированной области. Для работы алгоритма также потребуется функция вычисления высоты дерева — `max_height()`.

Функция `max_height()`

Рекурсивная реализация функции довольно проста, ей достаточно передавать всего три аргумента:

1. текущий узел, рассматриваемый функцией;
2. переменная, в которой хранится наибольшая достигнутая глубина, передаваемая по ссылке;
3. текущая глубина (добавим значение по умолчанию, равное 1).

```
void max_height(node *x, short &max, short deepness = 1) { // требует проверки на
    существование корня
    if (deepness > max) max = deepness;
```

```

    if (x->left) max_height(x->left, max, deepness + 1);
    if (x->right) max_height(x->right, max, deepness + 1);
}

```

Эта функция требует проверки на существование корня перед вызовом, так как она обращается к его полям, поэтому желательно сделать её приватной (ООП реализация приведена [ниже](#)). Результат сохраняется в переменной, которая была передана вторым параметром по ссылке. Стоит отметить, что функция также подходит для нахождения высоты поддеревьев при передаче первым параметром их корней, это позволяет сделать модернизацию, приведённую в разделе [«Модернизации и альтернатива»](#).

Функция print()

Сама функция не должна принимать никаких аргументов и что-либо возвращать, тогда заготовка функции примет следующий вид:

```

void print() {
}

```

В самом начале проверим, что дерево не является пустым, чтобы не задумываться об этом далее:

```

if (root) {
}

```

Все дальнейшие действия будем осуществлять внутри этого условного оператора. Теперь нужно узнать высоту дерева, от этого будут отталкиваться дальнейшие вычисления. Для этого создадим переменную с минимально возможным значением высоты дерева, в данном случае — единичным, и воспользуемся функцией `max_height()`.

```

short max = 1;
max_height(root, max);

```

Теперь немного подробнее разберём механизм вывода. Мы будем использовать библиотеку `windows.h`, которая предоставляет интерфейс доступа к Windows API, в нашем случае — позволит работать с консолью. Отсчёт координат в консоли ведётся от левой верхней ячейки её буфера — точка (0; 0). Поэтому, чтобы вывести дерево целиком, необходимо либо выводить его построчно, что требует организации поуровневого обхода, либо заранее выделить место под всё дерево и просто выводить элементы в определённых местах, устанавливая каретку в нужные координаты (первый способ будет рассмотрен в разделе [«Модернизации и альтернатива»](#)). Данный алгоритм подразумевает второй способ, поэтому далее нам предстоит выделить место, однако перед этим нужно проверить, имеет ли консоль достаточный размер.

С высотой всё просто, необходимой и достаточной будет просто высота дерева, уже хранящаяся в переменной `max`, с шириной всё несколько сложнее. Количество элементов на k -ом

уровне при нумерации с нуля равно 2^k . Пусть на нижнем уровне для вывода каждого элемента будет отводиться 4 символа: крайние 2 для выдерживания расстояния между элементами, а центральные 2 — для вывода однозначных и двузначных чисел. Тогда необходимая ширина, определяющаяся по нижнему уровню, вычисляется как $4 * 2^k = 2^{k+2}$. Однако размеры монитора, а следовательно и консоли, ограничены. Чтобы покрыть большинство случаев, ограничимся шириной вывода в 128 символов, то есть не более 32-ух элементов на нижнем уровне. Таким образом, гарантированно мы сможем выводить деревья с высотой только от 1 до 6, хотя Вы можете изменить максимальное значение ширины, если это позволяют сделать размеры монитора и настройки шрифта. Разумеется, в иных случаях можно выводить сообщение, что дерево слишком велико для вывода, и прекращать выполнение функции, но мы будем пытаться выводить любое дерево, тем более, зачастую это будет выглядеть вполне естественно. Для этого просто не позволим значению ширины `width` стать больше 128-ми. Возведение двойки в степень реализуем с помощью побитовых сдвигов (приоритет у них ниже, чем у арифметических операций).

```
short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
if (width > max_w) width = max_w;
```

Здесь используется `max + 1`, а не `max + 2`, т.к. переменная `max` хранит в себе количество уровней, тем самым как бы нумеруя их с единицы, а не с нуля.

Теперь проверим, выполняются ли необходимые требования по высоте и ширине. Для этого воспользуемся функцией `isSizeOfConsoleCorrect()`, к реализации которой мы перейдём несколько позже. Сейчас отметим только то, что она принимает два параметра: необходимые для вывода ширину и высоту; и возвращает `true`, если всё соблюдено, или выводит необходимые сообщения в терминал и возвращает `false` в противном случае. Тогда мы можем ставить программу «на паузу» до выполнения всех требований следующим способом:

```
while ( !isSizeOfConsoleCorrect(width, max) ) system("pause");
```

Учимся работать с кареткой

Отойдём на некоторое время от реализации функции `print()` и изучим имеющиеся в распоряжении инструменты библиотеки `windows.h`.

Первая функция на рассмотрение — `SetConsoleCursorPosition()`, она принимает два аргумента: дескриптор буфера экрана консоли типа данных `HANDLE`, и координаты новой позиции курсора типа `COORD`, представляющий собой структуру следующего вида (здесь приводится упрощённое представление):

```
struct COORD {
    short X;
    short Y;
};
```

Учитывая, что дескриптор будет общим для всей программы и будет использоваться в разных местах, объявим его **глобально** и инициализируем с помощью функции `GetStdHandle()` и параметра `STD_OUTPUT_HANDLE`, отвечающего за стандартное выходное устройство:

```
HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
```

Попробуем реализовать маленькую программу. Сначала просто выведем текст из двух строк, предварительно очистив консоль:

```
system("cls"); // очистка консоли
cout << "It's just text\nFor example";
```

Получим следующий вывод:

```
It's just text
For example
PS C:\Users\sevas\Desktop\vs\DS&A\RBT> 
```

Теперь после вывода поставим курсор в позицию (0; 0) и выведем ещё одно сообщение, программа примет следующий вид:

```
#include <iostream>
#include <windows.h>
using namespace std;

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);

int main() {
    system("cls");
    cout << "It's just text\nFor example";
    SetConsoleCursorPosition(outp, {0, 0});
    cout << "I am here!";
    return 0;
}
```

Вывод будет следующим:

```
I am here!text
PS C:\Users\sevas\Desktop\vs\DS&A\RBT> 
```

Как видно из рисунка, предыдущий вывод был перезаписан, а путь до текущей директории полностью «перекрыл» вторую строчку. Попробуем сохранить её нетронутой. Для этого сначала сохраним координаты курсора после первого вывода, сделаем второй и вернём курсор на место. В этом нам поможет объект типа `CONSOLE_SCREEN_BUFFER_INFO`, представляющий собой структуру, хранящую сведения о буфере экрана консоли. Однако только объявить переменную недостаточно, нужно будет проинициализировать её с помощью функции `GetConsoleScreenBufferInfo()`, принимающей два параметра: дескриптор буфера экрана консоли и ссылку на структуру. Объявим переменную глобально с именем `csbInfo`, программа примет следующий вид:

```
#include <iostream>
#include <windows.h>
using namespace std;
```

```

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

int main() {
    system("cls");
    cout << "It's just text\nFor example";
    GetConsoleScreenBufferInfo(outp, &csbInfo); // инициализируем глобальную переменную
    COORD endPos = csbInfo.dwCursorPosition; // сохраняем координаты, обратившись к нужному
    полю структуры
    SetConsoleCursorPosition(outp, {0, 0});
    cout << "I am here!";
    SetConsoleCursorPosition(outp, endPos); // возвращаем курсор на место
    return 0;
}

```

Вывод будет следующим:

```

I am here!text
For example
PS C:\Users\sevas\Desktop\vs\DS&A\RBT>

```

Также отметим, что у структуры типа `CONSOLE_SCREEN_BUFFER_INFO` есть поле `dwSize`, содержащее размер буфера экрана консоли в символьных столбцах и строках.

Функция `print()`: продолжение

Прежде всего объявим обе глобальные переменные, введённые в предыдущем разделе, так как они понадобятся нам в дальнейшем:

```

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

```

После того как мы сделали необходимую проверку размера консоли и уверены в том, что нам хватит места, можем приступить к дальнейшим действиям. Выделим необходимые нам `max` строчек (по высоте дерева) следующим простым образом:

```

for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода

```

Теперь сохраним конечные координаты, на которых сейчас и находится курсор (координата по оси $X = 0$).

```

GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
COORD endPos = csbInfo.dwCursorPosition;

```

Теперь вызовем функцию `print_helper()`, непосредственно выводящую дерево. Одним из передаваемых параметров (рассмотрение всех параметров будет приведено ниже, вместе с реализацией) будет координата позиции, относительно которой будет рассчитываться положение элемента при выводе. Для корня дерева — это позиция, располагающаяся на `max` строчек

выше, чем конечная. Конечно, до резервирования места можно было сохранить начальные координаты и вычислять уже конечные, но тогда, в случае, если пользователь забыл бы перенести курсор на новую строку, наше дерево сместилось бы вправо, что не очень хорошо, а в нашем случае просто перезапишется часть последней выведенной строки (последним перезаписанным символом будет тот, на месте которого выводится разряд единиц значения корня дерева). После того, как функция закончит свою работу, вернём каретку в нужное место.

```
print_helper(root, {0, short(endPos.Y - max)}, width >> 1);
SetConsoleCursorPosition(outp, endPos);
```

На этом реализация функции `print()` закончена, её общий вид следующий (не забудьте про глобальные переменные):

```
HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE); // глобально
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

void print() {
    if (root) {
        short max = 1;
        max_height(root, max);
        short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
        if (width > max_w) width = max_w;
        while (!isSizeOfConsoleCorrect(width, max)) system("pause");
        for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода
        GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
        COORD endPos = csbInfo.dwCursorPosition;
        print_helper(root, {0, short(endPos.Y - max)}, width >> 1);
        SetConsoleCursorPosition(outp, endPos);
    }
}
```

Функция `isSizeOfConsoleCorrect()`

Согласно описанию, приведённому при реализации функции `print()`, заготовка функции `isSizeOfConsoleCorrect()` должна иметь следующий вид:

```
bool isSizeOfConsoleCorrect(const short &width, const short &height) {
}
```

Приступим к реализации. Сначала получим размеры консоли в данный момент времени:

```
GetConsoleScreenBufferInfo(outp, &csbInfo);
COORD szOfConsole = csbInfo.dwSize;
```

Теперь проверим, достаточны ли размеры консоли для вывода дерева, сравнивая текущие размеры с переданными и выводя соответствующие сообщения в случае недостатка места.

```

if (szOfConsole.X < width && szOfConsole.Y < height) cout << "Please increase the height
    and width of the terminal. ";
else if (szOfConsole.X < width) cout << "Please increase the width of the terminal. ";
else if (szOfConsole.Y < height) cout << "Please increase the height of the terminal. ";

```

Чтобы пользователь знал необходимые и текущие размеры, выведем их. Также вернём в место вызова `true` или `false`, если места достаточно или нет соответственно.

```

if (szOfConsole.X < width || szOfConsole.Y < height) {
    cout << "Size of your console now: " << szOfConsole.X << ' ' << szOfConsole.Y << ".
    Minimum required: "
        << width << ' ' << height << ".\n";
    return false;
}
else return true;

```

Общий вид функции `isSizeOfConsoleCorrect()` следующий:

```

bool isSizeOfConsoleCorrect(const short &width, const short &height) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD szOfConsole = csbInfo.dwSize;
    if (szOfConsole.X < width && szOfConsole.Y < height) cout << "Please increase the
    height and width of the terminal. ";
    else if (szOfConsole.X < width) cout << "Please increase the width of the terminal. ";
    else if (szOfConsole.Y < height) cout << "Please increase the height of the terminal.
    ";
    if (szOfConsole.X < width || szOfConsole.Y < height) {
        cout << "Size of your terminal now: " << szOfConsole.X << ' ' << szOfConsole.Y
            << ". Minimum required: " << width << ' ' << height << ".\n";
        return false;
    }
    return true;
}

```

Также возможен менее «многословный» и громоздкий её вариант:

```

bool isSizeOfConsoleCorrectShort(const short &width, const short &height) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD szOfConsole = csbInfo.dwSize;
    if (szOfConsole.X < width || szOfConsole.Y < height) {
        cout << "Please increase the size of the terminal. Size of your terminal now: " <<
        szOfConsole.X << ' '
            << szOfConsole.Y << ". Minimum required: " << width << ' ' << height << ".\n";
        return false;
    }
    return true;
}

```

Функция print_helper()

Заготовка функции имеет следующий вид:

```
void print_helper(node *x, const COORD pos, const short offset) {  
  
}
```

Первый параметр нужен, очевидно, для обхода дерева, второй отвечает за координаты позиции, в которую будет выводиться элемент с помощью функции `setw()` из библиотеки `iomanip`, а третий нужен для смещения элементов по оси X, в том числе в качестве параметра для `setw()`.

Напомним, что делает функция `setw()`. Она устанавливает ширину следующего вывода, передаваемую ей в качестве параметра, например, для кода `cout << 100 << setw(5) << 200 << 300 << '\n'`; вывод будет следующим:

```
100 200300
```

То есть под запись числа 200 было отведено 5 символов, и, так как число оказалось короче, то оно «прижалось» к правому краю, а на число 300 действие функции уже не распространяется (если выводимый объект требует места для своего вывода больше, чем ему предоставлено, то его запись будет начата в самой левой позиции отведённого места и продолжена вправо до полного вывода).

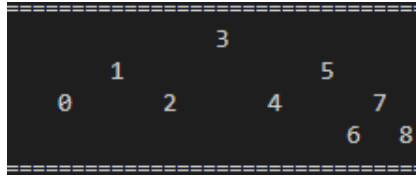
Ранее мы решили, что на нижнем уровне для каждого элемента будет отводиться 4 символа. На предпоследнем уровне, если он существует, соответственно, будет выделяться в 2 раза больше символов — 8, и так далее. Используя функцию `setw()`, мы можем выводить двузначное число посередине предоставленной ему области чётной ширины следующим образом: в качестве параметра будем передавать сумму единицы и половины отведённого для вывода числа символов. Например, если это нижний уровень, то таким образом будет передано число $\frac{4}{2} + 1 = 3$, то есть разряд единиц будет выведен на месте третьего символа из четырёх, а разряд десятков — на месте второго. По этой причине при вызове функции `print_helper()` из функции `print()` третьим параметром передавалась необходимая для печати всего дерева ширина, делённая пополам: `width >> 1`, так как корневой элемент должен располагаться посередине всего дерева. А на каждом последующем уровне это смещение (отсюда и имя переменной — `offset`) уменьшается в 2 раза. Если число однозначное, то оно просто будет выведено в правой ячейке.

Предполагая, что передаваемые в функцию координаты места вывода верны, будем сразу устанавливать в них курсор и выводить значение текущего узла:

```
SetConsoleCursorPosition(outp, pos);  
cout << setw(offset + 1) << x->key;
```

После этого функция должна рекурсивно вызываться для левого и правого ребёнка, если они существуют, при этом передаваемое смещение будет в 2 раза меньше текущего. Осталось разобраться только с координатами. Здесь всё просто: если ребёнок левый, то координата по X должна быть неизменной, а элемент будет напечатан левее из-за меньшего смещения в `setw()` (таким образом значение левого ребёнка будет напечатано на половину текущего

смещения левее, чем рассматриваемый элемент), а если правый, то его значение должно отстоять на половину текущего смещения правее рассматриваемого элемента, добиться этого можно, прибавив к координате по X текущее смещение.



Координата по Y же просто увеличивается на 1:

```
if (x->left) print_helper(x->left, {pos.X, short(pos.Y + 1)}, offset >> 1);
if (x->right) print_helper(x->right, {short(pos.X + offset), short(pos.Y + 1)}, offset >> 1);
```

Общий вид функции print_helper() следующий:

```
void print_helper(node *x, const COORD pos, const short offset) {
    SetConsoleCursorPosition(outp, pos);
    cout << right << setw(offset + 1) << x->key;
    if (x->left) print_helper(x->left, {pos.X, short(pos.Y + 1)}, offset >> 1);
    if (x->right) print_helper(x->right, {short(pos.X + offset), short(pos.Y + 1)}, offset >> 1);
}
```

Общий вид

Общий вид всех требуемых библиотек, глобальных переменных и функций следующий:

```
#include <iostream>
#include <windows.h>
#include <iomanip>
using namespace std;

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

void max_height(node *x, short &max, short deepness = 1) { // требует проверки на существование корня
    if (deepness > max) max = deepness;
    if (x->left) max_height(x->left, max, deepness + 1);
    if (x->right) max_height(x->right, max, deepness + 1);
}

bool isSizeOfConsoleCorrect(const short &width, const short &height) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD szOfConsole = csbInfo.dwSize;
    if (szOfConsole.X < width && szOfConsole.Y < height) cout << "Please increase the height and width of the terminal. ";
    else if (szOfConsole.X < width) cout << "Please increase the width of the terminal. ";
    else if (szOfConsole.Y < height) cout << "Please increase the height of the terminal. ";
    if (szOfConsole.X < width || szOfConsole.Y < height) {
        cout << "Size of your terminal now: " << szOfConsole.X << ' ' << szOfConsole.Y
            << ". Minimum required: " << width << ' ' << height << ".\n";
        return false;
    }
}
```

```

    return true;
}

void print_helper(node *x, const COORD pos, const short offset) {
    SetConsoleCursorPosition(outp, pos);
    cout << setw(offset + 1) << x->key;
    if (x->left) print_helper(x->left, {pos.X, short(pos.Y + 1)}, offset >> 1);
    if (x->right) print_helper(x->right, {short(pos.X + offset), short(pos.Y + 1)}, offset >> 1);
}

void print() {
    if (root) {
        short max = 1;
        max_height(root, max);
        short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
        if (width > max_w) width = max_w;
        while (!isSizeOfConsoleCorrect(width, max)) system("pause");
        for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода
        GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
        COORD endPos = csbInfo.dwCursorPosition;
        print_helper(root, {0, short(endPos.Y - max)}, width >> 1);
        SetConsoleCursorPosition(outp, endPos);
    }
}

```

Разумеется, данный алгоритм подходит и, в случае необходимости, легко адаптируется под любое бинарное дерево. Одна из таких адаптаций приведена далее.

Красно-чёрное дерево

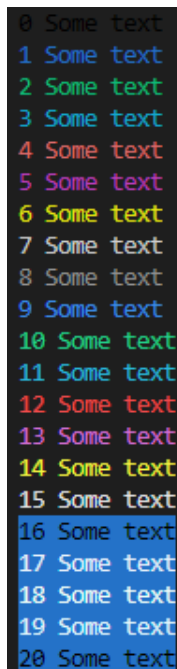
Вывод красно-чёрного дерева (КЧД) будет отличаться от вывода простого дерева бинарного поиска только тем, что мы просто «раскрасим» его элементы.

Цвет выводимому элементу (точнее, всем выводимым далее символам) мы зададим с помощью команды `SetConsoleTextAttribute()`, передав дескриптор буфера экрана консоли `outp` и номер цвета — атрибут (всего их 256 — по «вместимости» байта). Чтобы ознакомиться со всеми цветами, напомним следующую программу:

```
#include <iostream>
#include <windows.h>
using namespace std;

int main() {
    HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
    for(int k = 0; k <= 255; ++k) {
        SetConsoleTextAttribute(outp, k);
        cout << k << " Some text" << '\n';
    }
    SetConsoleTextAttribute(outp, 7);
    return 0;
}
```

Стандартным цветом текста в консоли является цвет с кодом 7, поэтому, после вывода всех цветов, вернём исходный, в противном случае все дальнейшие записи в консоли будут иметь последний установленный цвет. Так выглядят первые несколько цветов:



```
0 Some text
1 Some text
2 Some text
3 Some text
4 Some text
5 Some text
6 Some text
7 Some text
8 Some text
9 Some text
10 Some text
11 Some text
12 Some text
13 Some text
14 Some text
15 Some text
16 Some text
17 Some text
18 Some text
19 Some text
20 Some text
```

Первые 16 атрибутов задают цвет только тексту, все последующие атрибуты будут циклически использовать те же цвета текста и будут отличаться лишь фоном. На скриншоте работы программы это плохо заметно из-за контраста, однако попробуйте убедиться в сказанном самостоятельно.

Для вывода красных элементов дерева будем использовать цвет с кодом 12 — ярко-красный, а для чёрных — 8 — серый, так как цвет с номером 0 плохо различим.

Пусть структура узла красно-чёрного дерева имеет следующий вид:

```
struct node {
    int key;
    bool color;
    node *parent, *left, *right;
};
```

А цвета пусть задаются с помощью «дефайнов» (директива `#define` определяет идентификатор и последовательность символов, которой будет замещаться данный идентификатор при его обнаружении в тексте программы), пусть красным цветом будет булево значение `true` :

```
#define RED true
#define BLACK false
```

Тогда, чтобы задать красный цвет узлу `x`, достаточно написать `x.color = RED`.

В написанные ранее функции достаточно добавить всего две строчки. В `print_helper()` перед выводом значения нужно установить нужный атрибут в зависимости от цвета узла. Сделаем это с помощью тернарного оператора:

```
SetConsoleTextAttribute(outp, x->color == RED ? 12 : 8);
```

А после вызова функции `print_helper()` в функции `print()` необходимо установить стандартный цвет:

```
SetConsoleTextAttribute(outp, 7);
```

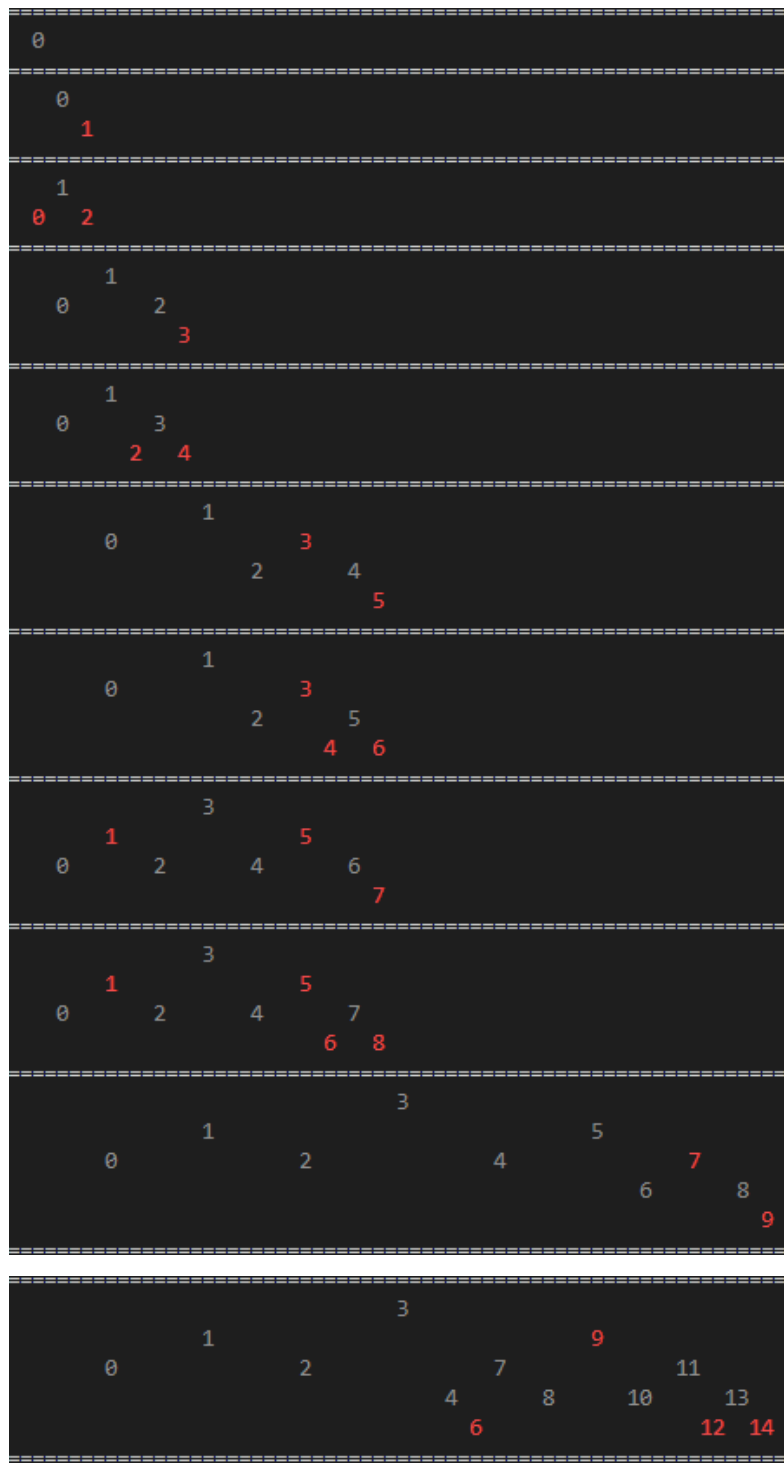
Изменённые функции примут следующий вид:

```
void print_helper(node *x, const COORD pos, const short offset) {
    SetConsoleTextAttribute(outp, x->color == RED ? 12 : 8);
    SetConsoleCursorPosition(outp, pos);
    cout << setw(offset + 1) << x->key;
    if (x->left) print_helper(x->left, {pos.X, short(pos.Y + 1)}, offset >> 1);
    if (x->right) print_helper(x->right, {short(pos.X + offset), short(pos.Y + 1)}, offset >> 1);
}

void print() {
    if (root) {
        short max = 1;
        max_height(root, max);
        short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
        if (width > max_w) width = max_w;
        while (!isSizeOfConsoleCorrect(width, max)) system("pause");
        for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода
        GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
        COORD endPos = csbInfo.dwCursorPosition;
```

```
print_helper(root, {0, short(endPos.Y - max)}, width >> 1);
SetConsoleCursorPosition(outp, endPos);
SetConsoleTextAttribute(outp, 7);
}
}
```

Примеры работы программы:



ООП реализация красно-чёрного дерева

Здесь просто рассмотрим, какие методы лучше сделать приватными, а также сделаем приватным полем корень дерева:

```
#include <iostream>
#include <windows.h>
#include <iomanip>
using namespace std;

#define RED true
#define BLACK false

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

bool isSizeOfConsoleCorrect(const short &width, const short &height);

template <typename T>
struct node {
    T key;
    bool color;
    node<T> *parent, *left, *right;
    // здесь должны быть конструктор(-ы) и деструктор
};

template <typename T>
class RBT {
public:
    // прочие публичные методы
    void print();

private:
    // прочие приватные методы
    void max_height(node<T> *x, short &max, short deepness = 1); // знач. по умолчанию
    void print_helper(node<T> *x, const COORD pos, const short offset);

    // например, размер дерева
    node<T> *root;
};

template <typename T>
void RBT<T>::max_height(node<T> *x, short &max, short deepness) { // требует проверки на
    существование корня
    if (deepness > max) max = deepness;
    if (x->left) max_height(x->left, max, deepness + 1);
    if (x->right) max_height(x->right, max, deepness + 1);
}

// прочие реализации методов
```

Модернизации и альтернатива

Альтернативный алгоритм

Разумеется, рассмотренный алгоритм не является единственным оптимальным подходом к выводу дерева. В случае возникновения проблем с перемещением курсора, например, при работе с другой операционной системой, можно прибегнуть к следующему способу. Сначала перенесём дерево в двумерный динамический массив, причём сохранять в нём будем указатели на узлы, чтобы, в случае необходимости (например, для цветного вывода КЧД), иметь доступ ко всем полям. Делать это будем, опять же, рекурсивно. И только потом уже будем выводить дерево идя по строкам массива. При этом функция проверки размера консоли сведётся к проверке только ширины. Приступим к реализации.

Аналогично предыдущему алгоритму, проверим, что корень существует, с помощью уже реализованной функции `max_height()` узнаем высоту дерева и, соответственно, количество строк в массиве, затем объявим его и проинициализируем значения нулевыми указателями. Количество элементов в каждой строке равно максимально возможному количеству элементов на этом уровне дерева. В конце не забудем освободить выделенную память.

```
void print() {
    if (root) {
        short max = 1, offset = 1, width = 1;
        max_height(root, max);
        node ***arr = new node **[max];
        for(short i = 0; i < max; ++i) {
            arr[i] = new node *[offset];
            for (short j = 0; j < offset; ++j)
                arr[i][j] = nullptr;
            offset <= 1;
        }

        for(short i = 0; i < max; ++i)
            delete [] arr[i];
        delete [] arr;
    }
}
```

Исходя из уже рассмотренного **принципа выделения места** для вывода дерева, нам необходимо 2^{max+1} символов, а после цикла $offset = 2^{max}$, поэтому домножим его на 2, также это понадобится далее, в цикле вывода дерева, для сокращения числа делений. Затем вызовем функции проверки размера консоли и заполнения массива (в случае возникновения трудностей с изменением под unix-систему функции проверки размера консоли, её можно исключить из программы).

```
offset <= 1;
while (!isSizeOfConsoleCorrect(offset)) system("pause"); // опционально
print_helper(arr, root);
```

Функция проверки размера консоли теперь выглядит следующим образом:

```
bool isSizeOfConsoleCorrect(const short &width) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    if (csbInfo.dwSize.X < width) {
        cout << "Please increase the width of the terminal. Width of your terminal now: "
              << csbInfo.dwSize.X << ". Minimum required: " << width << ".\n";
        return false;
    }
    return true;
}
```

Рекурсивная функция заполнения массива крайне проста: нужно лишь правильно вычислять индексы элементов в массиве (`deepness` и `ind`) и оставлять пустые указатели на местах, где соответствующий узел дерева отсутствует. Код такой функции может выглядеть следующим образом:

```
void print_helper(node ***arr, node *x, const short deepness = 0, const short ind = 0) {
    arr[deepness][ind] = x;
    if (x->left) print_helper(arr, x->left, deepness + 1, 2 * ind);
    if (x->right) print_helper(arr, x->right, deepness + 1, 2 * ind + 1);
}
```

Вернёмся к функции `print()`. В предыдущем алгоритме получилось так, что на каждом уровне самый левый элемент отстоит от левого края терминала на $2^{max-k} - 1$ символ, где k — номер уровня в дереве, на котором находится элемент, при нумерации с нуля (при том что на вывод элемента отводилось 2 символа), а расстояние между элементами на каждом уровне было равно $2^{max+1-k} - 2$, при этом в функцию `setw()` передаётся число на 2 большее (с учётом места для вывода числа). Тогда код цикла вывода может выглядеть следующим образом:

```
for (short i = 0; i < max; ++i) {
    cout << setw((offset >> 1) + 1);
    arr[i][0] ? cout << arr[i][0]->key : cout << ' ';
    for (short j = 1; j < width; ++j) {
        cout << setw(offset);
        arr[i][j] ? cout << arr[i][j]->key : cout << ' ';
    }
    offset >>= 1;
    width <= 1;
    cout << '\n';
}
```

Общий вид всех требуемых библиотек, глобальных переменных и функций следующий:

```
#include <iostream>
#include <windows.h>
#include <iomanip>
using namespace std;
```



```

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;

void max_height(node *x, short &max, short deepness = 1) { // требует проверки на существование корня
    if (deepness > max) max = deepness;
    if (x->left) max_height(x->left, max, deepness + 1);
    if (x->right) max_height(x->right, max, deepness + 1);
}

bool isSizeOfConsoleCorrect(const short &width) {
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    if (csbInfo.dwSize.X < width) {
        cout << "Please increase the width of the terminal. Width of your terminal now: "
             << csbInfo.dwSize.X << ". Minimum required: " << width << ".\n";
        return false;
    }
    return true;
}

void print_helper(node ***arr, node *x, const short deepness = 0, const short ind = 0) {
    arr[deepness][ind] = x;
    if (x->left) print_helper(arr, x->left, deepness + 1, 2 * ind);
    if (x->right) print_helper(arr, x->right, deepness + 1, 2 * ind + 1);
}

void print() {
    if (root) {
        short max = 1, offset = 1, width = 1;
        max_height(root, max);
        node ***arr = new node **[max];
        for(short i = 0; i < max; ++i) {
            arr[i] = new node *[offset];
            for (short j = 0; j < offset; ++j)
                arr[i][j] = nullptr;
            offset <= 1;
        }
        offset <= 1;
        while (!isSizeOfConsoleCorrect(offset)) system("pause"); // опционально
        print_helper(arr, root);
        for (short i = 0; i < max; ++i) {
            cout << setw((offset >> 1) + 1);
            arr[i][0] ? cout << arr[i][0]->key : cout << ' ';
            for (short j = 1; j < width; ++j) {
                cout << setw(offset);
                arr[i][j] ? cout << arr[i][j]->key : cout << ' ';
            }
            offset >= 1;
            width <= 1;
            cout << '\n';
        }
        for(short i = 0; i < max; ++i)
            delete [] arr[i];
        delete [] arr;
    }
}

```

В случае КЧД перед каждым выводом элемента достаточно установить тексту нужный атрибут, сделав проверку на непустоту указателя в массиве.

При исключении функции `isSizeOfConsoleCorrect()` получается полностью рабочая версия, не требующая для функционирования эксклюзивной для Windows библиотеки с её функциями.

Модернизация функции print(): вывод поддерева

Все приведённые реализации построены таким образом, чтобы с минимальными усилиями их можно было перестроить для вывода поддерева. Для этого достаточно передавать в функцию print() указатель на узел, с которого нужно начинать вывод и заменить все вхождения корня — переменной root — на переданное значение. А чтобы можно было вызывать функцию без передачи параметра, можно пойти на следующую хитрость: значением по умолчанию можно сделать пустой указатель, а в начале функции сделать проверку, в результате которой пустой указатель будет заменяться на корень, а непустой будет оставаться нетронутым. На примере самой первой реализации функции print() рассмотрим данную модернизацию:

```
void print(node *head = nullptr) {
    if (!head) head = root;
    if (head) { // т.к. root может быть nullptr
        short max = 1;
        max_height(head, max);
        short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
        if (width > max_w) width = max_w;
        while (!isSizeOfConsoleCorrect(width, max)) system("pause");
        for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода
        GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
        COORD endPos = csbInfo.dwCursorPosition;
        print_helper(head, {0, short(endPos.Y - max)}, width >> 1);
        SetConsoleCursorPosition(outp, endPos);
    }
}
```

Эта и следующая модернизации применимы ко всем реализациям.

Модернизация функции print(): вывод ограниченного числа уровней

Чтобы выводить не больше n уровней дерева, можно передавать n в функцию print() и ограничивать число max сверху числом n так же, как мы ограничивали width числом 128. Затем передавать уже ограниченное число max в функцию print_helper(), чтобы она переставала рекурсивно вызываться при достижении этой глубины, разумеется, также придётся поддерживать и глубину рекурсии (если это уже не делается в выбранной реализации). Сочетание данной модернизации с предыдущей позволит выводить не уместящиеся в консоли деревья в несколько этапов.