Procs
28 сен 2015 в 14:25

Бинарные деревья поиска и рекурсия – это просто

8 мин **599K**

Программирование*, C++*, Алгоритмы*

Из песочницы

Существует множество книг и статей по данной теме. В этой статье я попробую понятно рассказать самое основное.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

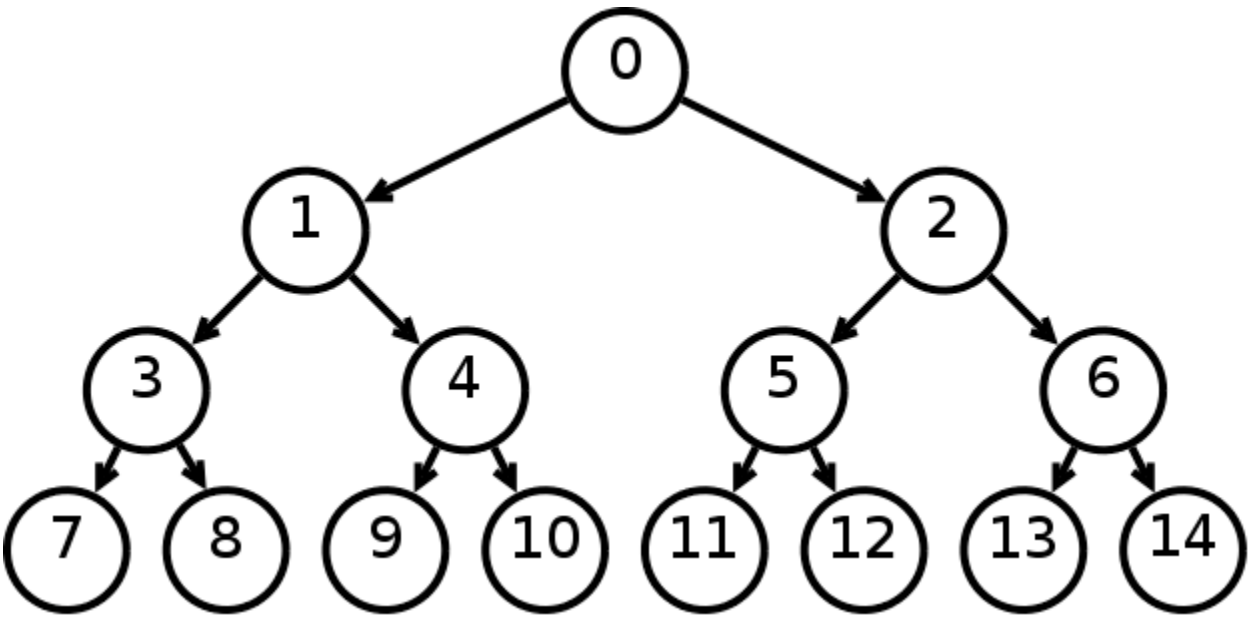


Рис. 1 Бинарное дерево

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске

элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

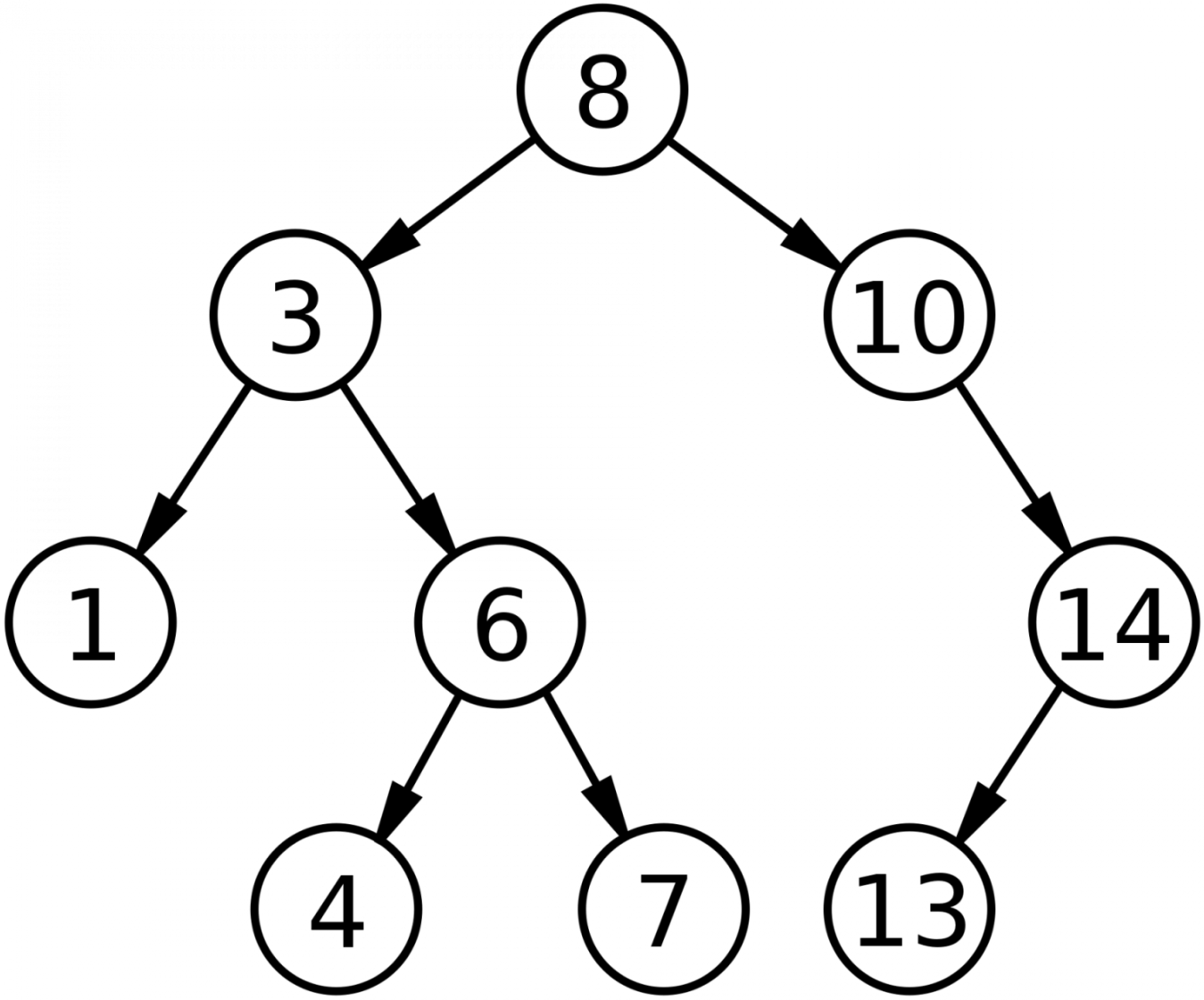


Рис. 2 Бинарное дерево поиска

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне

важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

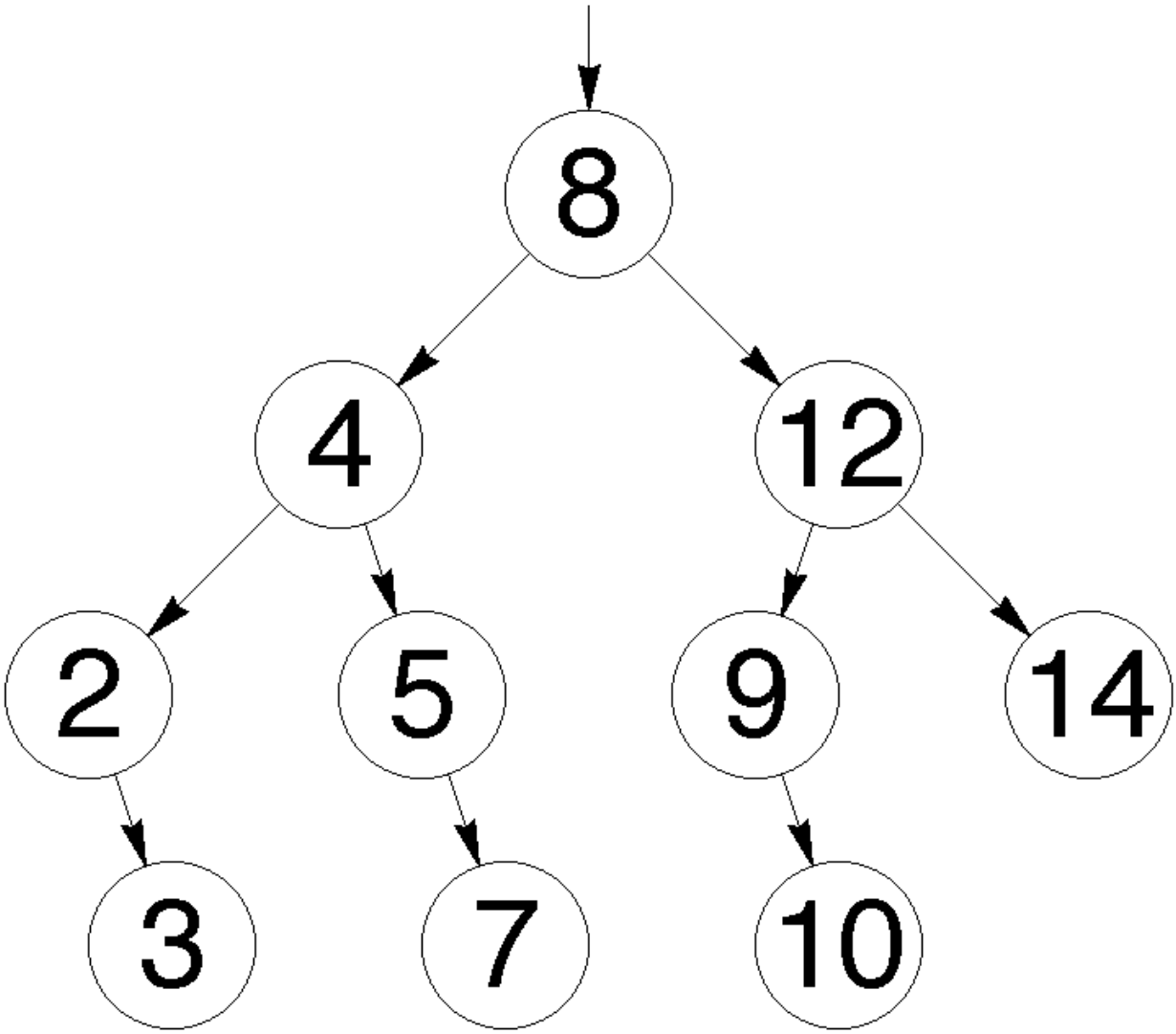


Рис. 3 Сбалансированное бинарное дерево поиска

Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалениями существующих. В случае, если набор элементов, хранящийся в структуре данных фиксирован и нет новых вставок и удалений, то массив предпочтительнее. Потому что поиск можно осуществлять алгоритмом бинарного поиска за то же логарифмическое время, но отсутствуют дополнительные издержки по хранению и использованию указателей. Например, в C++ ассоциативные контейнеры set и map представляют собой сбалансированное бинарное дерево поиска.

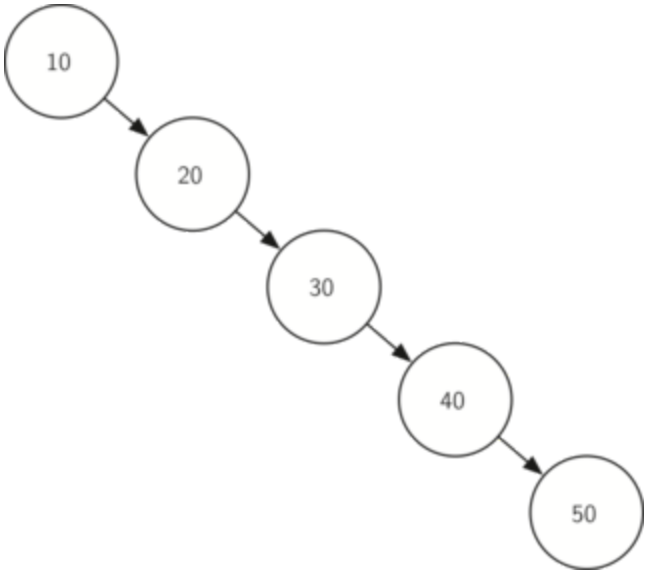


Рис. 4 Экстремально несбалансированное бинарное дерево поиска

Теперь кратко обсудим рекурсию. Рекурсия в программировании – это вызов функцией самой себя с другими аргументами. В принципе, рекурсивная функция может вызывать сама себя и с теми же самыми аргументами, но в этом случае будет бесконечный цикл рекурсии, который закончится переполнением стека. Внутри любой рекурсивной функции должен быть базовый случай, при котором происходит выход из функции, а также вызов или вызовы самой себя с другими аргументами. Аргументы не просто должны быть другими, а должны приближать вызов функции к базовому случаю. Например, вызов внутри рекурсивной функции расчета факториала должен идти с меньшим по значению аргументом, а вызовы внутри рекурсивной функции обхода дерева должны идти с узлами, находящимися дальше от корня, ближе к листьям. Рекурсия может быть не только прямой (вызов непосредственно себя), но и косвенной. Например А вызывает Б, а Б вызывает А. С помощью рекурсии можно эмулировать итеративный цикл, а также работу структуры данных стек (LIFO).

```
int factorial(int n)
{
    if(n <= 1)  // Базовый случай
    {
        return 1;
    }
    return n  * factorial(n - 1); //рекурсивеый вызов с другим аргументом
}
// factorial(1):     return 1
// factorial(2):     return 2 * factorial(1) (return 2 * 1)
```

```
// factorial(3):     return 3 * factorial(2) (return 3 * 2 * 1)
// factorial(4):     return 4 * factorial(3) (return 4 * 3 * 2 * 1)
// Вычисляет факториал числа n (n должно быть небольшим из-за типа int
// возвращаемого значения. На практике можно сделать long long  и вообще
// заменить рекурсию циклом. Если важна скорость рассчетов и не жалко память, то
// можно совсем избавиться от функции и использовать массив с предварительно
// посчитанными факториалами).

void reverseBinary(int n)
{
    if (n == 0)  // Базовый случай
    {
        return;
    }
    cout << n%2;
    reverseBinary(n/2); //рекурсивеый вызов с другим аргументом
}
// Печатает бинарное представление числа в обратном порядке

void forvardBinary(int n)
{
    if (n == 0)  // Базовый случай
    {
        return;
    }
    forvardBinary(n/2); //рекурсивеый вызов с другим аргументом
    cout << n%2;
}
// Поменяли местами две последние инструкции
// Печатает бинарное представление числа в прямом порядке
// Функция является примером эмуляции стека

void ReverseForvardBinary(int n)
{
    if (n == 0)  // Базовый случай
    {
        return;
    }
    cout << n%2; // печатает в обратном порядке
```

```
    ReverseForvardBinary(n/2); //рекурсивеый вызов с другим аргументом
    cout << n%2; // печатает в прямом порядке
}
// Функция печатает сначала бинарное представление в обратном порядке,
// а затем в прямом порядке


int product(int x, int y)
{
    if (y == 0)  // Базовый случай
    {
        return 0;
    }
    return (x + product(x, y-1)); //рекурсивеый вызов с другим аргументом
}
// Функция вычисляет произведение x * y ( складывает x y раз)
// Функция абсурдна с точки зрения практики,
// приведена для лучшего понимания рекурсии
```

Кратко обсудим деревья с точки зрения теории графов. Граф – это множество вершин и ребер. Ребро – это связь между двумя вершинами. Количество возможных ребер в графе квадратично зависит от количества вершин (для понимания можно представить турнирную таблицу сыгранных матчей). Дерево – это связный граф без циклов. Связность означает, что из любой вершины в любую другую существует путь по ребрам. Отсутствие циклов означает, что данный путь – единственный. Обход графа – это систематическое посещение всех его вершин по одному разу каждой. Существует два вида обхода графа: 1) поиск в глубину; 2) поиск в ширину.

Поиск в ширину (BFS) идет из начальной вершины, посещает сначала все вершины находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее. Алгоритм поиска в ширину является по своей природе нерекурсивным (итеративным). Для его реализации применяется структура данных очередь (FIFO).

Поиск в глубину (DFS) идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины. Алгоритм поиска в глубину по своей природе является рекурсивным. Для эмуляции рекурсии в итеративном варианте алгоритма применяется структура данных стек.

С формальной точки зрения можно сделать как рекурсивную, так и итеративную версию как поиска в ширину, так и поиска в глубину. Для обхода в ширину в обоих случаях необходима очередь. Рекурсия в рекурсивной реализации обхода в ширину всего лишь эмулирует цикл. Для обхода в глубину существует рекурсивная реализация без стека, рекурсивная реализация со стеком и итеративная реализация со стеком. Итеративная реализация обхода в глубину без стека невозможна.

Асимптотическая сложность обхода и в ширину и в глубину O(V + E), где V – количество вершин, E – количество ребер. То есть является линейной по количеству вершин и ребер. Записи O(V + E) с содержательной точки зрения эквивалентна запись O(max(V,E)), но последняя не принята. То есть, сложность будет определятся максимальным из двух значений. Несмотря на то, что количество ребер квадратично зависит от количества вершин, мы не можем записать сложность как O(E), так как если граф сильно разреженный, то это будет ошибкой.

DFS применяется в алгоритме нахождения компонентов сильной связности в ориентированном графе. BFS применяется для нахождения кратчайшего пути в графе, в алгоритмах рассылки сообщений по сети, в сборщиках мусора, в программе индексирования – пауке поискового движка. И DFS и BFS применяются в алгоритме поиска минимального покрывающего дерева, при проверке циклов в графе, для проверки двудольности.
Обходу в ширину в графе соответствует обход по уровням бинарного дерева. При данном обходе идет посещение узлов по принципу сверху вниз и слева направо. Обходу в глубину в графе соответствуют три вида обходов бинарного дерева: прямой (pre-order), симметричный (in-order) и обратный (post-order).

Прямой обход идет в следующем порядке: корень, левый потомок, правый потомок. Симметричный — левый потомок, корень, правый потомок. Обратный – левый потомок, правый потомок, корень. В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Если нам дано изображение дерева, и нужно найти его обходы, то может помочь следующая техника (см. рис. 5). Обводим дерево огибающей замкнутой кривой (начинаем идти слева вниз и замыкаем справа вверх). Прямому обходу будет соответствовать порядок, в котором огибающая, двигаясь от корня впервые проходит рядом с узлами слева. Для симметричного обхода порядок, в котором огибающая, двигаясь от корня

впервые проходит рядом с узлами снизу. Для обратного обхода порядок, в котором огибающая, двигаясь от корня впервые проходит рядом с узлами справа. В коде рекурсивного вызова прямого обхода идет: вызов, левый, правый. Симметричного – левый, вызов, правый. Обратного – левый правый, вызов.

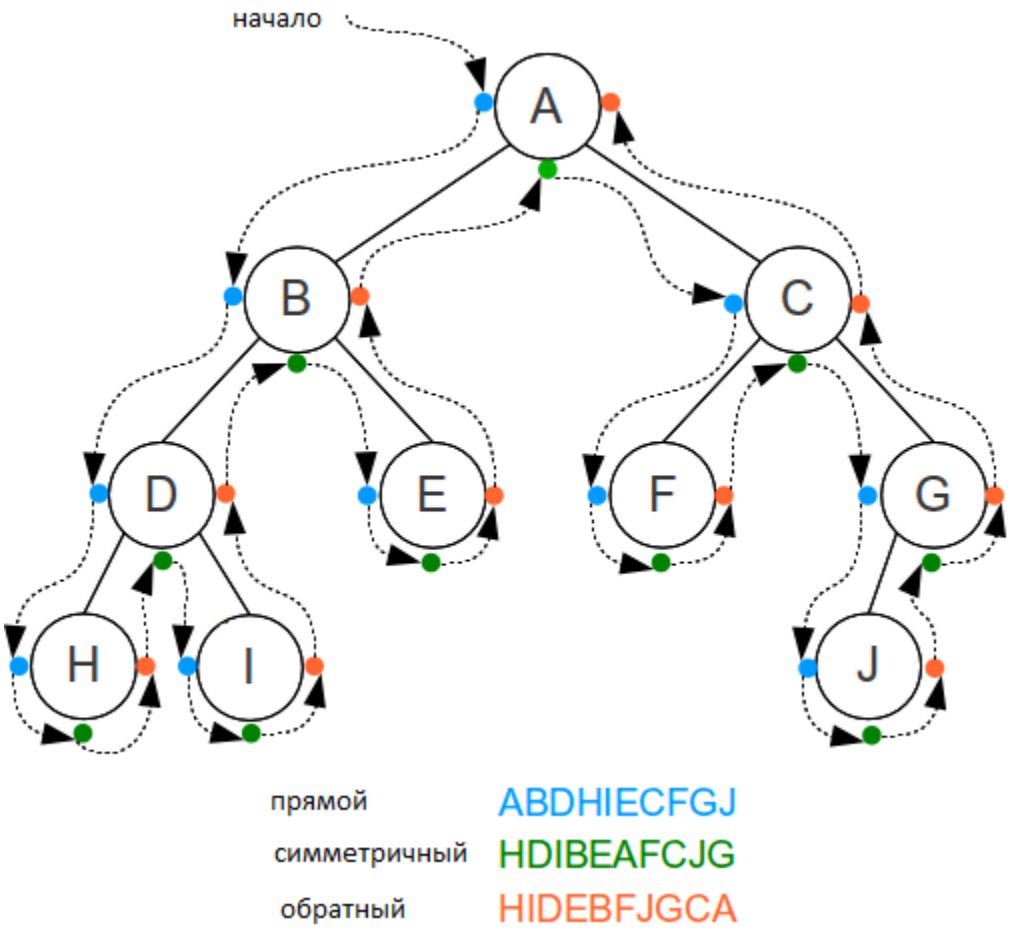


Рис. 5 Вспомогательный рисунок для обходов

Для бинарных деревьев поиска симметричный обход проходит все узлы в отсортированном порядке. Если мы хотим посетить узлы в обратно отсортированном порядке, то в коде рекурсивной функции симметричного обхода следует поменять местами правого и левого потомка.

```
struct  TreeNode
{
    double data; // ключ/данные
    TreeNode *left; // указатель на левого потомка
    TreeNode *right; // указатель на правого потомка
};
```

```
void levelOrderPrint(TreeNode *root) {
    if (root == NULL)
    {
       return;
    }
    queue<TreeNode *> q; // Создаем очередь
    q.push(root); // Вставляем корень в очередь

    while (!q.empty() ) // пока очередь не пуста
    {
        TreeNode* temp = q.front(); // Берем первый элемент в очереди
        q.pop();  // Удаляем первый элемент в очереди
        cout << temp->data << " "; // Печатаем значение первого элемента в очере

        if ( temp->left != NULL )
            q.push(temp->left);  // Вставляем  в очередь левого потомка

        if ( temp->right != NULL )
            q.push(temp->right);  // Вставляем  в очередь правого потомка
   }
}


void preorderPrint(TreeNode *root)
{
    if (root == NULL)   // Базовый случай
    {
       return;
    }
    cout << root->data << " ";
    preorderPrint(root->left);   //рекурсивный вызов левого поддерева
    preorderPrint(root->right);  //рекурсивный вызов правого поддерева
}
// Функция печатает значения бинарного дерева поиска в прямом порядке.
// Вместо печати первой инструкцией функции может быть любое действие
// с данным узлом

void inorderPrint(TreeNode *root)
```

```
{
    if (root == NULL)   // Базовый случай
    {
       return;
    }
    preorderPrint(root->left);   //рекурсивный вызов левого поддерева
    cout << root->data << " ";
    preorderPrint(root->right);  //рекурсивный вызов правого поддерева
}
// Функция печатает значения бинарного дерева поиска в симметричном порядке.
// То есть в отсортированном порядке

void postorderPrint(TreeNode *root)
{
    if (root == NULL)   // Базовый случай
    {
       return;
    }
    preorderPrint(root->left);   //рекурсивный вызов левого поддерева
    preorderPrint(root->right);  //рекурсивный вызов правого поддерева
    cout << root->data << " ";
}
// Функция печатает значения бинарного дерева поиска в обратном порядке.
// Не путайте обратный и обратноотсортированный (обратный симметричный).

void reverseInorderPrint(TreeNode *root)
{
    if (root == NULL)   // Базовый случай
    {
       return;
    }
    preorderPrint(root->right);  //рекурсивный вызов правого поддерева
    cout << root->data << " ";
    preorderPrint(root->left);   //рекурсивный вызов левого поддерева

}
// Функция печатает значения бинарного дерева поиска в обратном симметричном пор
// То есть в обратном отсортированном порядке
```

```
void iterativePreorder(TreeNode *root)
{
    if (root == NULL)
    {
       return;
    }
    stack<TreeNode *> s;  // Создаем стек
    s.push(root);  // Вставляем корень в стек
    /* Извлекаем из стека один за другим все элементы.
       Для каждого извлеченного делаем следующее
       1) печатаем его
       2) вставляем в стек правого! потомка
          (Внимание! стек поменяет порядок выполнения на противоположный!)
       3) вставляем в стек левого! потомка */
    while (s.empty() == false)
    {
        // Извлекаем вершину стека и печатаем
        TreeNode *temp = s.top();
        s.pop();
        cout << temp->data << " ";

        if (temp->right)
            s.push(temp->right); // Вставляем в стек правого потомка
        if (temp->left)
            s.push(temp->left);  // Вставляем в стек левого потомка
    }
}
// В симметричном и обратном итеративном обходах просто меняем инструкции
// местами по аналогии с рекурсивными функциями.
```

Надеюсь Вы не уснули, и статья была полезна. Скоро надеюсь последует продолжение ~~банкета~~ статьи.

Теги: алгоритмы, программирование, c++

Хабы: Программирование, C++, Алгоритмы

22 Карма 0 Рейтинг

Радченко Евгений @Procs
Программист C++

Подписаться

Комментарии 9

Вопрос. Как сбалансировать несбалансированное бинароное дерево и обычное дерево (неотсортированное)?

НЛО прилетело и опубликовало эту надпись здесь

НЛО прилетело и опубликовало эту надпись здесь

Записи вида O(V + E) вполне себе осмысленны и используются когда мы говорим о произвольных графах. В случае же дерева V = E + 1, что действительно делает подобную запись абсолютно бессмысленной. И никакой квадратичности, как видим :-)

Совершенно верно. В дереве, в отличие от графа количество ребер всегда равно количеству вершин минус 1. То есть количество ребер линейно зависит от количества вершин.

«Граф – это множество вершин и ребер» — общее 0_o?! Все в одну кучу что ли?! Может хотя бы: «упорядоченная пара множеств»? Не подумайте, что я придераюсь, но воспоминания, что дискретная математика не прощает таких ошибок (читать неточностей) ещё свежи в голове… Я указал лишь одно место, но по тексту много таких «деталей».

На практике, для обхода дерева в глубину (ну и для балансировки тоже) удобно, чтобы были восходящие связи — от детей к родителям. Тогда можно двигаться прямо по картинке, не используя стек.
Аналогично двунаправленному списку против однонаправленного.

В терминах графов, — это означает, что каждой вершине соответствуют наборы входящих и исходящих рёбер.

А для быстрого обхода в ширину нужны списки вершин на каждом ярусе дерева.
Либо мы можем свести задачу к последовательности обходов в глубину, опускаясь каждый раз на один ярус дальше. Это, конечно, даст квадратичное время (и логарифмическую память под стек, если рекурсивно) вместо линейной памяти под очередь.

Спасибо) разобрался благодаря вам в базовых вещах.

Во всех функциях preorderPrint вместо рекурсивных вызовов

Публикации

CyberexTech
9 часов назад

Делаем вентилятор умным или как улучшить микроклимат в ванной комнате с помощью домашней автоматизации

Средний 6 мин 5.9K

Кейс

+22 46 20

InlyIT
10 часов назад

Четыре принципа разработки ПО, которым я научился на горьком опыте

4 мин 3.5K

Перевод

+19 41 1

big-mdm
4 часа назад

Калькулятор резьбовых соединений для FreeCad

Простой 1 мин 500

Обзор

+12 13 5

Sergei2405
7 часов назад

Лазеры, сервопривод, WiFi MESH-сети и сноуборд. Часть 2

Простой 2 мин 646

+9 5 9

yoda_code
2 часа назад

Использование Laravel драйвера centrifugo для широковещания

Средний 21 мин 171

Туториал

+6 2 0

alexandrakilov
2 часа назад

Голографические принтеры

4 мин 345

+6 1 0

nikolz
7 часов назад

Как заработать 2.2 млн.руб себе и 18 млн.руб фирме без работников

Простой 2 мин 3.1K

+5 9 23

MGP5951
14 часов назад

Сквозная аналитика в HR

Простой 7 мин 1.1K

Из песочницы

+6 10 1

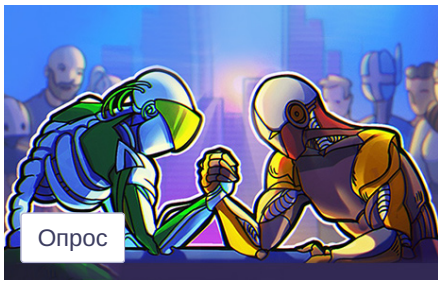
Кто победит в гонке IT-работодателей: участвуй в опросе от Хабра и Экопси

Опрос

Показать еще

МИНУТОЧКУ ВНИМАНИЯ

Оплата смартфоном через NFC в России: миф или реальность?



Какой IT-работодатель всех сильней? Проводим опрос

Ты веришь в судьбу, Нео? Я верю в скидки от промокодуса

ВАКАНСИИ

Fullstack developer (JS, C++)
от 300 000 ₽ · Сбер · Москва

Senior C++/Qt
от 240 000 ₽ · daobit · Санкт-Петербург

Программист Delphi/C++
от 300 000 до 500 000 ₽ · Базис-Центр · Коломна

C++/Qt/QML разработчик в команду Android
от 250 000 до 600 000 ₽ · 2GIS · Можно удаленно

Разработчик C++ (сетевое ПО)
от 300 000 ₽ · Fplus · Москва · Можно удаленно

Больше вакансий на Хабр Карьере

ЧИТАЮТ СЕЙЧАС

Хакеры опубликовали базу с 9,9 миллиардами утёкших паролей

10K 37

Как войти в Айти и надо ли вам туда в 2024 году

16K 54

Делаем вентилятор умным или как улучшить микроклимат в ванной комнате с помощью домашней автоматизации

5.9K 20

Федеральная торговая комиссия США предупредила ASRock, Gigabyte и Zotac, что наклейки Warranty Void If Removed незаконны

15K 163

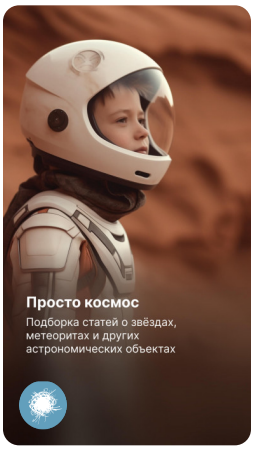
Stability AI стала бесплатной для пользователей и малого бизнеса

3.9K 13

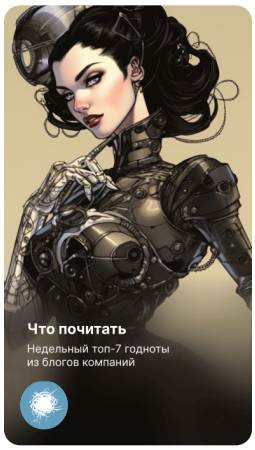
Кто победит в гонке IT-работодателей: участвуй в опросе от Хабра и Экопси

Опрос

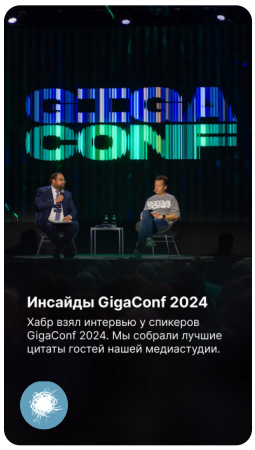
ИСТОРИИ



Статьи о космосе

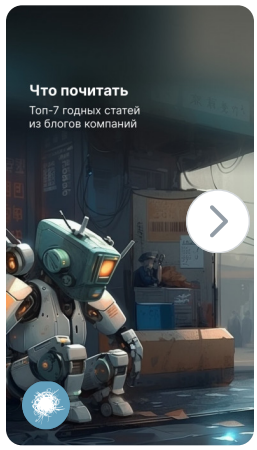


Годнота из блогов компаний



Инсайды GigaConf 2024

Новости нейромира



Топ-7 годный статей из блогов компаний

РАБОТА

Программист C++
111 вакансий

QT разработчик

10 вакансий

Все вакансии

БЛИЖАЙШИЕ СОБЫТИЯ

8 июля

Открытый урок «Дерево решений — простой и интерпретируемый ML-алгоритм»

Онлайн

Разработка

Больше событий в календаре

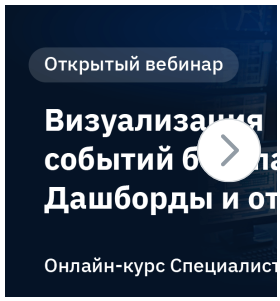
9 июля

Открытый вебинар «Пишем API онлайн-чата на Golang»

Онлайн

Разработка

Больше событий в календаре



9 июля

Вебинар «Виз событий безо Дашборды и SIEM»

Онлайн

Администрировани

Другое

Больше событий в ка

Ваш аккаунт

Войти

Регистрация

Разделы

Статьи

Новости

Хабы

Компании

Авторы

Песочница

Информация

Устройство сайта

Для авторов

Для компаний

Документы

Соглашение

Конфиденциальность

Услуги

Корпоративный блог

Медийная реклама

Нативные проекты

Образовательные программы

Стартапам

Настройка языка

Техническая поддержка