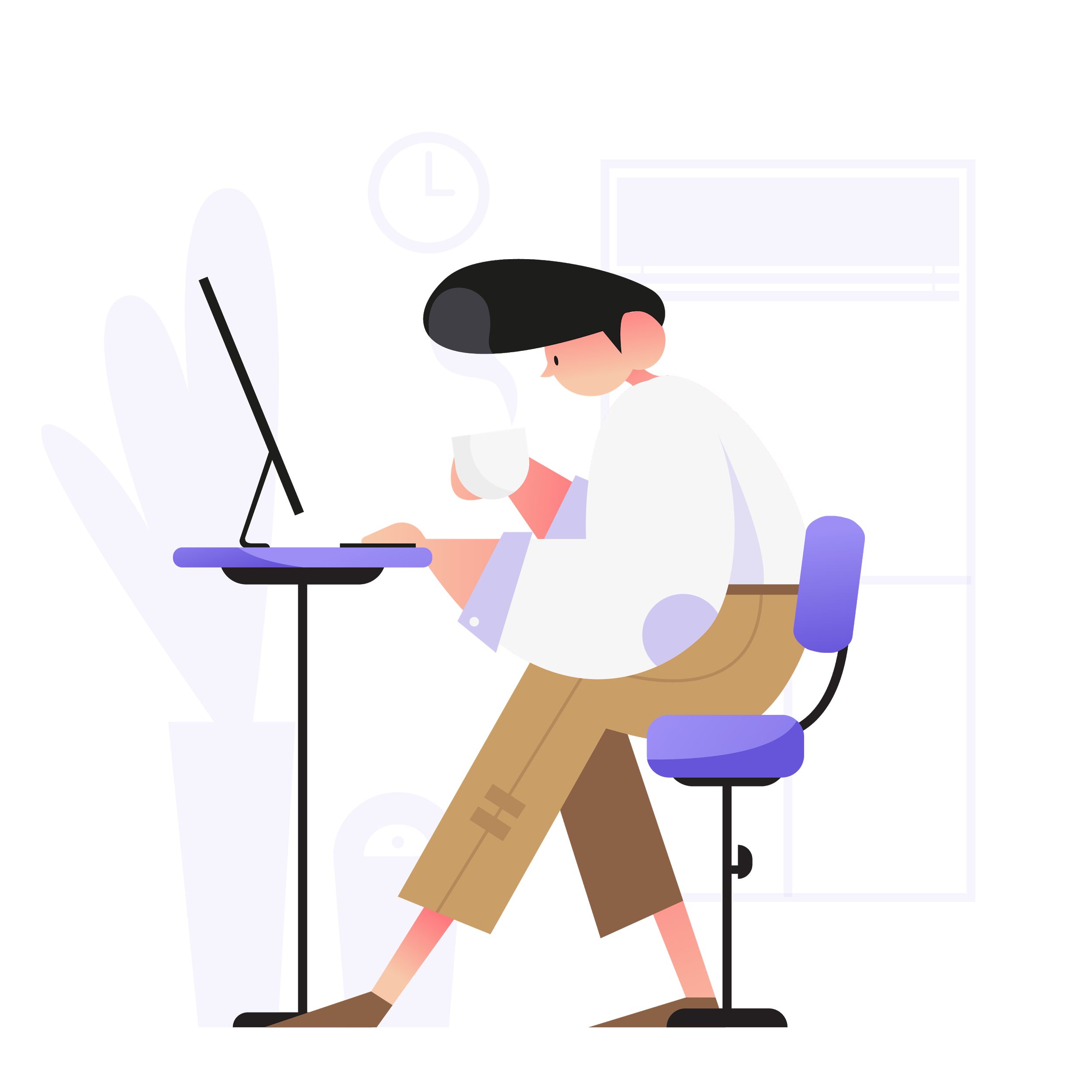
Алгоритмы и структуры данных на С#

Деревья, хеш-таблицы

Netcore 3.1



**На этом уроке**

1. Узнаем, что такое деревья поиска.
2. Освоим рекурсивный обход двоичного дерева.
3. Научимся построению сбалансированного дерева.
4. Узнаем, что такое двоичное дерево поиска и как происходит хранение двоичного дерева в массиве.
5. Узнаем особенности MD5 (Message Digest).
6. Рассмотрим применимость хеширования.
7. Выясним, чем различаются прямое связывание (хеширование с цепочками) и хеширование с открытой адресацией.
8. Рассмотрим пример работы с HashSet.

**Оглавление**

[Введение](#_bnpd2ayc0jmv)

[Структура данных «дерево»](#_q0mjjhh0x072)

[Деревья поиска](#_9qyj54z2s7vr)

[Использование динамических структур данных](#_7779n3lwl6h)

[Рекурсивный обход двоичного дерева](#_kldchlkwkz6r)

[Построение сбалансированного дерева](#_4d34og8)

[Двоичное дерево поиска](#_2s8eyo1)

[Хранение двоичного дерева в массиве](#_17dp8vu)

[Другие деревья](#_la8hbvn1wo02)

[Хеш-функция](#_x0y8xq2xlqis)

[MD5 (Message Digest)](#_t5en1rrjb2cl)

[Применимость хеширования](#_2sre81pjd9e8)

[Хеш-функции и хеш-таблицы](#_sv9laxcjnhxp)

[Прямое связывание (хеширование с цепочками)](#_z6sob37qsh7s)

[Хеширование с открытой адресацией](#_m8gc706geif5)

[HashSet](#_2ot7j9o74hi2)

[Заключение](#_8ixasns8c1ch)

[Практическое задание](#_x6cdw8xzm3sq)

[Дополнительные материалы](#_2jxsxqh)

[Используемая литература](#_7c9jrnh1eqkx)

# Введение

Дерево — это структура данных, которая очень часто встречается в базах данных и различных иерархических структурах, например UI. Её важно знать, чтобы в будущем более глубоко понимать, как работают базы данных и иерархические структуры.

Алгоритмы хеширования позволяют сократить время на проверку схожести объектов, организовать более быструю проверку наличия или получения объекта или получение самого объекта в хеш-таблицах. Что такое хеширование, хеш-таблицы и деревья, мы рассмотрим на этом уроке.

# 

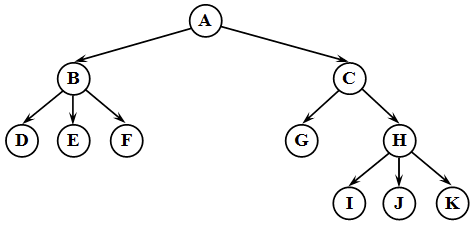
# Структура данных «дерево»

**Дерево** — это структура, отражающая иерархию, то есть отношения подчинённости, многоуровневые связи.

Дерево состоит из **узлов** и связей между ними — они называются **дугами**. Самый первый узел, расположенный на верхнем уровне, — это **корень дерева**. Конечные узлы, из которых не выходит ни одной дуги, называются **листьями**. Все остальные узлы, кроме корня и листьев, — промежуточные. Из двух связанных узлов тот, который находится на более высоком уровне, называется **родителем**, а другой — **сыном**. Корень — это единственный узел, у которого нет родителя, а у листьев нет сыновей.

Используются также понятия «предок» и «потомок». **Потомок** узла — это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, **предок** какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

На рисунке родитель узла F — это узел B, а предки узла F — это B и A, для которых узел F — потомок. Потомки узла A (корня дерева) — все остальные узлы.



**Высота дерева** — это наибольшее расстояние (количество дуг) от корня до листа. Высота дерева на рисунке равна трём.

Формально дерево можно определить следующим образом:

1. Пустая структура — это дерево.
2. Дерево — это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

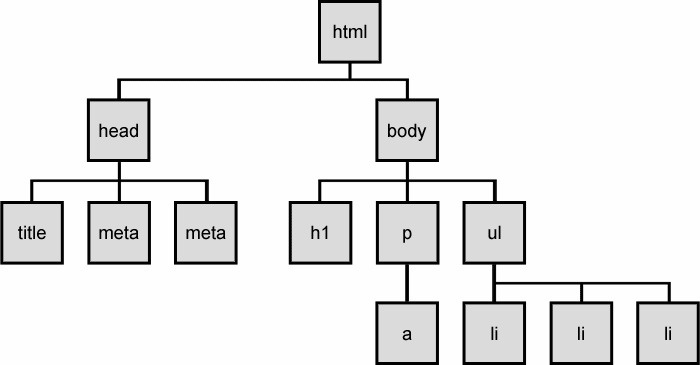
Здесь множество объектов (дерево) определяется через само это множество на основе простого базового случая — пустого дерева. Такой приём называется рекурсией. Значит, согласно этому определению, дерево — рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Деревья широко применяются в следующих задачах:

1. Сортировка данных.
2. Вычисление арифметических выражений (Expression tree).
3. Оптимальное кодирование данных (метод сжатия Хаффмана).
4. В тех задачах, где требуется часто изменять коллекцию и проводить по ней поиск.
5. В задачах частого поиска по диапазону.
6. В базах данных (B-tree).
7. В компиляторах языков программирования (синтаксическое дерево).

В виде дерева удобно хранить структуру организации компании или каталог частей, из которых состоит, скажем, автомобиль.

Можно особо отметить объектную модель документа (DOM) в HTML. При считывании HTML-страницы браузер, анализируя её код, выстраивает иерархию объектов в виде дерева.



Чаще всего в информатике используются двоичные (бинарные) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

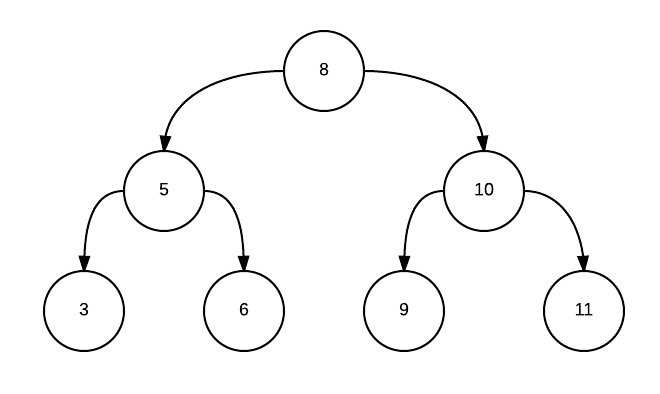
Двоичное дерево:

1. Пустая структура — это двоичное дерево.
2. Двоичное дерево — это корень и два связанных с ним отдельных двоичных дерева (левое и правое поддеревья).

## 

## Деревья поиска

Чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева. Например, как показано на рисунке:



Значения, связанные с узлами дерева, по которым выполняется поиск, называются ключами этих узлов. Кроме ключа узел может содержать множество других данных. Перечислим важные свойства дерева, показанного на рисунке:

1. Слева от каждого узла находятся узлы, ключи которых меньше или равны ключу этого узла.
2. Справа от каждого узла находятся узлы, ключи которых больше ключа этого узла.

Дерево, обладающее такими свойствами, называется **двоичным деревом поиска.**

Например, нужно найти узел, ключ которого равен 6. Начинаем поиск по дереву с корня. Ключ корня — 8, больше заданного, поэтому дальше нужно искать только в левом поддереве и так далее. Если при линейном поиске в массиве за одно сравнение отсекается один элемент, то здесь — сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально log2N, то есть алгоритм имеет асимптотическую сложность O(log2N).

Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто. Например, в базах данных.

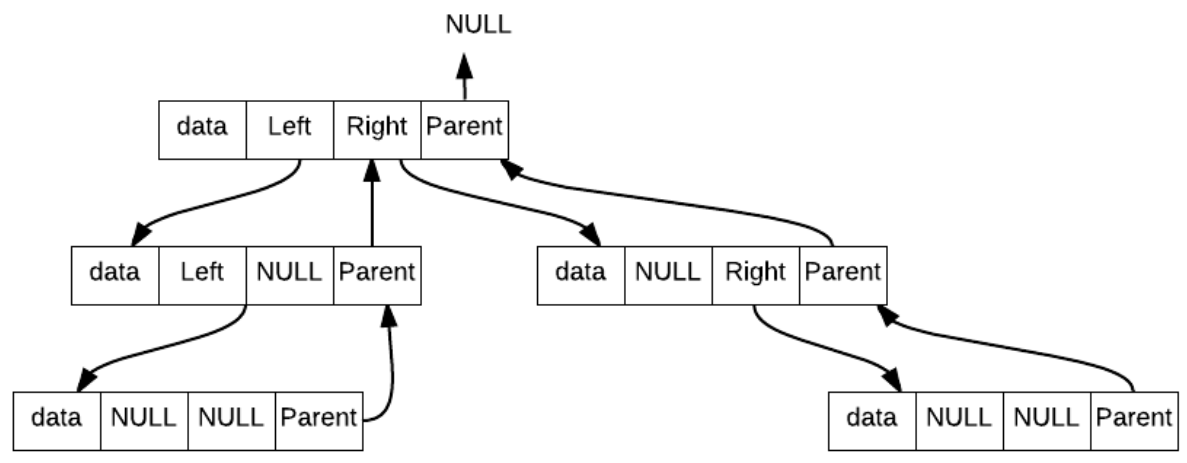
Индексы баз данных — это как раз и есть упорядочивание элементов с использованием структур типа B-tree. В дальнейшем это позволяет сократить время поиска в базе данных.

## 

## Использование динамических структур данных

Поскольку двоичное дерево — нелинейная структура данных, мы будем использовать связанные узлы. Каждый такой узел — это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). Также иногда добавляют указатель на предыдущий узел (Parent). У листьев нет сыновей, так что в этом случае в указатели будем записывать значение NULL (нулевой указатель).

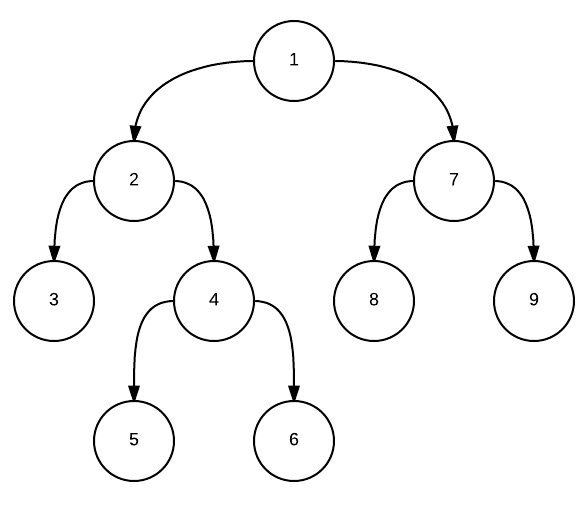
Пример дерева, состоящего из пяти таких узлов:



Структура, описывающая узел дерева:

|  |
| --- |
| public class Node<T> {  public T Data { get; set; }  public Node<T> Left { get; set; }  public Node<T> Right { get; set; }  public Node<T> Parent { get; set; } } |

## Рекурсивный обход двоичного дерева



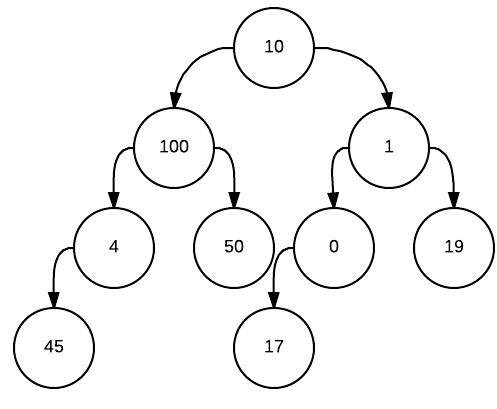
Реализация рекурсивного обхода наглядна:

|  |
| --- |
| public void PreOrderTravers(Node<T> root) {  if (root != null)  {  Console.WriteLine("", root.Data);  PreOrderTravers(root.Left);  PreOrderTravers(root.Right);  } } |

## Построение сбалансированного дерева

Предположим, что нужно построить дерево, значения в узлах которого — n чисел, считываемых из входного файла. Чтобы сделать задачу интереснее, будем строить дерево с n узлами, имеющее минимальную высоту. Чтобы получить минимальную высоту при заданном числе узлов, нужно размещать максимально возможное число узлов на всех уровнях, кроме самого нижнего. Очевидно, что этого можно достичь, распределяя новые узлы поровну слева и справа от каждого узла.

Пример сбалансированного дерева на рисунке ниже:



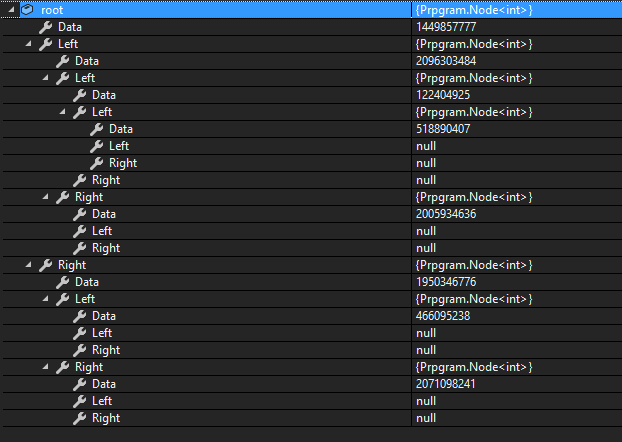
Правило равномерного распределения при известном числе узлов n лучше всего сформулировать рекурсивно:

1. Использовать один узел в качестве корня.
2. Построить таким образом левое поддерево с числом узлов nl = n / 2.
3. Построить таким образом правое поддерево с числом узлов nr = n – nl – 1.

Это правило реализуется рекурсивной процедурой, которая читает входной файл и строит идеально сбалансированное дерево.Дерево называется идеально сбалансированным, если для каждого узла число узлов в левом и правом поддеревьях отличается не больше, чем на единицу.

|  |
| --- |
| public class Node<T> {  public T Data { get; set; }  public Node<T> Left { get; set; }  public Node<T> Right { get; set; } }  *// Построение идеально сбалансированного дерева с n узлами* *// со случайными значениями* public static Node<int> Tree(int n) {  Node<int> newNode = null;  if (n == 0)  return null;  else  {  var nl = n / 2;  var nr = n - nl - 1;  newNode = new Node<int>();  newNode.Data = new Random().Next();  newNode.Left = Tree(nl);  newNode.Right = Tree(nr);  }  return newNode; } |

Пример построенного дерева:



## Двоичное дерево поиска

Дан файл с числами произвольной длины. Необходимо:

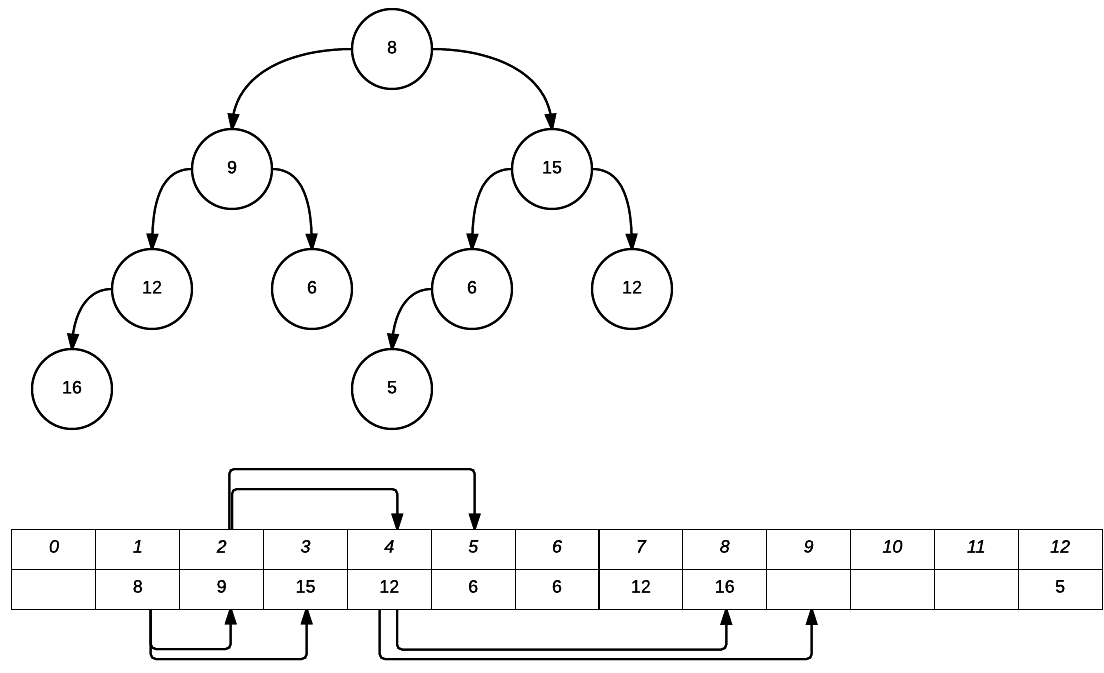
1. Считать файл с диска и построить по данным файла двоичное дерево поиска.
2. Реализовать функции считывания данных из файла в двоичное дерево и поиска заданного числа в двоичном дереве.

Вставка в двоичное дерево поиска:

|  |
| --- |
| public static Node<int> Insert(Node<int> head, int value)  {  Node<int> tmp = null;  if (head == null)  {  head = GetFreeNode(value, null);  return head;  }   tmp = head;  while (tmp != null)  {  if (value > tmp.Data)  {  if (tmp.Right != null)  {  tmp = tmp.Right;  continue;  }  else  {  tmp.Right = GetFreeNode(value, tmp);  return head;  }  }  else if (value < tmp.Data)  {  if (tmp.Left != null)  {  tmp = tmp.Left;  continue;  }  else  {  tmp.Left = GetFreeNode(value, tmp);  return head;  }  }  else  {  throw new Exception("Wrong tree state"); *// Дерево построено неправильно*  }  }  return head;  } |

## Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в массиве почти так же, как и списки. Вопрос в том, как сохранить структуру (взаимосвязь узлов). Если нумерация элементов начинается с 1, то сыновья элемента a[i] — a[2×i] и a[2×i+1]. Каждому числу 1, 2, 3, 4 будет соответствовать пара: 1 — (2,3), 2 — (4,5), 3 — (6,7) и т. д. На рисунке ниже показано, как можно сохранить часть двоичного дерева в массиве. Обратите внимание, что некоторые элементы остались пустыми. Это значит, что их родитель — лист дерева.



## 

## Другие деревья

Есть и другие виды деревьев:

* красно-чёрные;
* Splay;
* B-tree;
* B\*-tree;
* B+-tree;
* R-tree.

Все они в том или ином виде расширяют классические деревья за счёт введения дополнительной информации или методов вставки-удаления. Красно-чёрные деревья имеют обозначение цвета и за счёт хранения этой информации и дополнительных ограничений в худшем случае работают лучше, чем двоичное дерево поиска за log(N).

Семейство B-деревьев использует большее количество ссылок на потомков в узле, что позволяет уменьшить глубину дерева. Количество ссылок может быть различным. В модификациях для баз данных B-деревья содержат данные только в листовых узлах.

R-деревья — многомерные деревья, которые могут быть полезны для многомерных данных, особенно в задачах поиска по диапазону. Например, имея список координат магазинов, можно найти ближайшие от вас магазины, не перебирая все варианты сразу.

# 

# Хеш-функция

Часто при скачивании торрентов в их описании можно увидеть что-то наподобие A9990CD6D22117C1D4E3C5E9CF2FC5AA20305143, нередко с припиской MD5. Это хеш-код — результат, который выдаёт хеш-функция после обработки входящих данных. Расшифровать хеш MD5 очень сложно. Можно сказать, практически невозможно. Эта функция предназначена для преобразования входящих данных сколь угодно большого размера в результат фиксированной длины. Сам процесс такого преобразования называется хешированием, а результат — хешем или хеш-кодом. Ещё встречаются, но редко, термины «отпечаток» или «дайджест сообщения».

Есть масса различных алгоритмов, позволяющих превратить любой массив данных в последовательность символов определённой длины. Наибольшее распространение получил алгоритм под названием MD5, который был разработан ещё в 1991 году. Хотя MD5 несколько устарел и к использованию не рекомендуется, он до сих пор в ходу. Часто вместо слова «хеш-код» на сайтах просто пишут MD5 и указывают сам код.

## 

## MD5 (Message Digest)

Для функции, описанной выше, если известен результат, можно легко найти параметр, для которого результат будет тем же. А вот для функции MD5 это сделать не так-то просто. Если у нас есть только результат функции MD5, то мы не сможем найти параметр, для которого функция выдаст этот же результат. Речь даже не идёт об однозначном восстановлении параметра.

Например, MD5 используют для хранения паролей. Приведём ситуацию, когда хранение паролей в открытом виде опасно. Предположим, у вас есть сайт и база данных с логинами и паролями для доступа к нему. У вас возникла необходимость сделать локальную копию этого сайта, и нужно перенести логины и пароли пользователей. Можно просто перенести базу данных с логинами и паролями, но тогда будет две базы, что снижает безопасность. А можно посчитать для каждого пароля хеш-значение и перенести базу с ними.

Также MD5 можно использовать в качестве контрольной суммы. Предположим, необходимо куда-то скопировать файл или скачать его из интернета. Нет гарантий, что файл будет доставлен без повреждений. Перед отправкой можно посчитать MD5 от содержимого файла и передать результат вместе с ним. Затем посчитать MD5 от принятого файла и сравнить два результата. Если результаты различные, это означает, что файл или результат был испорчен при передаче.

## 

## Применимость хеширования

Цель таких хеш-функций, как MD5, — проверка целостности файла или поиск дубликатов. Да, всегда есть вероятность что существует два разных файла с одинаковым хешем. Поэтому при поиске дубликатов по MD5 следует сравнить два файла между собой, если хеши совпадают, чтобы исключить этот вариант. Этим же принципом пользуется Dictionary в C#. Но даже с таким ограничением хеш может значительно уменьшить количество проверок файлов один к одному и простых сравнений хешей.

Есть более продвинутые алгоритмы хеширования, например SHA-2. Алгоритмы хеширования обычно не обратимы, но этого от и них и не требуется. Например, из 16 байт MD5 вернуть файл размером 1 Гб, по которому считался MD5, будет невозможно, так как в 16 байтах содержится недостаточно информации для восстановления 1 Гб.

Хеш часто используется для хранения паролей. Привлекательность этого метода в том, что если украдут БД с паролями, то восстановить их будет затруднительно. Но это не значит, что невозможно, особенно учитывая, что пароль обычно не бывает больше 16 символов. Злоумышленники используют специальные методы, чтобы обратить хеш снова в пароль, — радужные таблицы. MD5 неустойчив к таким методам, поэтому чаще для хранения используют bcrypt или Argon2.

## 

## Хеш-функции и хеш-таблицы

Хеш-таблицы и хеш-функции позволяют быстро искать значения. Информация хранится не в отсортированном списке, а в особым образом организованной хеш-таблице и с помощью хеш-функций извлекается из неё.

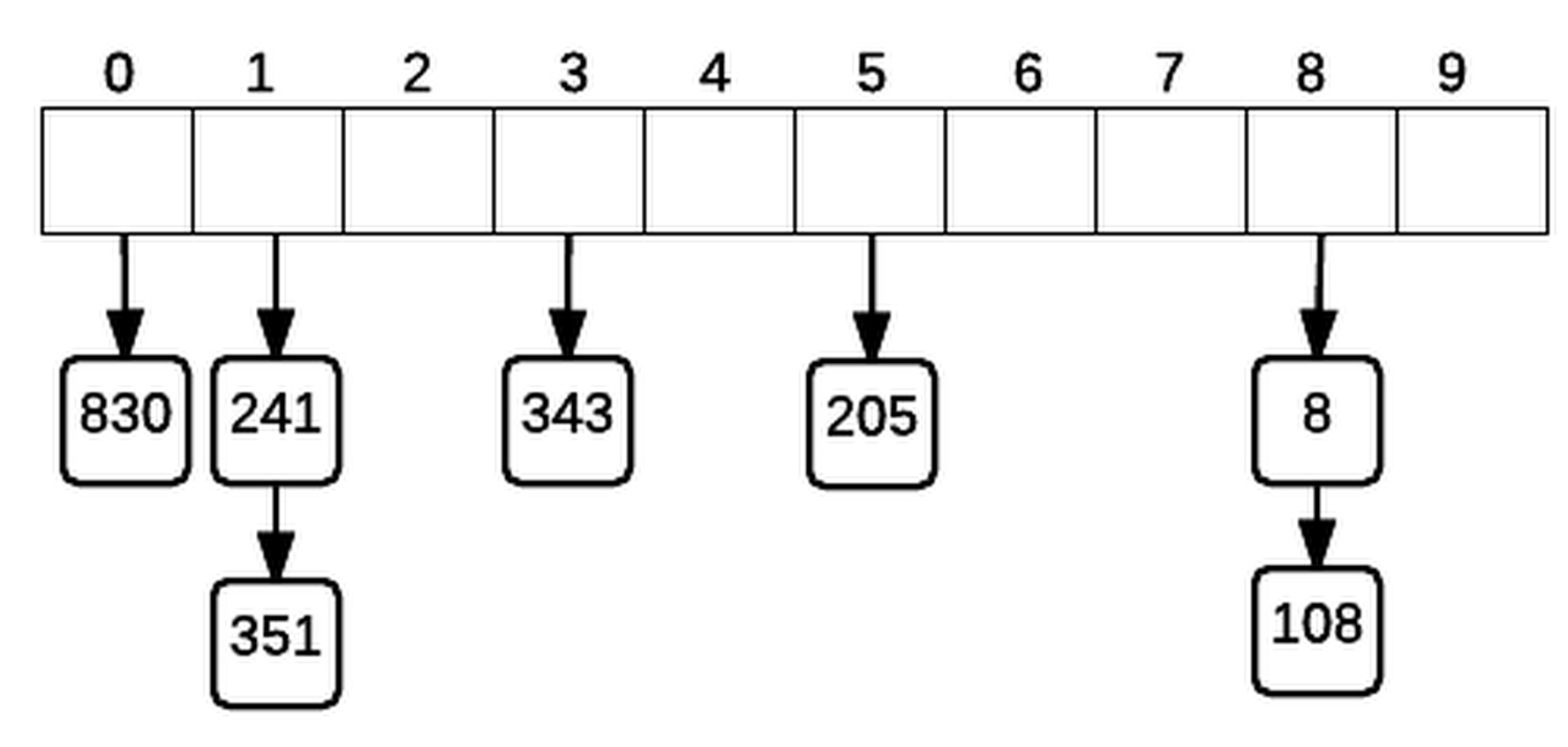
Предположим, у вас есть небольшая компания из 50 сотрудников с идентификационными номерами (ID) длиной в 4 цифры. Можно организовать массив из 100 элементов, в котором элемент массива будет соответствовать ID % 100 (остаток от деления на 100). Например, сотрудник с ID 3468 будет размещён в позиции 68, а сотрудник с ID 4677 — в позиции 77. Чтобы найти определённого работника, достаточно воспользоваться хеш-функцией **ID % 100** и выбрать элемент со значением, которое эта функция возвращает.

В реальности, если сотрудников окажется довольно много, среди них найдутся несколько, у которых ID будет соответствовать одному и тому же значению. Скажем, ID 4465 и 3165 должны будут занять в таблице позицию 65. Этот случай называется коллизией. Для работы с хеш-таблицами есть алгоритмы обработки коллизий.

## 

## Прямое связывание (хеширование с цепочками)

В хеш-таблицах с прямым связыванием значения ключей хранятся в специальных наборах записей, называемых блоками или bucket. Чаще встречается понятие bucket. Каждый из них — вершина связного списка, в котором находятся привязанные к блоку элементы.

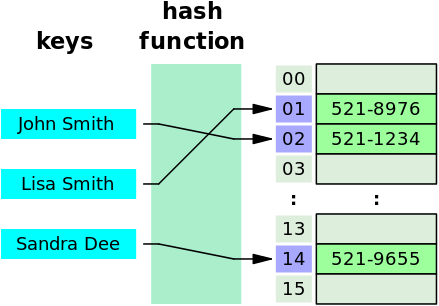


Обычно блоки расположены в массиве таким образом, что можно использовать простую функцию хеширования для определения их ключа. Например, если у вас N блоков, а ключи численные, свяжите ключ K с блоком под номером K % N. Для таблицы на рисунке выше N = 10.

## 

## Хеширование с открытой адресацией

В случае метода открытой адресации или, по-другому, закрытого хеширования все элементы хранятся непосредственно в хеш-таблице без использования связанных списков.



В отличие от хеширования с цепочками при использовании метода открытой адресации хеш-таблица может оказаться полностью заполненной, так что добавлять в неё новые элементы станет невозможно. Решением может быть динамическое увеличение размера хеш-таблицы с одновременной её перестройкой.

## 

## HashSet

Пример хеш-таблицы с прямым связыванием в C# — HashSet. HashSet хранит уникальные элементы и подходит для проверки наличия данных. Например, если вы собираетесь написать примитивную игру в города, то HashSet будет подходящей структурой данных, чтобы добавлять уже произнесённые города. В отличие от решения просто складывать названные города в массив, этот способ позволит не делать линейный поиск в массиве, если слово уже произнесено. Внутри HashSet представляет собой хеш-таблицу с прямым связыванием, то есть массив с бакетами. Ниже приведён пример работы с HashSet.

|  |
| --- |
| using System; using System.Collections.Generic;  public class User {  public string FirstName { get; set; }  public string SecondName { get; set; }   public override bool Equals(object obj)  {  var user = obj as User;    if (user == null)  return false;   return FirstName == user.FirstName && SecondName == user.SecondName;  }   public override int GetHashCode()  {  int firtsNameHashCode = FirstName?.GetHashCode() ?? 0;  int secondNameHashCode = SecondName?.GetHashCode() ?? 0;  return firtsNameHashCode ^ secondNameHashCode;  } }  class Program {  static void Main(string[] args)  {  var hashSet = new HashSet<User>();   var user = new User() { FirstName = "Barbara", SecondName = "Santa" };   hashSet.Add(user);   var searchUsser = new User() { FirstName = "Barbara", SecondName = "Santa" };   Console.WriteLine($"contains user {hashSet.Contains(user)}, contains searchUsser {hashSet.Contains(searchUsser)}");  } } |

Стоит отметить, что при работе с HashSet или Dictionary нужно переопределять методы GetHashCode и Equals, как было в примере выше. При работе с HashSet эти два метода активно используются самой структурой: GetHashCode — для получения хеш-кода объекта, на основе которого будет производиться вычисление индекса, Equals — для корректного сравнения объектов на тот случай, если произошла коллизия и два разных объекта выдали два одинаковых хеш-кода. Если не переопределить эти методы, то они не будут учитывать данные, которые содержатся в классе Equals. В данном случае будут просто сравнивать ссылки.

# 

# Заключение

Мы узнали, что деревья состоят из узлов и содержат связи с дочерними элементами, а деревья поиска хранят данные в отсортированном виде. Хеш-таблицы позволяют получать объекты за значительно более короткое время, чем если бы мы линейно искали объект в массиве.

# Практическое задание

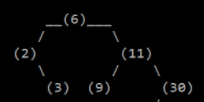
**1. Протестируйте поиск строки в HashSet и в массиве**

Заполните массив и HashSet случайными строками, не менее 10 000 строк. Строки можно сгенерировать. Потом выполните замер производительности проверки наличия строки в массиве и HashSet. Выложите код и результат замеров.

**2. Реализуйте двоичное дерево и метод вывода его в консоль**

Реализуйте класс двоичного дерева поиска с операциями вставки, удаления, поиска. Дерево должно быть сбалансированным. Также напишите метод вывода в консоль дерева, чтобы увидеть, насколько корректно работает ваша реализация.

Пример работы вывода:



# Дополнительные материалы

1. [Двоичное дерево поиска](http://learnc.info/adt/binary_search_tree.html).
2. [Хеширование](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

# Используемая литература

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.