

Глава 3

СТРУКТУРЫ ДАННЫХ

Цель

Обзор основных структур данных как реализаций АТД

Алгоритмическая задача

Математическая
модель

Неформальные
алгоритмы



Инструментарий
для обработки
данных

Алгоритм на псевдокоде
АТД

Программная реализация
Алгоритм + СД

Классификация типов данных (ТД) по сложности

Базовые ТД

ТД переменной – множество значений, которые она может принимать (логический, числовой, символьный и другие)

Составные ТД

Сконструированный ТД на основе базовых, составных, в том числе и самого себя (структура, запись и другие)

АТД – математическая модель + различные операторы, определенные в рамках этой модели

Основные понятия

Тип данных — конечное множество значений, каждое из которых требует для своего хранения фиксированное число ячеек памяти

Ячейка памяти — структурная единица компьютера, позволяющая хранить фиксированный объем информации, выражающийся целым числом байт

Структура данных

Структура данных — программная единица компьютера, описывающая способ организации ячеек памяти, хранящих данные одного и того же типа

Она **позволяет**:

- хранить необходимые наборы данных в памяти компьютера;
- эффективно ими управлять с помощью определенных операций

Некоторые из таких операций обычно реализуются на машинном уровне, поэтому их трудоемкость оценивается как $O(1)$ и не зависит от типа данных, с которым оперирует рассматриваемая структура

1. Элементарные структуры данных

Под элементарными структурами данных обычно понимают такие структуры, которые предоставляются непосредственно компьютером или моделируются с использованием доступных типов данных

Предполагается, что для их моделирования не используются другие элементарные структуры данных

Массивы и связанные списки

1.1 Массив как простейший механизм агрегирования ячеек

Это линейная и однородная элементарная структура данных, представляющая собой непрерывную упорядоченную последовательность ячеек памяти и предназначенная для хранения данных обычно **одного и того же типа**

Массив как отображение множества индексов во множество ячеек

Доступ к элементам массива осуществляется по **индексу** либо по набору индексов, где под индексом обычно понимается некоторое целочисленное значение

Многомерный массив

Массив A с m измерениями называют m -мерным и обозначают через $A[n_1, n_2, \dots, n_m]$,

где n_i — число элементов, которое может быть записано на уровне i , $i=1..m$.

Он позволяет хранить внутри себя $k = n_1 \cdot n_2 \cdot \dots \cdot n_m$

элементов с затратами памяти $k \cdot s = O(k)$,

где s — количество ячеек памяти, требуемое для хранения одного элемента

Формула доступа к элементам многомерного массива

Любой массив – это непрерывный участок памяти, поэтому многомерный массив технически представляет собой одномерный массив, который можно рассматривать как *многомерный параллелепипед*

$$(i_1, i_2, \dots, i_m) \Leftrightarrow ((\dots (i_1 \cdot n_2 + \dots) \cdot n_{m-2} + i_{m-2}) \cdot n_{m-1} + i_{m-1}) \cdot n_m + i_m,$$

$$0 \leq i_j < n_j, 1 \leq j \leq m.$$

Процедура вычисления индекса элемента массива

- Индексация с нуля

```
01:  Get Index Impl( $i_1, i_2, \dots, i_m$ )  
{ 02:      index = 0  
  { 03:      for j = 1 to m - 1 do  
    { 04:          index = (index +  $i_j$ ) *  $n_{j+1}$   
  { 05:      index = index +  $i_m$   
  06:      return index
```

Ступенчатый (*jagged*) массив

Многомерный массив можно неявно рассматривать как массив массивов одинакового размера

Но существует также понятие **рваного** (ступенчатого) массива, использование которого оправдано в том случае, если его элементами являются массивы произвольной длины. При этом следует явно обозначить, что он состоит из массивов

Некоторые из ЯП предоставляют возможности для создания рваных массивов как одного из типов данных

Пример рваного массива A[5][]

	1	2	3	4	5	6
1						
2						
3						
4						
5						

Временная сложность основных операций с элементами массива

- 1) **Чтение/запись** элемента требуют $O(1)$ времени, так как массив является структурой данных с произвольным доступом;
- 2) трудоемкость **поиска** элементов в массиве зависит от того, упорядочен он или нет: в случае **упорядоченного** массива поиск необходимых данных можно осуществить за $O(\log n)$ операций, **иначе** потребуется $O(n)$ операций в худшем случае;
- 3) **вставка/удаление** элементов в массив требуют $O(n)$ операций в худшем случае, так как изменение структуры массива требует перемещения элементов.

Вставка/удаление элементов только **на конце массива** осуществляется с трудоемкостью $O(1)$.

Недостатки массивов

Операции вставки и удаления элемента в массив осуществляются в худшем случае за $O(n)$ операций

Поэтому возникает необходимость в разработке структур данных, которые позволяют избавиться от данного недостатка и, как следствие, предоставят эффективные методы по включению/исключению элементов

Достоинства массивов

Экономное расходование памяти, требуемое для хранения данных,
быстрое время доступа к элементам массива

Вывод по целесообразности применения массивов

Применение массивов оправдано в том случае, если обращение к данным происходит по целочисленному индексу и в ряде случаев, не предполагающих модификацию набора данных путем изменения числа его элементов

Рандомизация массива

Один из способов рандомизации данных —
перестановка элементов входного массива

Будем предполагать, что массив содержит
числа от 1 до n

Задача: случайным образом переставить
элементы массива

Метод 1 ($O(n \log n)$)

Присвоить каждому элементу случайный приоритет, затем отсортировать в соответствии с назначенным приоритетом

PERMUTE-BY-SORTING(A)

1 $n = A.length$

2 Пусть $P[1..n]$ — новый массив

3 **for** $i = 1$ **to** n

4 $P[i] = \text{RANDOM}(1, n^3)$

5 Отсортировать A , используя P в качестве ключей сортировки

Утверждение 1

При отсутствии одинаковых приоритетов (вероятность не меньше $1-1/n$) можно получить случайную перестановку входных значений с равномерным распределением

Метод 2 ($O(n)$)

Перестановка «на месте»: i -й элемент массива заменяется на случайно выбранный из множества элементов промежутка $i..n$

Также генерирует равномерно распределенные случайные перестановки.

RANDOMIZE-IN-PLACE(A)

1 $n = A.length$

2 **for** $i = 1$ **to** n

3 Обменять $A[i]$ и $A[\text{RANDOM}(i, n)]$

1.2 Линейный список

Это линейная и однородная элементарная структура данных, предназначенная для хранения данных одного и того же типа, в которой порядок объектов определяется указателями

В отличие от массива, элементы списка хранятся необязательно в последовательных ячейках памяти: при помощи указателей ячейки памяти связываются в единое целое.

Программные реализации линейного списка

Предполагается, что элемент линейного списка описывается составным типом данных, который хранит в себе пользовательские данные некоторого типа и указатель на следующий элемент списка.

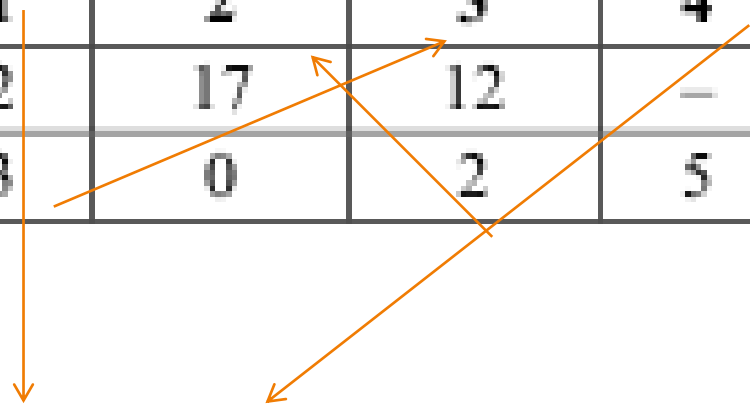
Различают концевые элементы: головной (начальный элемент списка) и хвостовой (конечный элемент списка)

Реализация списка [2,12,17]

1) с помощью двух массивов

i	1	2	3	4	5	6
$A[i]$	2	17	12	—	—	—
$B[i]$	3	0	2	5	6	0

ns=1, nf=4



Реализация списка [2,12,17] с помощью связанных компонентов

Каждый компонент списка состоит из двух ячеек памяти:

- первая содержит сам элемент либо указатель на его местоположение,
- вторая – указатель на следующий элемент



Базовые операции списка и их сложность (???)

- 1) Поиск некоторого заданного элемента с ключом
- 2) Поиск максимального (минимального) элемента
- 3) Включение элемента в список
- 4) Исключение элемента из списка
- 5) Формирование списка
- 6) Просмотр списка
- 7) Конкатенация (сцепление) списков
- 8) Разделение (расцепление) списков по заданному элементу

2. Абстрактные типы данных

Множество в математике и теории алгоритмов
(статичность и динамика)

Оптимальный способ реализации динамического
множества зависит от операций, которые им
должны поддерживаться



Абстрактные типы данных (АТД) –
математическая модель с определенным на
ней набором операций и правил поведения,
которые определяют ожидаемый результат
каждой из операций.

Операции над динамическим МНОЖЕСТВОМ

**Запросы – информация о
множестве**

- $\text{Search}(S, k)$
- $\text{Min}(S)$
- $\text{Max}(S)$
- $\text{Predecessor}(S, x)$
- $\text{Successor}(S, x)$

**Модифицирующие исходное
множество**

- $\text{Insert}(S, x)$
- $\text{Delete}(S, x)$

Инвариант АТД

Суть каждой из операций состоит в **изменении внутреннего состояния АТД**

Поведение определяет **инвариант** (существенное свойство), который должен сохраняться на протяжении всего жизненного цикла АТД вне зависимости от выполненных операций

Примерная классификация АТД

На основании поведения АТД можно разделить на:

- изменяемые и неизменяемые,
- персистентные (*сохраняющие свои предыдущие состояния и доступ к ним*) и неперсистентные

Персистентность может быть полная или частичная

Типы операций

Каждую из операций, определенных на АТД, можно отнести, в основном, к одному из трех типов – формирования, обновления, чтения

Операции **формирования** используются для создания конкретного экземпляра АТД с последующей инициализацией его внутреннего состояния

Выполнение операций **обновления** ведет к изменению внутреннего состояния АТД

Операции **чтения** – все иные операции

Интерфейс АТД

определяет ограниченный набор операций, доступный для манипулирования внутренним состоянием АТД

Основной целью интерфейса является предоставление возможности описания внутренних деталей АТД с использованием различных структур данных и других АТД

Вывод: интерфейс определяет набор операций и поведение, хотя конкретные реализации одного и того же АТД могут отличаться как по времени выполнения операций, так и по затратам памяти

Концепция контейнера

Под контейнером понимают АТД, предназначенный для хранения однотипных наборов данных и предоставляющий методы для работы с ними

Следует отличать от понятия контейнера понятие коллекции данных: если речь идет о конкретном АТД, то обычно говорят о контейнере, если же речь идет о наборе однотипных элементов, то говорят о коллекции. В этом смысле контейнер предоставляет возможности по сохранению коллекций данных

Вывод: большинство АТД суть контейнеры, основным предназначением которых является эффективное хранение и манипулирование коллекциями данных

Классификация контейнеров

- 1) По виду доступа к данным:
последовательные или произвольные
- 2) По способу хранения данных: в виде массива, связного списка или дерева
- 3) По форме итерирования: используют просмотр элементов коллекции данных в прямом, обратном или ином порядке
- 4) По форме организации данных:
последовательные и ассоциативные

Перечислитель (итератор) контейнера

Это неотъемлемый компонент контейнера, основное предназначение которого заключается в предоставлении набора операций, позволяющих осуществить просмотр содержимого контейнера

В зависимости от того, какая структура данных выбрана для сохранения коллекции данных, итераторы могут отличаться друг от друга для одного и того же АТД

2.1 АДД Список

Линейный АДД, предназначенный для хранения упорядоченных наборов данных одного и того же типа

Альтернативным определением списка является определение через последовательный контейнер с произвольным доступом к элементам коллекции данных

Основные операции над списком

- 1) Создание списка (как пустого, так и из массива элементов),
- 2) Вставка элемента в список,
- 3) Удаление элемента из списка,
- 4) Поиск элемента в списке

Реализация списка

В зависимости от того, какое поведение накладывается на операции над списком, в качестве внутренней структуры может быть выбран **массив** либо **связный список**

Реализация списка через массив лучшим образом подходит для тех случаев, в которых не предполагается осуществлять удаление элементов и известна емкость списка

Если же операция удаления необходима, то лучше прибегнуть к реализации через связный список

Классификация списков

Списки принято разделять на статические и динамические

Статические списки обычно не предполагают исключения элементов, поэтому их реализация обычно базируется на массивах

Наиболее же гибкая и эффективная реализация данного АТД базируется на основе двусвязного списка, поскольку все операции, кроме операции поиска, требуют в худшем случае $O(1)$ операций

Если же речь идет о неизменяемом списке, то в худшем случае вставка и удаление элементов будут требовать $O(n)$ операций

Двусвязный список

Связный список, в котором каждый из элементов содержит указатель как на следующий элемент списка, так и на предыдущий

Операции над двусвязным списком

Преимущество: позволяют итерироваться по элементам списка в **обоих направлениях**

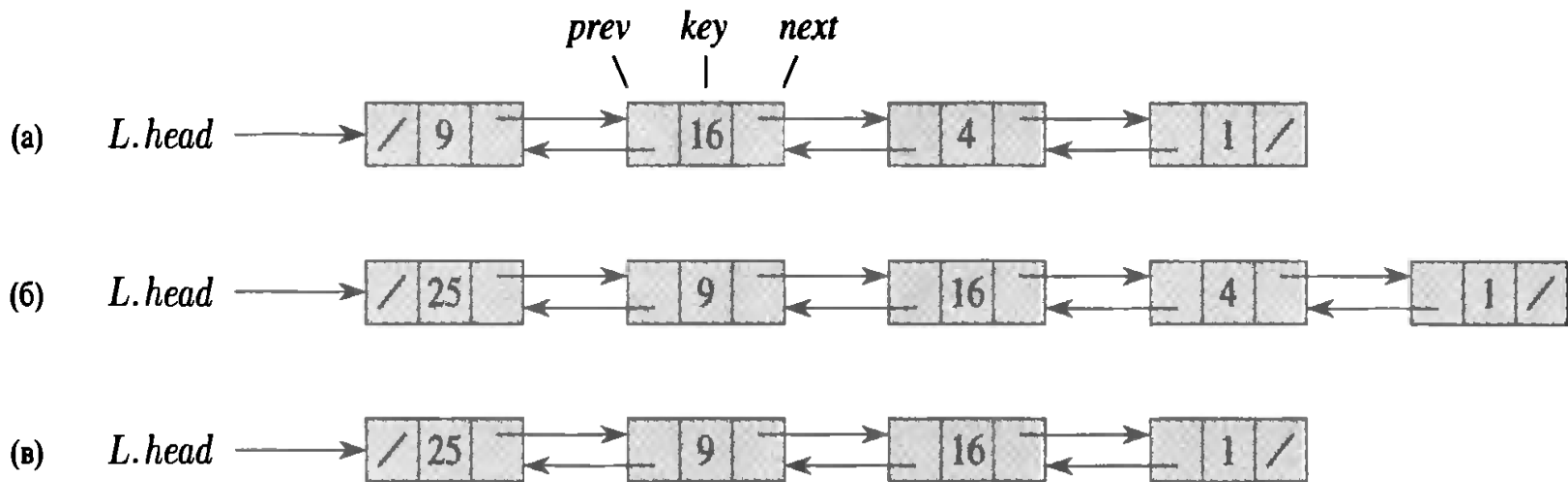
Операций над двусвязным списком незначительно отличаются от операций для односвязного списка

Трудоёмкость операций вставки, удаления, поиска составит $O(1)$ в случае изменяемой структуры данных

Для неизменяемого либо персистентного случая выполнение любых n операций над списком будет требовать времени порядка $O(n^2)$ из расчета, что работа начинается с пустого списка

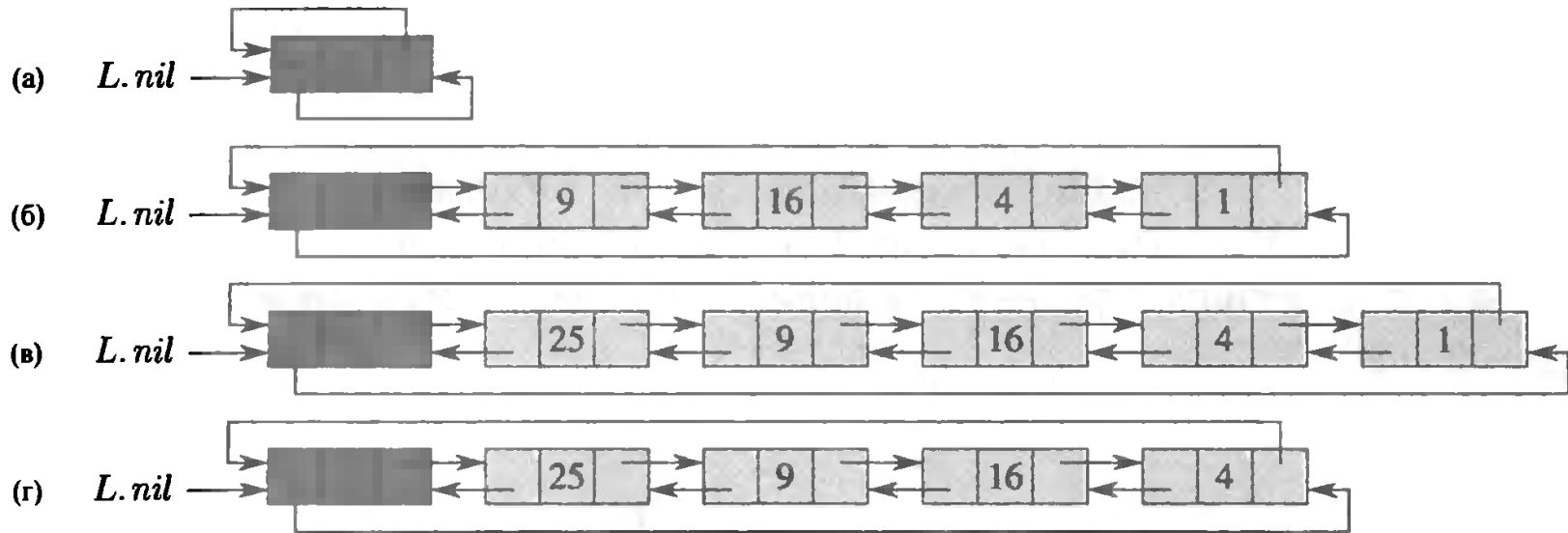
В этом случае можно говорить об амортизированном времени выполнения каждой из операций .

Схема двусвязного списка [9,16,4,1] и операций вставки и удаления



Ограничители

Циклический дважды связанный список с ограничителями



Реализация указателей и объектов

Реализация через массивы

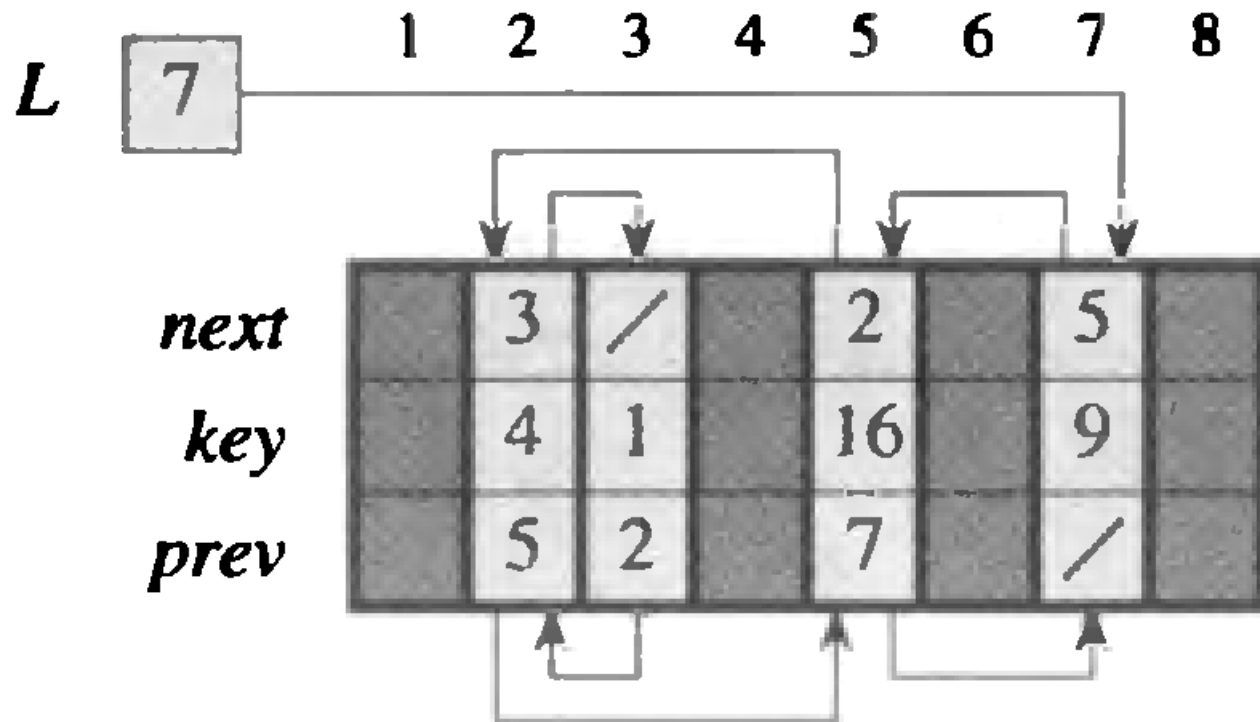
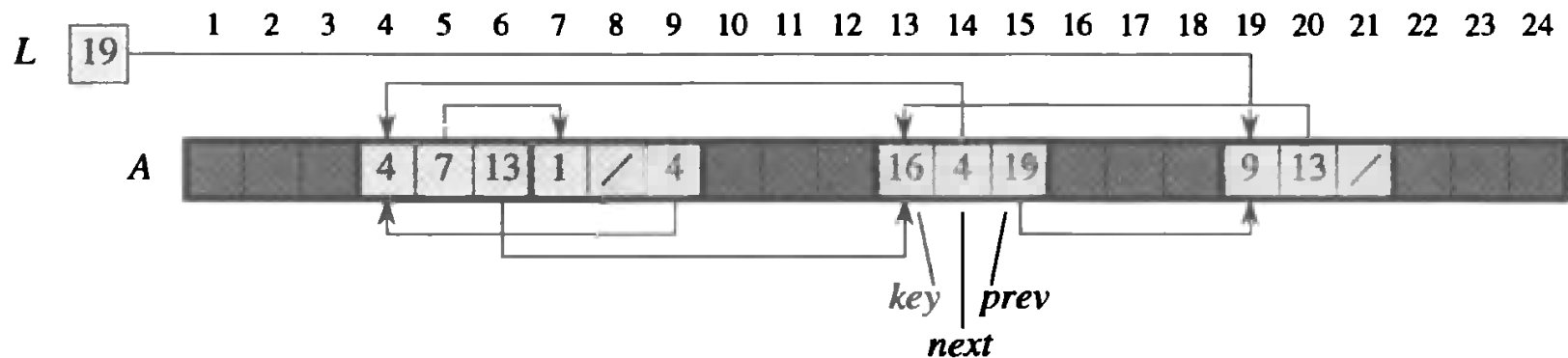


Схема реализации списка [9,16,4,1] через один массив



2.2. АТД Стек

Стек (магазин) – линейный АТД, доступ к элементам которого организован по принципу LIFO (last-in-first-out)

Поэтому для стека должна быть определена **точка входа** (вершина стека), через которую будет осуществляться вставка и удаление элементов

В частном случае стек можно рассматривать как **список**, работа с элементами которого происходит только на одном из концов списка

Реализация стека

Эффективные реализации стека базируются на массивах и на связных списках

Рекомендуется использовать реализацию через массив в тех случаях, когда требуется добиться высокой производительности и экономного расходования памяти

В общем же случае наиболее предпочтительным оказывается подход, базирующийся на идее односвязного списка: он представляется наиболее интересным как с точки зрения расширения функционала, поддерживаемого стеком, так и с точки зрения преобразования данного АТД в неизменяемый тип данных

2.3 АД Очередь

Очередь — линейный АД, доступ к элементам которого организован по принципу FIFO (first-in-first-out)

Это значит, что для очереди должны быть определены **две точки входа**: начало очереди, откуда происходит удаление элементов, и конец очереди, куда происходит добавление элементов

В частном случае очередь можно рассматривать как список, вставка элементов в который происходит на одном конце, а удаление элементов — на другом

Подходы к реализации очереди

Эффективные реализации очереди базируются на массивах и на связных списках

Рекомендуется использовать реализацию через массив тогда, когда требуется добиться высокой производительности и экономного расходования памяти, а также в случаях, когда заранее известно, что операции добавления и удаления элементов встречаются равновероятно

Реализация через 2 стека

Моделирование очереди через два стека интересно с точки зрения реализации интерфейса одного АТД через другой АТД, а также возможностей, предлагаемых данным подходом

В этом случае отдельного внимания заслуживает операция **балансировки стеков**, суть которой заключается в том, чтобы переместить данные из входного стека в выходной

В силу того, что элементы, добавленные в очередь, помещаются в каждый из двух стеков ровно один раз, то можно считать, что время, затраченное на выполнения 1 операций добавления и m операций удаления будет $O(1+m)$

Если же очередь реализована через массив либо связный список, то учетное время выполнения каждой из операций составит $O(1)$.

Двусторонняя очередь (дек)

Наиболее тривиальное определение дека — определение через список, в который запрещена вставка элементов в середину либо определение через очередь, где оба конца открыты для полного манипулирования данными

Дек является наиболее гибкой и, как следствие, наиболее сложной структурой данных, предоставляющей широкий доступ к формированию последовательности, в которой будут храниться необходимые наборы данных

Интуитивная реализация

Наиболее подходящей СД для реализации дека является **двусвязный список**, позволяющий производить все операции, кроме поиска, с трудоемкостью $O(1)$

Если же требуется эффективная реализация неизменяемого дека, то структура неизменяемого двусвязного списка не подойдет по причине неэффективности операций, выполняемых над ним (в этом случае каждая из операций вставки и удаления будет выполняться за $O(n)$)

Следовательно, требуется использовать другой подход, отличный от использования связных списков

Эффективная реализация

Идея подхода заключается в том, что любой дек можно представить в виде трех последовательностей элементов: левый элемент, средняя часть дека и правый элемент

Интерфейс дека

Как одиночный дек, так и пустой дек подчиняются общему интерфейсу дека

Так как дек является двусторонней очередью, то отдельно стоит рассмотреть операции вставки в начало / конец дека и удаления из начала / конца дека

Операции вставки/удаления

При реализации этих операций с рекурсивными вызовами время $O(n^2)$

Существуют модификации данного подхода, позволяющие улучшить время работы процедур до $O(\log n)$ и даже до $O(1)$

2.4 АТД Словарь

Под словарем будем понимать АТД, предназначенный для хранения множеств, состоящих из пар «ключ — значение ключа»

Словари иногда называют **отображениями** (множество ключей отображается на множество значений) или **ассоциативными массивами** (множество значений ассоциируется с множеством ключей)

К основным операциям над словарями относятся **операции:**

- создания словаря,
- вставки / удаления элементов,
- поиск значения по ключу

Трудоёмкость базовых операций для различных структур данных

Трудоёмкость словарных операций непосредственно зависит от трудоёмкости их выполнения над используемой СД:

- 1) массив или список – $O(n)$;
- 2) поисковое дерево – $O(\log n)$;
- 3) двоичные векторы – $O(1)$;
- 4) хеш-таблица – $O(1)$ - $O(n)$

Замечание. При использовании хеш-таблиц (ХТ) множество ключей не обязательно линейно упорядочено (когда элементы можно сравнить между собой), для большинства древовидных структур данных это условие является **необходимым**

1) ХТ как эффективная СД для реализации словаря

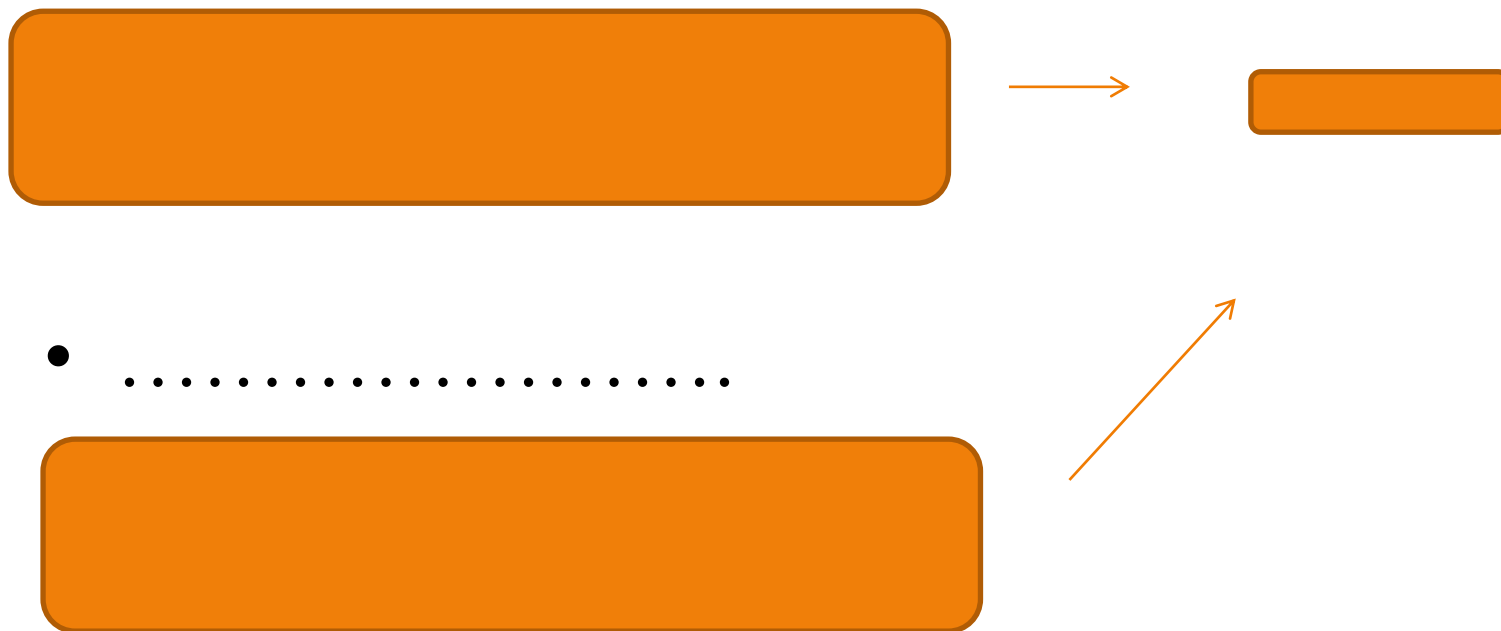
Семантика слова **to hash** (порубить, перемешать, запутать)



РАСПУТАТЬ!

Хеширование

Хеширование (коротко) – преобразование массива данных произвольной длины в массив заданной мощности



Индексирование по ключу

		i			
		i			

Метод поиска элемента с использованием индексирования по ключу позволяет к нему обратиться **немедленно**.

Значение ключей – индекс массива
(нет операций сравнения!)

Ключи – различные целые числа из того же диапазона, что и индексы.

Базовые словарные операции и хеширование

Хеширование эффективно применяется при расширенном варианте **поиска** с использованием индексирования по ключу в том случае, когда необходимо выполнять только словарные операции **добавления-удаления и поиска** элемента некоторого динамического множества

Что такое хеш-таблица ?

1. Обобщение обычного массива с доступом к любому элементу за $O(1)$
2. Эффективная структура данных для реализации словарей (неупорядоченных множеств, ассоциативных массивов (map))

НО!

Не обеспечивает эффективной реализации операций упорядоченного перебора, поиска минимального (min) элементов

Формы хеширования

Открытое (внешнее, с помощью цепочек)

Закрытое (внутреннее, с помощью открытой адресации)

Таблицы с прямой адресацией

$$U = \{0, 1, \dots, u - 1\}$$

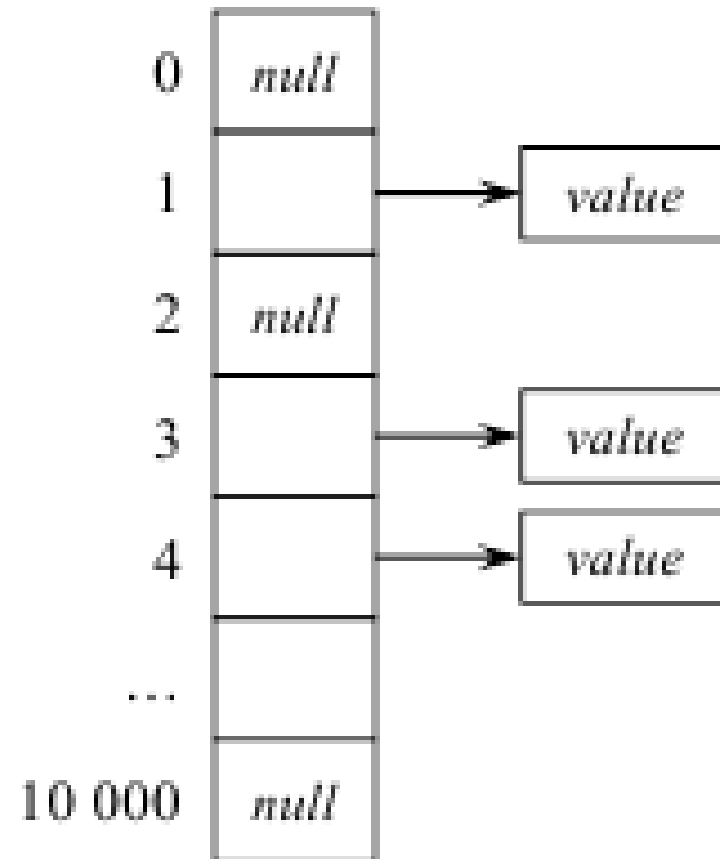
Таблицей с прямой адресацией называется массив $T[0..u-1]$, в котором каждой ячейке соответствует единственный ключ из множества U

Если ключ отсутствует, то соответствующая ячейка имеет значение *null*(nil, Nil)

Пример

$U = \{0, 1, \dots, 10000\}$

$K = \{1, 3, 4\}$



Трудоёмкость

Память и инициализация ячеек с пустыми значениями – $O(u)$ операций

Реализация словарных операций – $O(1)$

Проблема и ее возможное решение

Если $|U| \gg |K|$, то применение ТПА становится неэффективным и даже невозможным

Хеш-таблицы позволяют:

- снизить требования к оперативной памяти до $O(|K|)$;
- при этом поиск элемента остается $O(1)$ – в среднем, и $O(|K|)$ – в худшем случаях

Сравнение ТПО и ХТ

$$i=k$$

		k			
--	--	-----	--	--	--

$$h:U \rightarrow \{0,1,\dots,m-1\}$$

$$h(k) = i$$

$$i=h(k)$$

		k			
--	--	-----	--	--	--

Основная идея хеширования

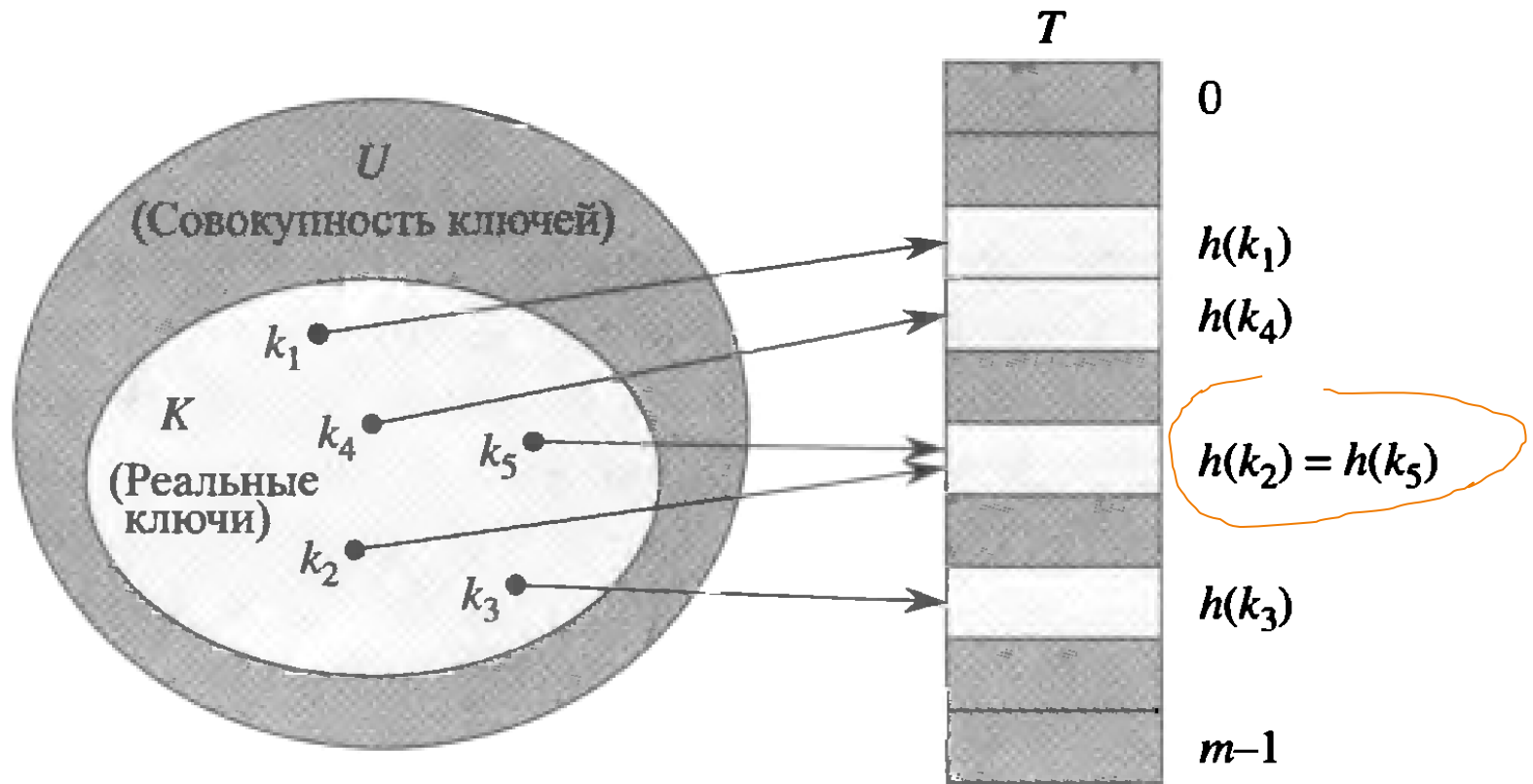
Исходное множество (возможно, бесконечное)
разбивается на конечное число классов

Для этих m классов, пронумерованных от 0 до $m-1$, строится хеш-функция h такая, что для любого элемента k исходного множества функция $h(k)$ принимает целочисленное значение из сегмента $\{0, \dots, m-1\}$, соответствующее классу, которому принадлежит элемент k

- $h(k)$ — хеш-значение ключа k (хеш, хеш-код)


Ключ k хешируется в ячейку $h(k)$

Пример хеширования



Коллизии

(нарушение инъективности)

$|U| > m$  коллизии — это совпадение хеш-значений нескольких ключей,

т.е. несколько ключей хешируются в одну ячейку

Это приводит к задачам определения «хороших»:

1. хеш-функций,
2. форм хеширования

для оптимального *разрешения коллизий*

(от них избавиться нельзя, только минимизировать!)

Методы разрешения коллизий,

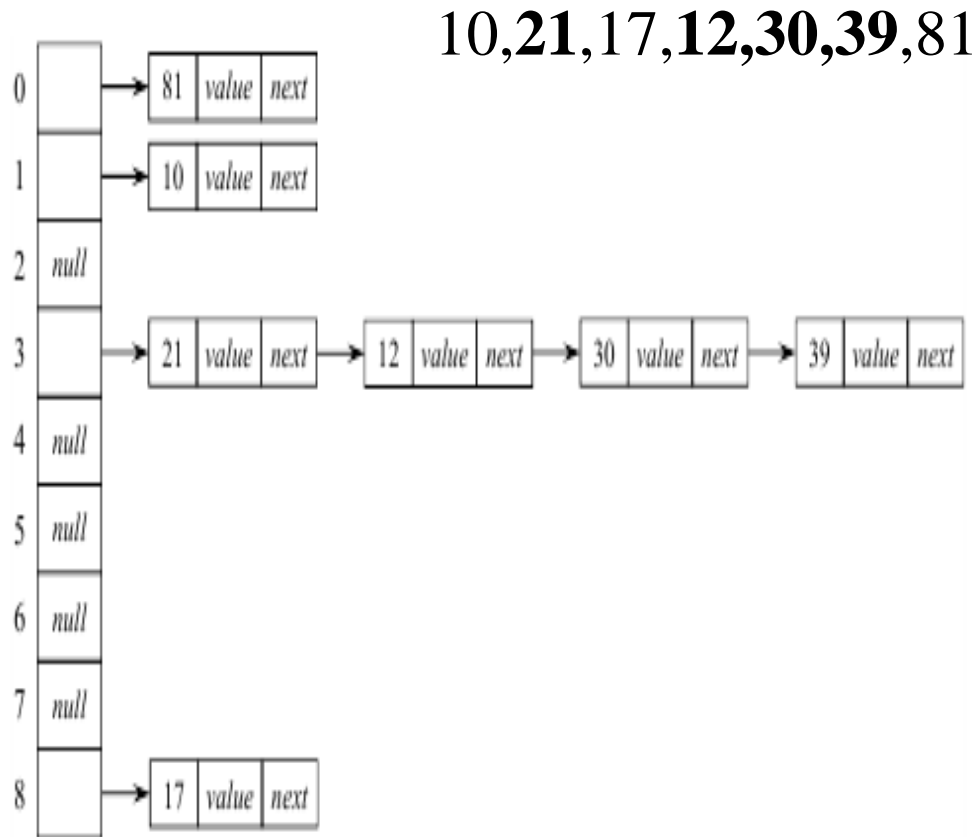
связанные с правилами хранения элементов с одинаковыми кодами

1. Открытое хеширование (метод цепочек)
2. Закрытое хеширование (открытая адресация)

Пример для сравнения форм хеширования

h -остаток от деления на 9 ($h=k\text{mod}9$)

Метод цепочек



Открытая адресация

	<i>key</i>	<i>value</i>	<i>state</i>
0	81		<i>Filled</i>
1	10		<i>Filled</i>
2			<i>Empty</i>
3	21		<i>Filled</i>
4	12		<i>Filled</i>
5	30		<i>Filled</i>
6	39		<i>Filled</i>
7			<i>Empty</i>
8	17		<i>Filled</i>

2) Метод цепочек

Идея: элементы, которым соответствует одно и то же хеш-значение, связываются в сегменте в список

Массив T , называемый таблицей сегментов, имеет размерность m

$T[i]$ — адрес (указатель) начала (или конца) списка ключей, имеющих хеш-значение, равное i

Если отсутствуют ключи с хеш-значением i , то в $T[i]$ хранится `null` — указатель на пустой список.

Реализация базовых операций

Вставка. Для добавления элемента в хеш-таблицу вычисляется хеш-код h заданного ключа key . Найденный хеш-код позволяет получить доступ к связанному списку $T[h]$ и добавить новый узел в его начало.

Вычислительная сложность вставки элемента в хеш-таблицу равна $O(1)$

Поиск. Для заданного ключа key вычисляется его хеш-код h . После чего в связанном списке $T[h]$ осуществляется поиск узла, содержащего заданный ключ key .

Удаление. Для того чтобы удалить из хеш-таблицы узел с ключом key , необходимо вычислить его хеш-код h и удалить из списка $T[h]$ соответствующий узел.

Вычислительная сложность поиска и удаления

Время выполнения поиска и удаления элемента из хеш-таблицы определяется длиной соответствующего связного списка. В худшем случае все ключи могут попасть в один связный список длины $\Theta(n)$ – случай неэффективной хеш-функции. Следовательно, вычислительная сложность операций поиска и удаления элемента из хеш-таблицы в худшем случае равна $\Theta(n)$.

Инициализация и удаление таблицы

Как и таблицы с прямой адресацией, хеш-таблицы требуют предварительной инициализации: выделения памяти под массив $T[0..m-1]$ и записи в каждую ячейку значения *null*. Эта операция выполняется за время $\Theta(m)$.

Для освобождения памяти из-под всей хеш-таблицы требуется пройти по всем ячейкам хеш-таблицы и удалить узлы всех непустых связанных списков. На это требуется $\Theta(m + n)$ операций.

Неэффективные операции

Поиск минимального, максимального,
предшествующего и последующего
элементов – $O(m+n)$

Равномерное хеширование

Средняя стоимость поиска зависит от того, насколько равномерно хеш-функция распределяет хеш-значения по цепочкам таблицы

Гипотеза равномерного хеширования заключается в предположении, что каждый из n элементов множества может попасть в любую из m цепочек хеш-таблицы с равной вероятностью и независимо от того, куда попал другой элемент

Коэффициент заполненности

Пусть $T[0..m-1]$ – хеш-таблица с цепочками, имеющая коэффициент заполнения $\alpha = n/m$,
и хеширование равномерно

Временная сложность неудачного поиска $\frac{1}{m} m\alpha = \alpha$

- **ТЕОРЕМА 1.** В ХТ с разрешением коллизий методом цепочек при равномерном хешировании среднее время поиска **отсутствующего** элемента (включая время на вычисление хеш-функции) равно $\Theta(1+\alpha)$.

Доказательство. В предположении простого равномерного хеширования любой ключ k , который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка $T[h(k)]$, ожидаемая длина которого — $E[n_{h(k)}] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α , а общее время, необходимое для поиска, включая время вычисления хеш-функции $h(k)$, равно $\Theta(1 + \alpha)$. ■

Временная сложность удачного поиска (удаления)

ТЕОРЕМА 2. Среднее время успешного поиска в хеш-таблице с цепочками и равномерным хешированием равно $\Theta(1+\alpha)$.

Доказательство (1-й способ -грубо)

Если ключ key присутствует в хеш-таблице, то он может с одинаковой вероятностью $1/\alpha$ находится в любом из α узлов связного списка $T[hash(key)]$. Если ключ находится в первом узле, то для его поиска требуется одно сравнение, если во втором – выполняется два сравнения и т. д. Математическое ожидание числа операций сравнения ключей при поиске узла равно

$$\frac{1}{\alpha} \cdot 1 + \frac{1}{\alpha} \cdot 2 + \dots + \frac{1}{\alpha} \cdot \alpha \approx \frac{1 + \alpha}{2}.$$

Следовательно, вычислительная сложность успешного поиска и удаления элемента из хеш-таблицы в среднем случае также равна $O(1 + \alpha)$.

Доказательство (2-й сп.-более
корректно) при условии добавления
элемента в начало цепочки

$$X_{ij} = I\{h(k_i) = h(k_j)\}. E$$

$$\Pr\{h(k_i) = h(k_j)\} = 1/m$$

$$= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}] \right)$$

$$= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right)$$

$$= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i)$$

$$= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right)$$


$$= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right)$$

$$= 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.$$

Вывод

Если количество ячеек в ХТ сделать пропорциональным числу всех элементов, то $n = O(m)$


$$\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$$

Значит, в среднем случае все базовые словарные операции в ХТ выполняются за время $O(1)$

3) Эвристики для выбора хеш-функции

Время выполнения операций хеш-таблицы во многом определяется эффективностью используемой хеш-функции. Рассмотрим основные требования, предъявляемые к ним.

1. *Равномерное хеширование* (uniform hashing). Хеш-функция должна как можно равномернее распределять входные ключи по m ячейкам хеш-таблицы (обеспечивать минимум коллизий).

2. *Детерминированность* (determinism). Для заданного ключа key хеш-функция должна всегда возвращать один и тот же хеш-код $hash(key)$.

3. Время вычисления хеш-функции не должно зависеть от количества n ключей, находящихся в хеш-таблице. Число операций, выполняемых хеш-функцией, должно зависеть только от длины ключа key . Поэтому считается, что вычислительная сложность хеш-функции равна $O(1)$ или $O(|key|)$.

Рассмотрим два основных подхода к построению хеш-функций: *метод деления* (division method) и *метод умножения* (multiplication method).

Деление с остатком

Пусть k – некоторый ключ, а m – число различных хеш-значений

Тогда хеш-функция определяется следующим образом:

$$h(k) = k \bmod m .$$

Пример $m=10, k=136 \longrightarrow h(136)=6$

Выбор числа сегментов

В качестве m предпочтителен выбор простого числа, далеко отстоящего от степени 2. При этом некоторых значений m стоит избегать:

1) если $m = 2^p$, то $h(k)$ – это p младших битов числа k , и если нет уверенности в том, что все комбинации младших битов будут встречаться с одинаковой частотой, то степень двойки в качестве числа m не выбирают;

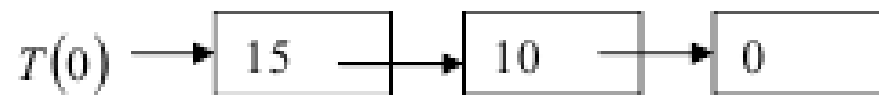
2) если ключи возникают как десятичные числа, то нежелательно выбирать в качестве m степень 10, так как в этом случае окажется, что часть цифр ключа уже полностью определяет хеш-значение;

3) если ключи – числа в системе счисления с основанием 2^p , то нежелательно брать $m = 2^p - 1$, поскольку при этом одинаковое хеш-значение имеют ключи, отличающиеся лишь перестановкой 2^p -ичных цифр.

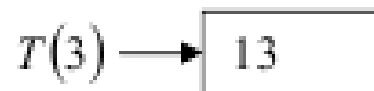
Пример

Используя открытое хеширование с хеш-функцией $h(k) = k \bmod 5$, сформировать хеш-таблицу T для последовательности ключей: 0, 10, 15, 13, 7, 17.

Решение. Вычислим хеш-значения для заданной последовательности ключей: $h(0) = 0 \bmod 5 = 0$; $h(10) = 10 \bmod 5 = 0$; $h(15) = 15 \bmod 5 = 0$; $h(13) = 13 \bmod 5 = 3$; $h(7) = 7 \bmod 5 = 2$; $h(17) = 17 \bmod 5 = 2$.



$T(1) = nil$



$T(4) = nil$

2. *Умножение.* Зафиксируем константу A в интервале $0 < A < 1$. Тогда хеш-функция определяется следующим образом:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor,$$

где $k \cdot A \bmod 1$ – дробная часть числа $k \cdot A$. Обычно в качестве m выбирают степень двойки. Утверждают, что наиболее удачное значение константы A есть 0,6180339887 («золотое сечение»).

$$kA \bmod 1 = \{kA\}$$

Пример 5.2. Пусть $k = 123456$, $m = 10000$ и $A = 0,6180339887$, тогда

$$h(k) = \lfloor 10000 \cdot (123456 \cdot 0,6180339887 \bmod 1) \rfloor = 41.$$

Замечание. Этот метод работает при любых m .
Особенно он удобен на практике для степеней 2,
когда метод деления не рекомендован.

Вычислительная сложность операций на основе метода цепочек

Операция	Средний случай	Худший случай
INITIALIZE($T[0..m - 1]$)	$\Theta(m)$	$\Theta(m)$
INSERT($T[0..m - 1], key, value$)	$O(1)$	$O(1)$
LOOKUP($T[0..m - 1], key$)	$O(1 + n/m)$	$O(n)$
DELETE($T[0..m - 1], key$)	$O(1 + n/m)$	$O(n)$

Дополнительные операции

MIN($T[0..m - 1]$)	$\Theta(m + n)$	$\Theta(m + n)$
MAX($T[0..m - 1]$)	$\Theta(m + n)$	$\Theta(m + n)$
PREDECESSOR($T[0..m - 1], key$)	$\Theta(m + n)$	$\Theta(m + n)$
SUCCESSOR($T[0..m - 1], key$)	$\Theta(m + n)$	$\Theta(m + n)$
DELETEHASHTABLE($T[0..m - 1], key$)	$\Theta(m + n)$	$\Theta(m + n)$

4) Открытая адресация

При открытой адресации в ХТ хранятся непосредственно ключи, а не указатели на заголовки списков

Поэтому в **каждом сегменте** хранится только **одно число**

Применяется методика повторного хеширования: если мы пытаемся поместить элемент в сегмент $h(k)$, который уже занят другим ключом, то выбирается последовательность других номеров сегментов, куда можно поместить данное ключевое значение

Каждое из местоположений последовательно проверяется, пока не будет найдено **свободное место**

Если свободных мест нет, то таблица целиком заполнена и элемент добавить нельзя

Проблема свободных мест

Случай 1. Удаление элементов не выполняется

Очередной элемент вносится в ячейку с номером своего хеша. Если она занята, то в **первую свободную**. Значит, дойдя до пустой ячейки при поиске ключа, мы дальше его не ищем

Случай 2. Удаление элементов реализуется

Тогда ячейки с удаленными элементами помечают, чтобы при поиске нужного ключа

— не возникала неопределенность по его наличию и

— чтобы освободившиеся места можно было использовать повторно

Реализация операций

Для выполнения вставки, поиска и удаления элементов необходимо определять, является ли ячейка таблицы **свободной** (первоначально вся ХТ пуста).

Это может быть реализовано несколькими способами:

- в каждой ячейке хранить дополнительное поле, информирующее о ее состоянии (свободна, занята, удалена и др.);
- использовать специальное значение ключа для обозначения незанятой ячейки;
- хранить в ячейке указатель на запись с ключом и данными

Стратегии исследований на наличие свободных мест

- 1) линейная;
- 2) квадратичная;
- 3) двойное хеширование

1) Функция линейной последовательности проб

$$h(k, i) = (h'(k) + i) \bmod m$$

$T[hash(key)], T[hash(key) + 1], \dots, T[m - 1], T[0], \dots, T[hash(key) - 1]$.

Недостаток линейной функции

Недостаток данной последовательности проб заключается в возможности образования кластеров (длинных последовательностей занятых ячеек, идущих подряд).

1. Это удлиняет поиск пустого сегмента, так как в этом случае необходимо просмотреть больше сегментов, чем при их **случайном** распределении.
2. Это увеличивает время добавления нового элемента и других операций.

ВЫВОД. Линейная последовательность проб довольно далека от **равномерного хеширования**.

Пример

Используя открытую адресацию с хеш-функцией

$$h(k, i) = ((k \bmod 5) + i) \bmod 5$$

сформировать хеш-таблицу T для последовательности ключей: 0, 10, 15, 13, 7.

Решение

Вычисление функции проб

Искомая таблица

$$h(0,0) = 0;$$

$$h(10,0) = 0; h(10,1) = 1;$$

$$h(15,0) = 0; h(15,1) = 1; h(15,2) = 2;$$

$$h(13,0) = 3;$$

$$h(7,0) = 2; h(7,1) = 3; h(7,2) = 4.$$

i	0	1	2	3	4
$T(i)$	0	10	15	13	7

2) Функция квадратичной последовательности проб

$$h(k, i) = \left(h'(k) + c_1 \cdot i + c_2 \cdot i^2 \right) \bmod m$$

Ячейки рассматриваются **не подряд**.

Номер ячейки **квадратично** зависит от номера попытки

Особенности квадратичной функции проб

Этот метод работает значительно лучше, чем линейный, но если нужно, чтобы при просмотре хеш-таблицы использовались все ячейки, значения параметров нельзя выбирать случайным образом

ВЫВОД. При данном способе выбора последовательности проб тенденции к образованию кластеров нет, но аналогичный эффект проявляется в форме образования вторичных кластеров

3) Двойное хеширование

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Последовательность проб в этой ситуации при работе с ключом k представляет собой **арифметическую прогрессию (по модулю m)** с первым членом $h_1(k)$ и шагом $h_2(k)$

Причины преимущества двойного хеширования

Перестановки индексов, возникающие при двойном хешировании, обладают многими свойствами, присущими **равномерному хешированию**

Особенности работы

Чтобы последовательность проб покрыла всю таблицу, значение $h_2(k)$ должно быть взаимно простым с m

1 вариант. Выбрать в качестве m степень двойки, а функцию $h_2(k)$ взять такую, чтобы она принимала только **нечетные** значения;

2 вариант. m – простое число, значения $h_1(k)$, $h_2(k)$ – целые положительные числа, меньшие m

Пример.

$$h_1(k) = k \bmod m,$$

$$h_2(k) = 1 + k \bmod m'$$

$$m' < m$$

Теорема 3 (неудачный поиск)

Математическое ожидание числа проб при
поиске в таблице с открытой адресацией
отсутствующего в ней элемента не
превосходит

$$\frac{1}{1 - \alpha}$$

Хеширование предполагается **равномерным**

Следствие (вставка)

В предположении равномерного хеширования при $\alpha < 1$ математическое ожидание числа проб при добавлении нового элемента в таблицу при закрытом хешировании не превосходит

$$\frac{1}{1 - \alpha}$$

Теорема 4 (удачный поиск)

Математическое ожидание числа проб при успешном поиске элемента в таблице при равномерном хешировании с $\alpha < 1$, если считать, что ключ для успешного поиска в таблице выбирается случайным образом и все такие выборы равновероятны, не превосходит

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

Другие методы хеширования

1. Ключи другой природы
2. Универсальное и идеальное хеширования

4) Задача о точках ($O(n)$)

Основная идея: алгоритм рассматривает точки в случайном порядке и поддерживает текущее значение δ для ближайшей пары в процессе обработки точек в этом порядке

Добравшись до новой точки p , алгоритм проверяет ее «окрестности»: не находятся ли какие-либо из ранее рассмотренных точек на расстоянии менее δ от p

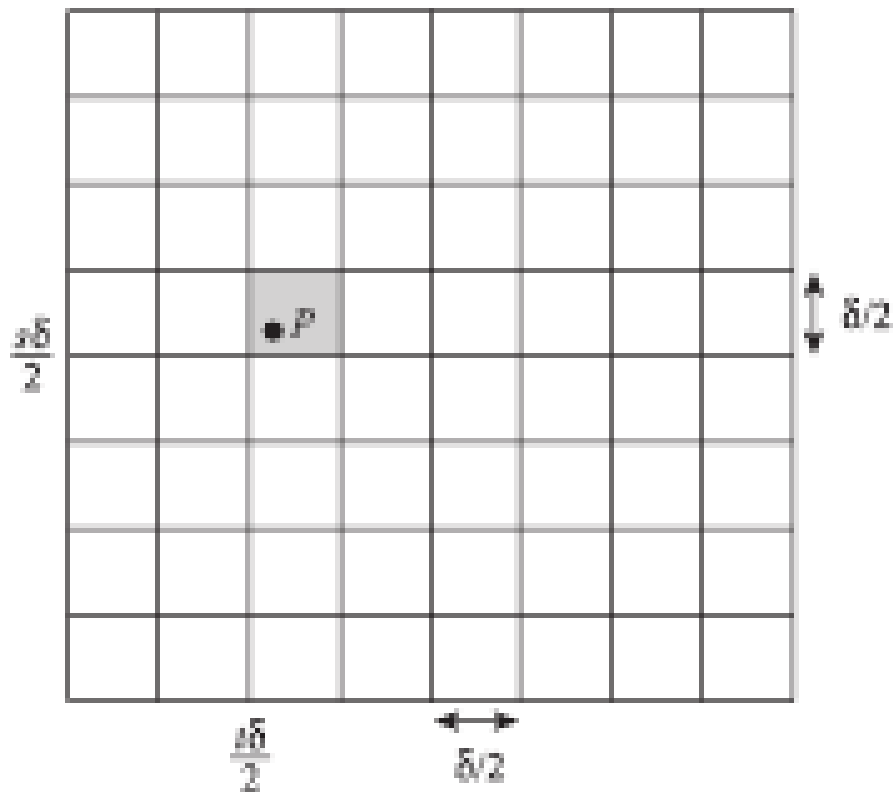
Если таких точек нет, значит, ближайшая пара не изменилась, и алгоритм переходит к следующей точке в случайном порядке

Иначе — ближайшая пара изменилась, и информацию о ней нужно обновить

Проверка расстояния

Основная часть алгоритма — метод, который проверяет, остается ли текущая пара точек с расстоянием δ ближайшей парой при добавлении новой точки, и если нет — находит новую ближайшую пару

Деление на подквадраты



Теоремы

1. Если две точки p и q принадлежат одному подквадрату S_{st} , то $d(p, q) < \delta$
2. Если для двух точек p, q из P выполняется $d(p, q) < \delta$, то подквадраты, содержащие эти точки, являются близкими

Понятие близких подквадратов,
доказательство!

Структура данных

Описание алгоритма зависит от возможности назвать подквадрат Sst и быстро определить, какие точки P содержатся в нем (если они есть)

Словарь — наиболее естественная структура данных для реализации таких операций

Описание алгоритма

Расположить точки в случайной последовательности p_1, p_2, \dots, p_n

Пусть δ – минимальное расстояние, обнаруженное на данный момент

Инициализировать $\delta = d(p_1, p_2)$

Вызвать *MakeDictionary* для сохранения подквадратов с длиной стороны $\delta/2$

Для $i = 1, 2, \dots, n$:

 Определить подквадрат S_{ij} , содержащий p_i

 Проверить 25 подквадратов, близких к p_i

 Вычислить расстояние от p_i до всех точек, находящихся в этих подквадратах

 Если существует точка p_j ($j < i$), для которой $\delta' = d(p_j, p_i) < \delta$

 Удалить текущий словарь

 Вызвать *MakeDictionary* для сохранения подквадратов со стороной $\delta'/2$

 Для каждой из точек p_1, p_2, \dots, p_i :

 Определить содержащий ее подквадрат с длиной стороны $\delta'/2$

 Вставить этот подквадрат в новый словарь

 Конец цикла

 Иначе

 Вставить p_i в текущий каталог

 Конец Если

Конец цикла

Анализ алгоритма

Алгоритм корректно поддерживает информацию ближайшей пары и выполняет

- не более $O(n)$ вычислений расстояния,
- $O(n)$ операций Lookip (проверка 25 подквадратов) и
- $O(n)$ операций MakeDictionary

2.5 АТД Дерево

Линейные структуры данных, предназначенные для хранения последовательностей элементов, задаются отношением линейного порядка. Обычно это *массив* и *связный список*

Задачи, в которых данные связаны более сложными отношениями, требуют разработки их структур, учитывающих особенности связей между рассматриваемыми элементами.

Одним из отношений, описывающим порядок, отличный от линейного, является общее отношение частичного порядка

Иерархия

Говорят, что набор данных образует **иерархию**, если на нем задано отношение частичного порядка, которое не допускает циклических зависимостей

Условно иерархию можно представить в виде уровней, на каждом из которых находятся элементы, попарно между собой не принадлежащие заданному отношению и следующие непосредственно за элементами предыдущего уровня, с которыми они образуют отношение.

Говоря об иерархии, обычно выделяют элемент, у которого нет предшественников

Дерево как модель иерархии

Наиболее удачной математической моделью для представления иерархий является дерево. Неформально дерево можно определить как совокупность элементов, называемых узлами (или вершинами), и отношений, образующих иерархическую структуру узлов

Формально же дерево можно определить несколькими способами

1) Основные понятия

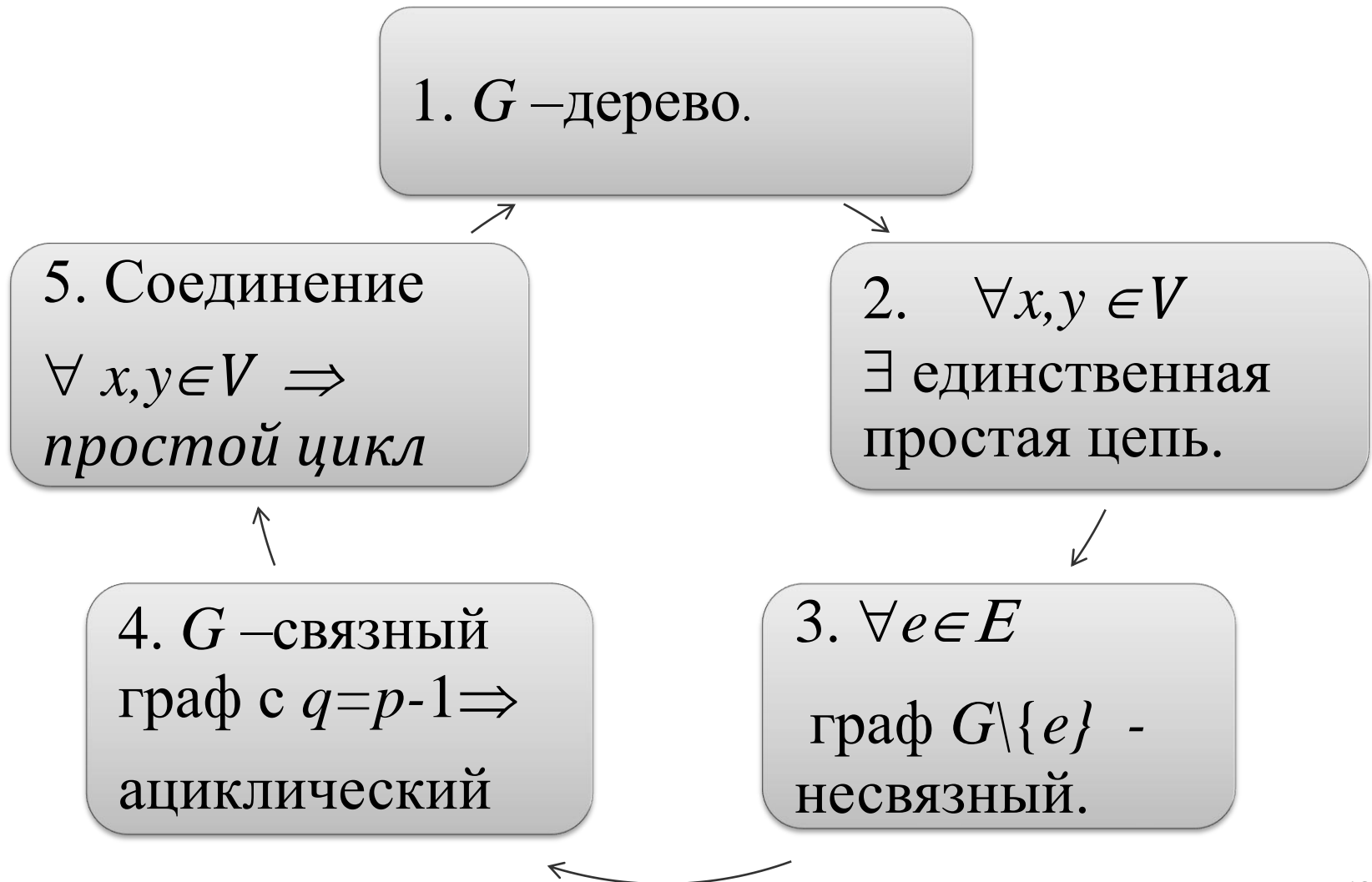
ДЕРЕВО – связный граф без циклов

ЛЕС – граф, все связные компоненты которого являются деревьями

Теорема о свойствах дерева (эквивалентные определения)

- Пусть $G(V, E)$ – неор. граф с p вершинами и q ребрами.
- Тогда следующие утверждения *эквивалентны*.
- 1. G есть дерево.
- 2. Любые две различные вершины графа G соединены единственной простой цепью.
- 3. G – связный граф, утрачивающий это свойство при удалении любого из его ребер.
- 4. G – связный ациклический граф с $p = q + 1$.
- 5. G – ациклический граф, в котором после соединения ребром двух любых вершин образуется единственный простой цикл.

Схема доказательства



Определение

Корневое дерево – это ориентированный граф, который удовлетворяет следующим условиям:

- 1) имеется в точности одна вершина, в которую не входит ни одна дуга (корень дерева);
- 2) в каждую вершину, кроме корня, входит ровно одна дуга;
- 3) из корня имеется путь к каждой вершине.

Вершина – узел

Основные понятия

Если в корневом дереве существует путь из вершины v в вершину w , то v называют **предком** вершины w , а w — **потомком** вершины v

Если вершина не имеет потомков, то ее называют **терминальной вершиной**, или **листом**

Нетерминальную вершину называют **внутренней**

Если (v, w) — дуга корневого дерева, то v называется **отцом** (непосредственным предком, родительской вершиной) вершины w , а w — **сыном** (непосредственным потомком, дочерней вершиной) вершины v

Вершина v корневого дерева T и ее потомки вместе образуют **поддерево** корневого дерева T с корнем в вершине v

Глубиной вершины v в корневом дереве называется длина пути из корня дерева в эту вершину v

Высотой вершины v в корневом дереве называется длина самого длинного пути из вершины v до одного из его потомков (листа)

Высотой корневого дерева называется высота корня дерева

Уровнем вершины v называется разность высоты корневого дерева и глубины вершины v

Определение

Упорядоченное корневое дерево — это корневое дерево, у которого дуги, выходящие из каждой вершины, упорядочены (например, слева направо)

Определение (рекурсивное)

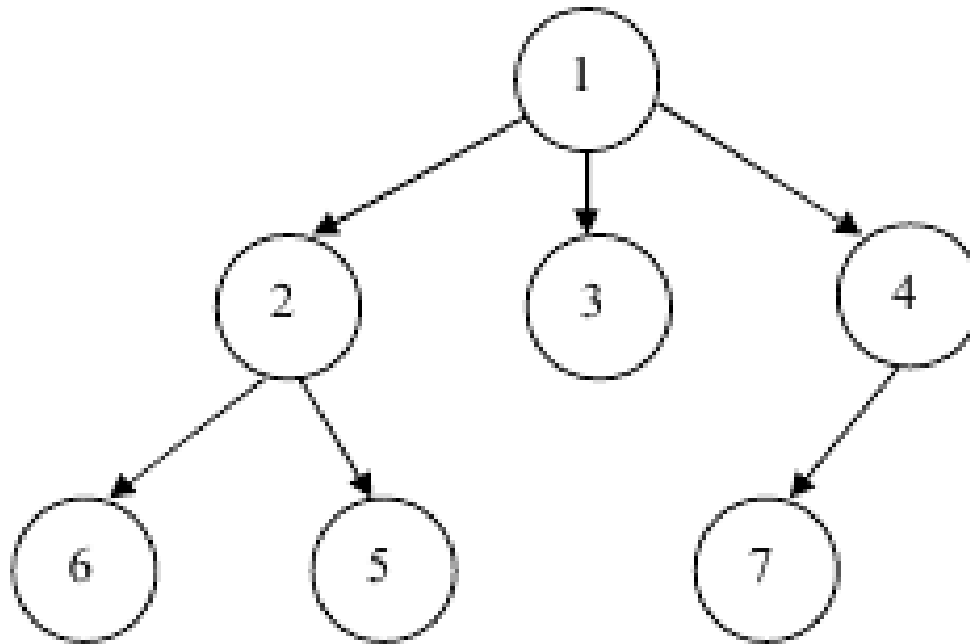
Дерево — это математический объект, определенный рекурсивно следующим образом:

1) один узел, называемый корнем дерева, является деревом;

2) пусть n — это узел, а T_1, \dots, T_k — непересекающиеся между собой деревья с корнями n_1, \dots, n_k соответственно. Тогда, сделав узел n родителем узлов n_i , получим новое дерево T с корнем n и поддеревьями T_1, \dots, T_k . В этом случае, узлы n_1, \dots, n_k называются *дочерними узлами* (или *детьми*) узла n .

Задание. Определите все введенные ранее понятия, взяв за основу определение 2

Каноническое представление (массив предков) корневых деревьев



i – вершина	1	2	3	4	5	6	7
$p[i]$ – отец вершины i	0	1	1	1	2	2	4

2) Бинарное дерево

Бинарное дерево — это упорядоченное корневое дерево, у каждой вершины которого имеется не более двух сыновей

В бинарном дереве каждый сын произвольной вершины определяется либо как **левый** сын, либо как **правый** сын

Поддерево (если оно существует), корнем которого является левый сын вершины v , называется **левым поддеревом** вершины v

Аналогично определяется **правое поддерево** для вершины v

Способы представления бинарных деревьев

- 1) представление в виде списковой структуры;
- 2) представление в виде двух массивов – Left и Right.
- Если вершина j – левый (правый) сын вершины i , то $\text{Left}[i] = j$; ($\text{Right}[i] = j$);
- если у вершины i нет левого (правого) сына, то $\text{Left}[i] = 0$; ($\text{Right}[i]=0$).

Преобразование произвольного дерева в бинарное дерево

Работа с произвольными деревьями затруднительна по причине того, что требуется хранить список детей для каждого из узлов дерева

Но существует широкий класс задач на бинарных деревьях, для которых разработаны эффективные алгоритмы их решения. Эти алгоритмы не всегда могут быть приспособлены для решения обобщенных версий задач на произвольных деревьях

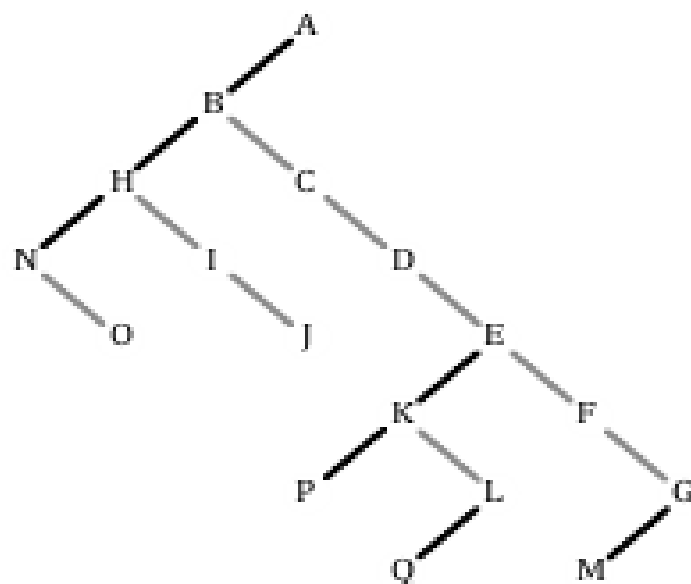
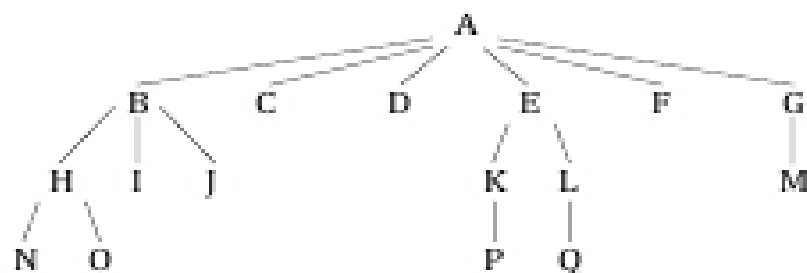
В связи с этим возникает потребность в разработке алгоритмов, которые позволят свести задачу на произвольных деревьях к задачам на бинарных деревьях

Одним из них является алгоритм преобразования произвольного дерева в соответствующее ему бинарное.

Между произвольными упорядоченными деревьями и бинарными деревьями существует **взаимно однозначное соответствие**. Для того чтобы построить такое соответствие, достаточно произвольное дерево рассмотреть с точки зрения «левый сын – правый брат»

Пусть T – произвольное дерево с узлами $\{n_1, \dots, n_k\}$, а B – некоторое бинарное дерево с узлами $\{n'_1, \dots, n'_k\}$. Тогда дерево B является взаимно-однозначным *соответствием* дерева T , если:

- 1) узел n_i дерева T соответствует узлу n'_i дерева B , причем метки узлов n_i и n'_i являются одинаковыми;
- 2) узел n_i есть корень дерева T , то узел n'_i – корень дерева B ;
- 3) узел n_j есть самый левый сын узла n_i , то n'_j – левый сын узла n'_i ; если у узла n_i нет детей, то у узла n'_i нет левого сына;
- 4) узел n_j есть правый брат узла n_i , то n'_j – правый сын узла n'_i .



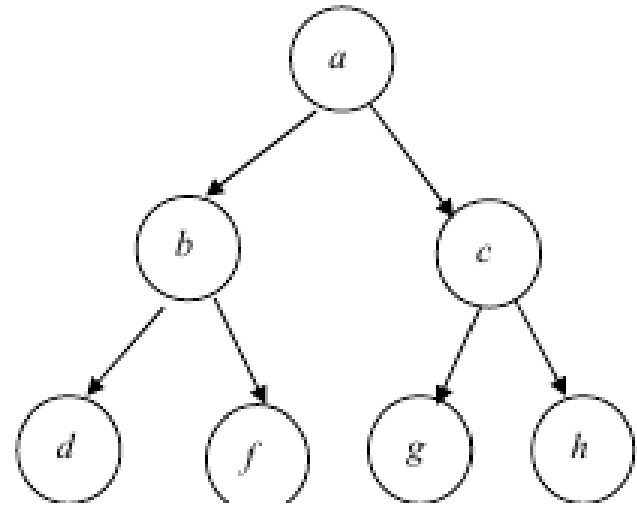
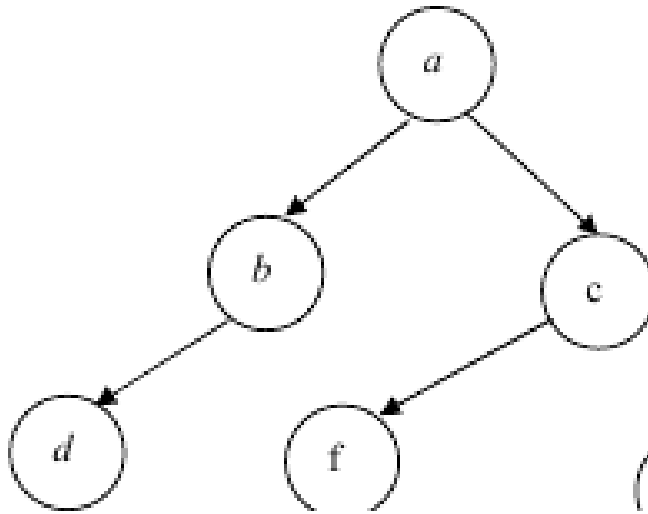
Задание. Изобразите аналогичное дерево из 24 узлов и преобразуйте его в бинарное

3) Полное бинарное дерево

Полным бинарным деревом будем называть такое упорядоченное корневое дерево, в котором:

- 1) каждая вершина имеет не более двух сыновей;
- 2) заполнение вершин дерева осуществляется от корня к листьям по уровням;
- 3) заполнение вершинами каждого уровня проводится слева направо

Неполное и полное БД



Очевидно, что минимальное количество вершин в полном трехуровневом бинарном дереве равно 4, а максимальное число в полностью заполненном – 7.

Минимальное и максимальное число вершин ПБД высоты h

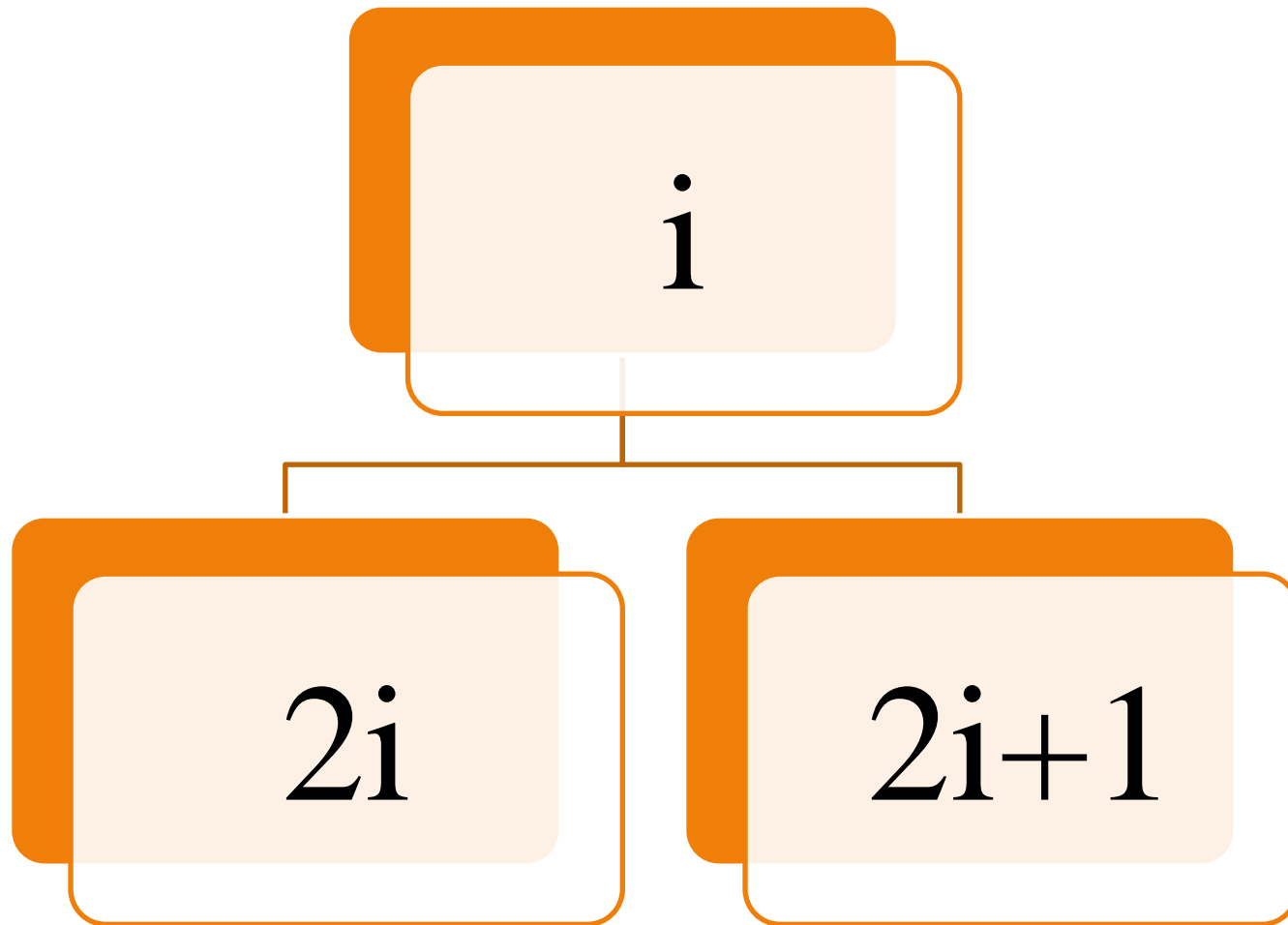
$$1 + 2^1 + 2^2 + \dots + 2^{h-1} + 1 = \sum_{i=0}^{h-1} 2^i + 1 = \frac{2^h - 1}{2 - 1} + 1 = 2^h$$

$$1 + 2^1 + 2^2 + \dots + 2^h = \sum_{i=0}^h 2^i = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

$$2^h \leq n \leq 2^{h+1} - 1,$$

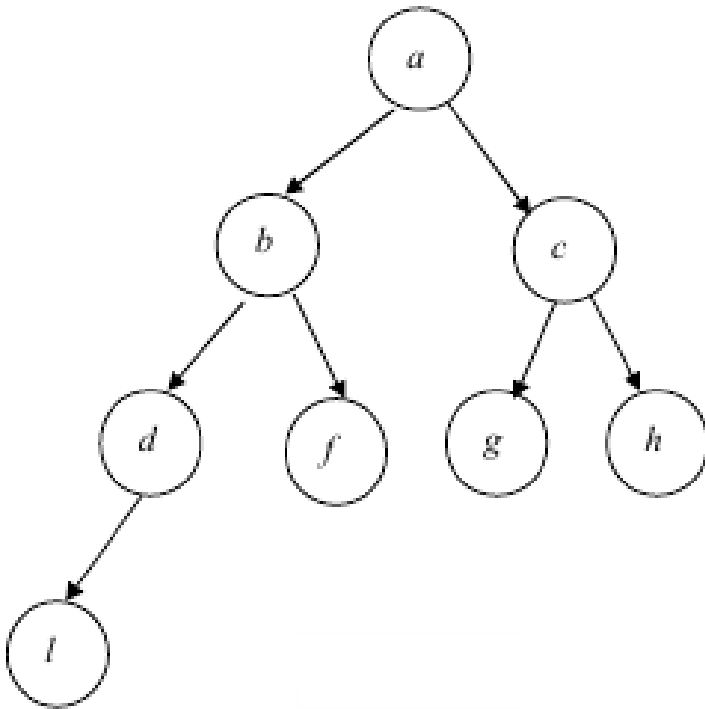
$O(\log n)$ – верхняя оценка количества уровней (высоты ПБД)

Представление полного бинарного дерева с K вершинами



Массив и переменная, равная количеству вершин в дереве

i	1	2	3	4	5	6	7	8
$h[i]$	a	b	c	d	f	g	h	l



2.6 АТД Множество

Под математической моделью множества будем понимать некоторую совокупность элементов, каждый из которых является либо множеством, либо примитивным элементом, (атомом)

Основные операции над множествами:

- создание множества,
- вставка / удаление элементов,
- поиск,
- объединение,
- пересечения и др.

АТД является «Множеством», если на нем поддерживаются операции, определенные на математической модели множества

1) Линейные способы представления множеств

Выбор того или иного способа представления информации зависит:

- от специфики задачи,
- задано ли на множестве элементов отношение порядка

Способы представления: массивы, линейные структуры данных, линейные АТД и битовые маски

2) АТД Системы непересекающихся множеств

Нелинейный АТД, предназначенный для хранения системы непересекающихся множеств и поддерживающий следующий набор операций:

- создание одноэлементного множества;
- объединение множеств;
- поиск множества, которому принадлежит заданный элемент

Основные структуры данных для хранения операций

- Массив (для каждого объекта сохранить в массив номер представителя множества, которому этот объект принадлежит),
- линейные списки (для хранения элементов каждого из подмножеств),
- деревья (хранить элементы каждого множества в виде дерева) ЛЕС

Упражнение. Разберите вопросы сложности операций при разных представлениях самостоятельно!

2.7 АДД Приоритетная очередь

Реализует операции:

- добавление элемента;
- удаление элемента с минимальным ключом

Пояснения к названию

Название «Очередь с приоритетами» обусловлено видом упорядочивания, которому подвергаются данные, хранящиеся посредством АТД

Термин «очередь» предполагает, что элементы ожидают некоторого обслуживания, а термин «приоритет» определяет последовательность обработки элементов

В отличие от обычных очередей, приоритетная очередь не придерживается принципа FIFO

Реализации приоритетных очередей

На основе массивов или списковых структур при
небольших объемах данных

На основе сбалансированных деревьев
(бинарные, биномиальные, фибоначчиевы
пирамиды (кучи)) при больших объемах
данных

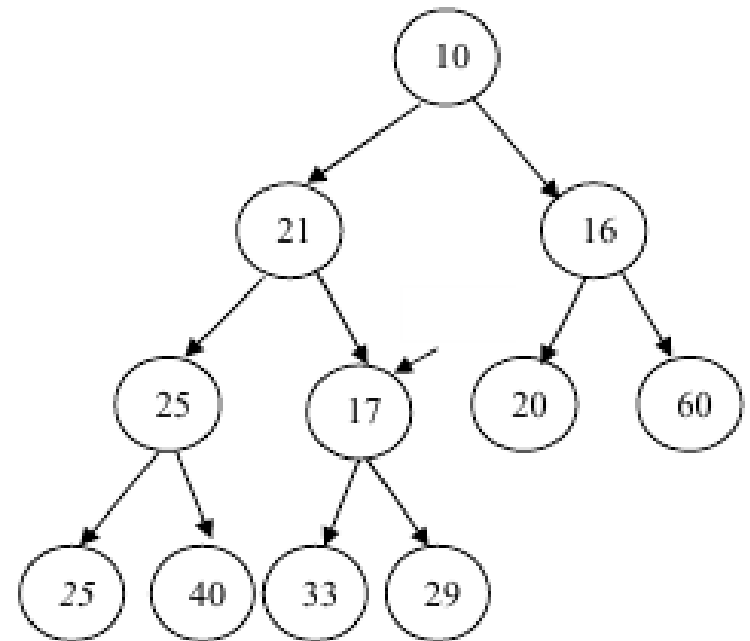
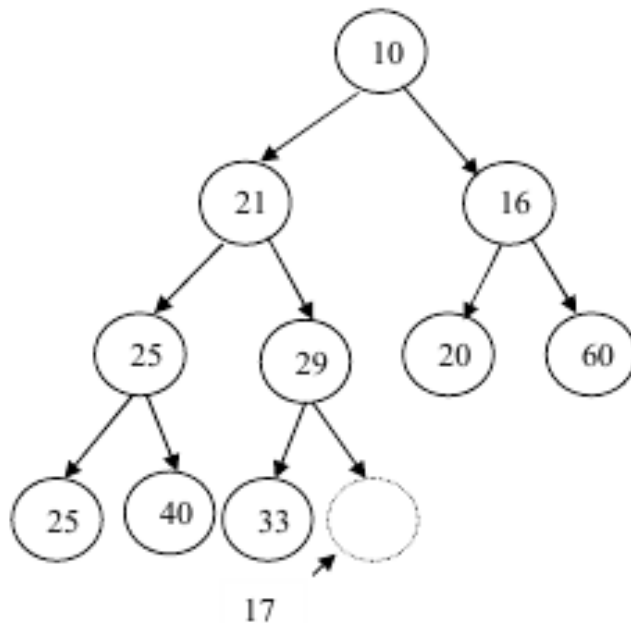
1) Бинарная пирамида (куча)

Полное бинарное дерево, для которого выполняется основное свойство (инвариант) структуры данных куча, — элементы организованы таким образом, что приоритет любой вершины не ниже приоритета каждого из ее сыновей,— называется бинарной кучей

Основные операции с бинарными кучами

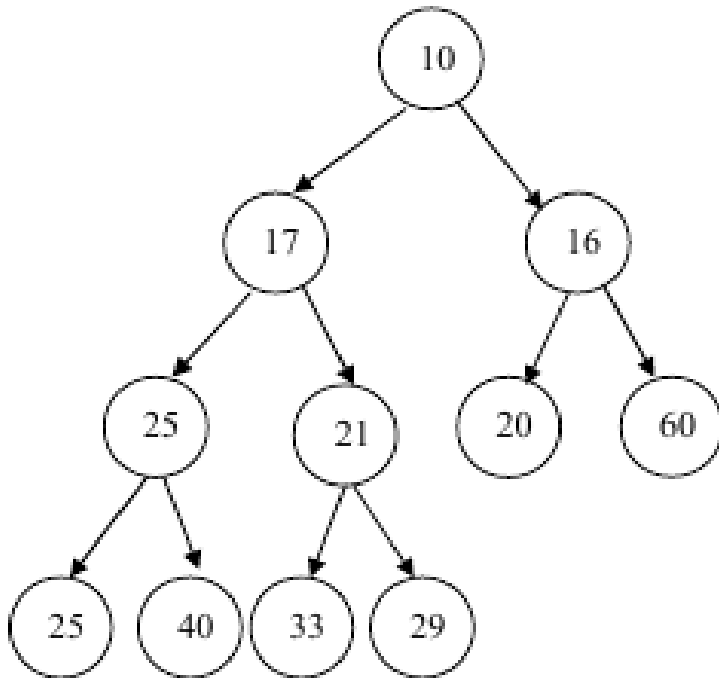
1) Добавление элемента к бинарной куче

1-й обмен



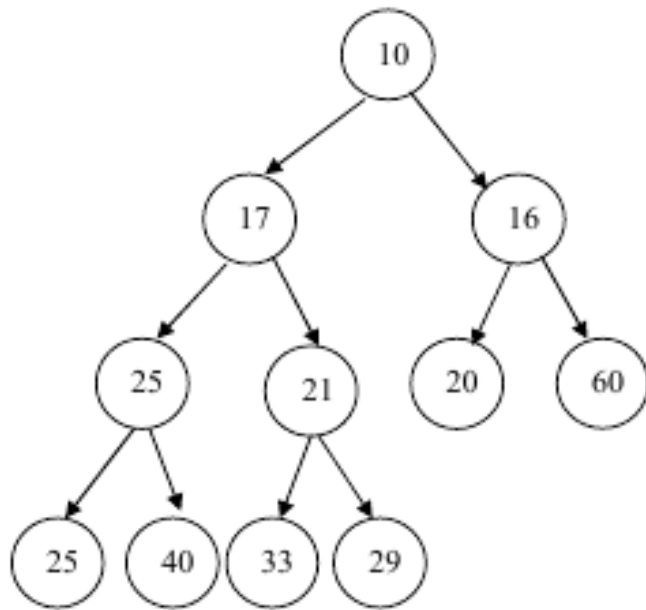
Трудоёмкость операции добавления элемента

2-й обмен



- Трудоёмкость операции добавления определяется количеством выполнения цикла `while` и не превышает количества уровней кучи, которое равно $O(\log n)$.

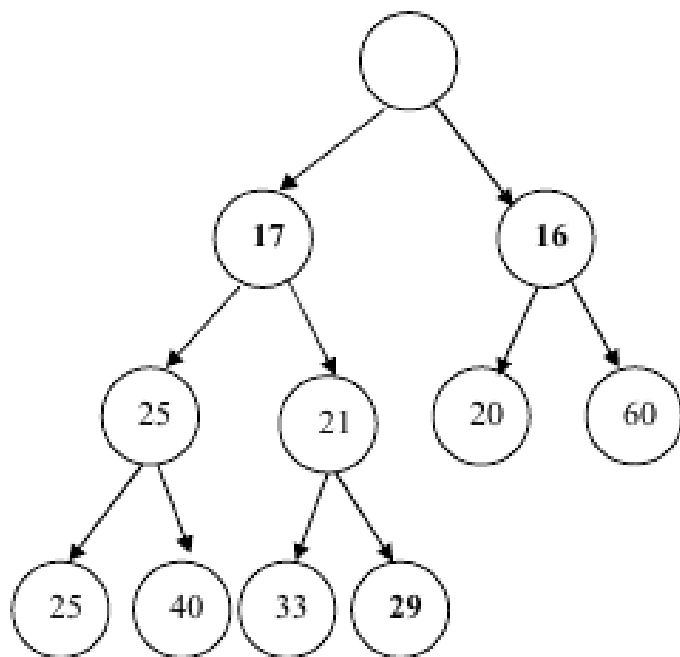
2) Удаление минимального элемента из бинарной кучи



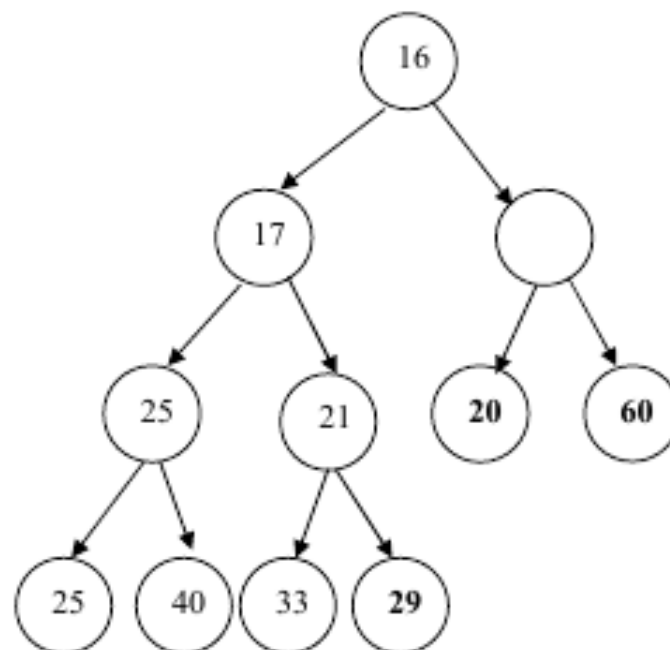
Краткое описание алгоритма

- Элемент с максимальным приоритетом находится в корне полного бинарного дерева, т. е. имеет индекс 1.
- Поэтому после удаления первого элемента его место займет тот из его сыновей или последний элемент, который имеет больший приоритет.
- Освободившееся место сына займет его сын или последний элемент и т. д.
- После таких преобразований внутри кучи не должно остаться «свободных» мест.

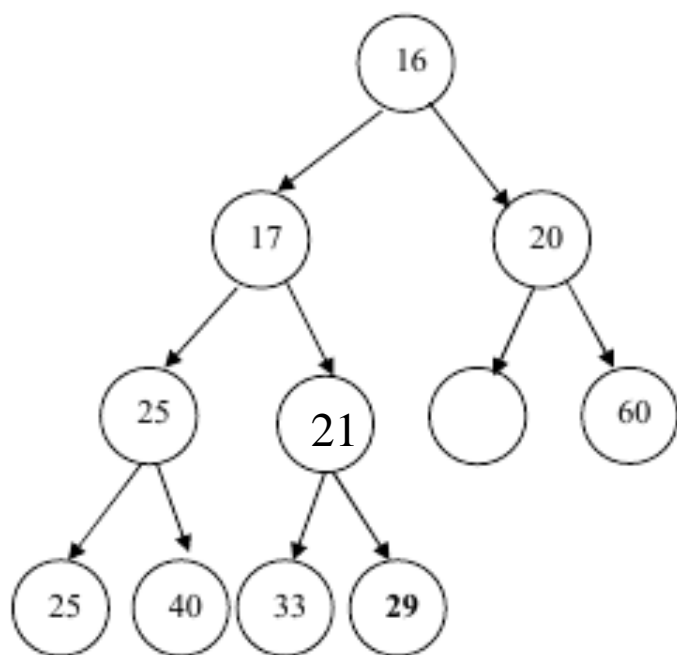
1-й шаг



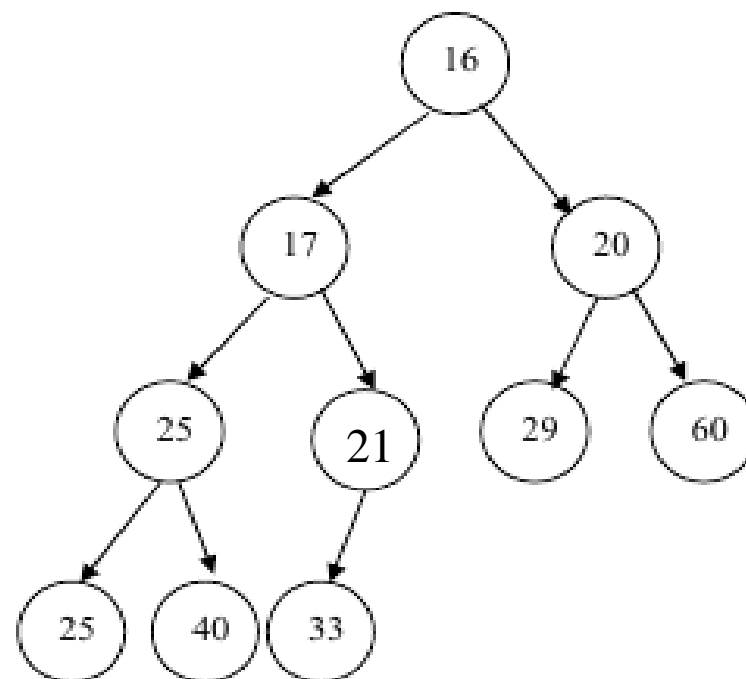
2-й шаг



3-й шаг



Последний шаг



Замечание

Можно использовать и другой алгоритм удаления минимального элемента, имеющий трудоемкость $O(\log n)$

Переносим в корень дерева последний элемент кучи, затем удаляем из дерева вершину, в которой он ранее находился, и полагаем корень дерева в качестве текущего элемента

После этих действий может нарушиться инвариант кучи, и для его восстановления текущий элемент перемещается вниз до тех пор, пока для него не будет обеспечен инвариант (одно перемещение заключается в том, что текущий элемент меняется местами с самым маленьким из своих сыновей)

3)Создание бинарной кучи

Одним из способов создания бинарной кучи для последовательности A из n элементов может быть выполнение n раз процедуры добавления элемента.

Поскольку сложность процедуры добавления нового элемента в бинарную кучу есть $O(\log n)$, то сложность построения бинарной кучи стоило бы ожидать порядка $O(n \log n)$

Покажем, что существует алгоритм построения бинарной кучи с трудоемкостью $O(n)$

2) Описание алгоритма

Первоначально для последовательности A из n элементов строим полное бинарное дерево

Для поддержки инварианта кучи выполним следующую последовательность шагов, которая преобразует построенное полное бинарное дерево в бинарную кучу. При этом обеспечение инварианта 1 будет происходить в порядке возрастания высот вершин (понятно, что можно начинать с тех узлов, которые заведомо имеют сыновей)

Процедура, обеспечивающая инвариант кучи для вершины с индексом i

Перемещение текущего элемента вниз до тех пор, пока для него не будет обеспечен этот инвариант

Количество перемещений вниз некоторого элемента не превосходит высоты, на которой он расположен

Вывод. Трудоемкость приведенного алгоритма создания бинарной кучи не превосходит суммы высот всех вершин полного бинарного дерева

3) Теорема

Сумма высот всех вершин полного бинарного
дерева высотой h не превосходит
величины

$$S = 2^{h+1} - h - 2$$

Доказательство

- 1. Назовите число вершин максимально заполненного ПБД высотой h .
- 2. Назовите число вершин в таком дереве, которые имеют высоту i .
- 3. Просуммируйте по i все такие вершины.

Выкладки

$$S = \sum_{i=0}^h 2^{h-i} \cdot i = h + 2(h-1) + 4(h-2) + 8(h-3) + \dots + 2^{h-1}$$

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \dots + 2^h$$



$$\begin{aligned} 2S - S &= -h + (2h - 2(h-1)) + (4(h-1) - 4(h-2)) + \dots + \\ &+ 2^{h-1} + 2^h = -h + \frac{2(2^h - 1)}{2 - 1} = 2^{h+1} - h - 2 \end{aligned}$$

Следствие

$$n \geq 2^h,$$



Трудоемкость описанного алгоритма
построения бинарной кучи есть $O(n)$

Задание. Постройте бинарную кучу по описанному алгоритму
для последовательности из 13 первых букв своих ФИО