Курс
"Алгоритмы и структуры данных"
1 семестр ФПМИ МФТИ

Даниил Гагаринов
github.com/yaishenka

Январь 2019

Содержание

Введение

Определение алгоритма. Примеры простых алгоритмов: вычисление числа Фибоначчи, проверка числа на простоту, быстрое возведение в степень.

Def. Алгоритм - это формально описанная вычислительная процедура, получающая исходные данные (input), называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход (output).

Note. Алгоритм определяет функцию (отображение) $F : X \to Y$. Где X - множество исходных данных, Y - множество значений.

Примеры.

1)Числа Фибоначчи (рекурсивный алгоритм)

```
int GetNthFibonacciNumber(int n) {
  if (n == 0 || n == 1) {
      return 1;
  }
  return GetNthFibNumber(n-1) + GetNthFibNumber(n-2);
}
```

2)Числа Фибоначчи (нерекурсивный алгоритм)

```
int GetNthFibonacciNumber(int n) {
  if (n == 0) {
    return 1;
  }
  int previous_number = 1;
  int current_number = 1;
  for (int i = 2; i <= n; ++i) {
    int temp = current_number;
    current_number += previous_number;
    previous_number = temp;
  }
    return current_number;
}
```

3)Проверка числа на простоту

```
bool IsPrime (int number) {
  if (number == 1) {
    return false;
  }
  for (int i = 2; i * i <= n; ++i) {
    if (number % i == 0) {
      return false;
    }
  }
  return true;
}
```

4)Быстрое возведение в степень

Представим степень в двоичном виде. Разложим в сумму степеней двойки.

```
double Power(double number, int power) {
  double result = 1;
  double number_in_power_of_2 = number;
  while (power > 0) {
    if ((power & 1) == 1) { //if last bit is 1
      result *= number_in_power_of_2;
    }
    number_in_power_of_2 *= number_in_power_of_2;
    power >>= 1;
  }
  return result;
}
```

Асимптотические обозначения, работа с ними.

Def. Для функции $g(n)$ записи $\Theta(g(n)), O(g(n))$ и $\Omega(g(n))$ означают следующие множества функций:

$\Theta(g(n)) = \{f(n):$ существуют положительные константы c_1, c_2 и $n_0,$ такие что
$0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\ \forall n \geqslant n_0\}$

$O(g(n) = \{f(n):$ существуют положительные константы c и n_0 такие что
$0 \leqslant f(n) \leqslant cg(n)\ \forall n \geqslant n_0$

$\Omega(g(n) = \{f(n):$ существуют положительные константы c и n_0 такие что
$0 \leqslant cg(n) \leqslant f(n)\ \forall n \geqslant n_0$

Def. T(n) - время работы алгоритма

Def. M(n) - память, задействованная алгоритмом

Примеры.

1)Оценка времени работы рекурсивного алгоритма чисел Фибоначчи

Формула Бине: $F(n) = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}$ где φ - золотое сечение
T(n) = T(n-1) + T(n-2) т.о. T(n) ведет себя как F(n)
$F(n) = [\varphi^n / \sqrt{5}] \Rightarrow T(n) = \Omega(\varphi^n)$
Объем доп. памяти M(n) = O(n) - максимальная глубина рекурсии.

2)Оценка времени работы нерекурсивного алгоритма чисел фибоначчи

T(n) = O(n) - количество итераций в цикле
Объем доп. памяти M(n) = O(1)

Определение структуры данных, абстрактного типа данных (интерфейса).

Def. Абстрактный тип данных (АТД) — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций.

Note. АТД = интерфейс

Def. Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.

Массив. Линейный поиск. Бинарный поиск.

Def. Массив – набор однотипных компонентов (элементов), расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу (индексам). Традиционно индексирование элементов массивов начинают с 0.

Def. Размерность массива – количество индексов, необходимое для однозначного доступа к элементу массива.

Задача. Проверить, есть ли заданный элемент в массиве.

Решение:

Последовательно проверяем все элементы массива, пока не найдем заданный элемент, либо пока не закончится массив. Время работы в худшем случае T(n) = O(n) , где n – количество элементов в массиве.

```
template <typename T>
bool HasValue(T* array, size_t array_size, T&& value) {
  for (int i = 0; i < array_size; ++i) {
    if (array[i] == value) {
      return true;
    }
  }
  return false;
}
```

Def. Упорядоченный по возрастанию массив – массив A, элементы которого сравнимы, и для любых индексов k и l : $k < l \hookrightarrow A[k] \leqslant A[l]$

Задача. Проверить, есть ли заданный элемент в упорядоченном массиве. Если он есть, вернуть позицию его первого вхождения. Если его нет, вернуть -1.

Решение:

Шаг. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения. Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

```
template <typename T>
int GetIndexOfValue(T* array, size_t size, T&& value) {
  int left_border = 0;
  int right_border = size;
  while (left_border < right_border) {
    int middle = (left_border + right_border) / 2;
    if (value <= array[middle]) {
      right_border = middle;
    } else {
      left_border = middle + 1;
    }
  }
  return (left_border == size || array[left_border] != element) ? -1 : left_border;
}
```

Тема 1. Базовые структуры данных

Динамический массив.

Def. АТД «Динамический массив» — интерфейс с операциями

– Add. Добавление элемента в конец массива (или PushBack),

– GetAt. Доступ к элементу массива по индексу (или оператор []).

Задача. Реализация динамического массива

Решение:

Реализация будет

- содержать массив фиксированной длины - буфер;

- хранить количество элементов.

Если буфер закончится, то:

- выделим новый буфер, в два раза больший исходного

- скопируем туда старый буфер

- добавим новый элемент

```
template <typename ValueType>
class DynamicArray {
 public:
  DynamicArray();
  ~DynamicArray();

  ValueType& GetAt(int index) const;
  ValueType operator[] (int index) const;
  ValueType& operator[] (int index);

  size_t GetSize() const;

  void AddElement(ValueType&& value);
 private:
  void Grow();

  ValueType* buffer_;
  size_t capacity_;
  size_t size_;
};

template <typename ValueType>
DynamicArray<ValueType>::DynamicArray() : buffer_{new ValueType[2]},
                                          capacity_{2}, size_{0} {}
template <typename ValueType>
DynamicArray<ValueType>::~DynamicArray() {
  delete [] buffer_;
}

template <typename ValueType>
ValueType& DynamicArray<ValueType>::GetAt(int index) const {
  assert (index >= 0 && index < size_ && buffer_ != nullptr);

  return buffer_[index];
}

template <typename ValueType>
ValueType DynamicArray<ValueType>::operator[](int index) const {
  return GetAt(index);
}

template <typename ValueType>
ValueType& DynamicArray<ValueType>::operator[](int index) {
  return GetAt(index);
}

template <typename ValueType>
```

```
void DynamicArray<ValueType>::AddElement(ValueType&& value) {
  if (size_ >= capacity_) {
    Grow();
  }
  buffer_[size_] = value;
  ++size_;
}

template <typename ValueType>
void DynamicArray<ValueType>::Grow() {
  capacity_ *= 2;
  auto new_buffer = new ValueType[capacity_];
  for (int i = 0; i < size_; ++i) {
    new_buffer[i] = buffer_[i];
  }
  delete [] buffer_;
  buffer_ = new_buffer;
}
```

Note. Время работы Add?

- В лучшем случае = O(1)
- В худшем случае = O(n)
- В среднем случае - ?

Амортизационный анализ. Амортизированное (учетное) время добавления элемента в динамический массив.

Def. При амортизационном анализе время, требуемое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям.

Note. Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность. При амортизационном анализе гарантируется средняя производительность операций в наихудшем случае.

Def. Пусть $S(N)$ - время выполнения последовательности всех n операций в наихудшем случае. Амортизированной стоимостью (временем) $AC(n)$ назвается среднее время, приходящееся на одну операцию. $AC(n) = \frac{S(N)}{n}$

Утверждение. Пусть в реализации функции grow() буфер удваивается. Тогда амортизированная стоимость функции Add составляет O(1).

Доказательство:

Рассмотрим последовательность из n операций Add. Обозначим $P(k)$ - время выполнения Add в случае, когда size = k, где size - количество элементов в массиве.

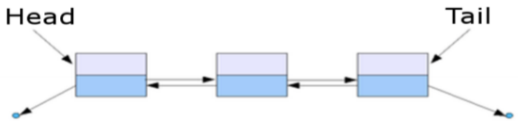
- $P(k) \leqslant c_1 k$, если $k = 2^m$;
- $P(k) \leqslant c_2$, если $k \neq 2^m$.

$S(N) = \sum\limits_{k=0}^{n-1} P(k) \leqslant c_1 \sum\limits_{m:2^m<n} 2^m + c_2 \sum\limits_{k:k\neq 2^m} 1 \leqslant 2c_1 n + c_2 n = (2c_1 + c_2)n.$
Амортизированное время $AC(n) = \frac{S(n)}{n} \leqslant 2c_1 + c_2 = O(1)$ ■

Двусвязный и односвязный список. Операции. Объединение списков.

Def. Односвязный список (однонаправленный связный список) Ссылка в каждом узле одна. Она указывает на следующий узел в списке.

Def. Двусвязный список (двунаправленный связный список) Ссылки в каждом узле указывают на предыдущий и на последующий узел.



Операции со списками:

- Поиск элемента, время работы в худшем случае = O(n)
- Вставка элемента, время работы в худшем случае = O(1)
- Удаление элемента, время работы в худшем случае = O(n)
- Объединение списков, время работы в худшем случае = O(n), где n - длина первого списка

Note. Тут должен быть код списка, но его украли :(

Сравнение списка с массивом

Недостатки списков:

- Нет доступа по индексу
- Расходуется доп. память

- Узлы могут располагаться в памяти разряженно

Преимущества списков:

- Быстрая вставка узла
- Быстрое удаление(если знаем указатель на элемент)

АТД Стек

Def. Стек - список элементов, организованный по принципу LIFO = Last In First Out

Операции:

- Push - вставка
- Pop - извлечение элемента, добавленного последним.

Задача. Реализация СД "Стек"

Решение: Реализация на массиве

Note. Стек так же можно реализовать на списке

```
template <typename T>
class Stack {
 public:
  const T None = T(-1);

  Stack() = default;
  ~Stack();

  size_t GetSize() const;

  T GetTop() const;
  T Pop();

  void Push(T value);

 private:
  void Resize();

  size_t size_{0};
  size_t capacity_{2};
  size_t tail_{0};

  T* buffer_{new T[capacity_]};
```

```
};

template <typename T>
Stack<T>::~Stack() {
  delete[] buffer_;
}

template <typename T>
size_t Stack<T>::GetSize() const {
  return size_;
}

template <typename T>
T Stack<T>::GetTop() const {
  if (size_ == 0) {
    return None;
  }

  return buffer_[tail_ - 1];
}

template <typename T>
T Stack<T>::Pop() {
  if (size_ == 0) {
    return None;
  }

  --size_;
  --tail_;

  return (buffer_[tail_]);
}

template <typename T>
void Stack<T>::Push(T value) {
  if (capacity_ == size_) {
    Resize();
  }

  buffer_[tail_] = value;

  ++tail_;
  ++size_;
}

template <typename T>
void Stack<T>::Resize() {
```

```
    T* new_buffer = new T[capacity_ * 2];

    for (size_t i = 0; i < tail_; ++i) {
      new_buffer[i] = buffer_[i];
    }
    capacity_ *= 2;

    delete[] buffer_;

    buffer_ = new_buffer;
  }
```

АТД Очередь

Def. Очередь - список элементов, организованных по принципу FIFO = First In First Out

Операции:

- Enqueue - вставка
- Dequeue - извлечение элемента, добавленного первым.

Задача. Реализация СД "Очередь"

Решение: Реализация на массиве

Note. Очередь так же можно реализовать на списке

Note. Тут должен был быть код очереди, но его украли :(Оставили только идею "Храним указатель на массив, текущее начало и конец очереди. Считаем массив зацикленным."

АТД Дек

Def. Двусвязная очередь (дек) – список элементов, в котором элементы можно добавлять и удалять как в начало, так и в конец.

Операции:

- PushBack - вставка в конец
- PushFront - вставка в начало
- PopBack - извлечение из конца
- PopFront - извлечение из начала

Задача. Реализация дека

```
template <typename T>
class Deque {
 public:
  const T None = T(-1);  // only if T have conversion to -1 or construct from -1

  Deque() = default;
  ~Deque();

  size_t GetSize() const;

  T PopFront();
  T PopBack();

  void PushFront(T value);
  void PushBack(T value);

 private:
  void Resize();

  size_t size_{0};
  size_t capacity_{2};
  size_t head_{0};
  size_t tail_{0};

  T* buffer_{new T[capacity_]};
};

template <typename T>
Deque<T>::~Deque() {
  delete[] buffer_;
}

template <typename T>
size_t Deque<T>::GetSize() const {
  return size_;
}

template <typename T>
T Deque<T>::PopFront() {
  T result = None;

  if (size_ != 0) {
    result = buffer_[head_];
    head_ = (head_ + 1) % capacity_;
    --size_;
```

```
  }

  return result;
}

template <typename T>
T Deque<T>::PopBack() {
  T result = None;
  if (size_ != 0) {
    tail_ = (tail_ - 1) % capacity_;
    result = buffer_[tail_];
    --size_;
  }

  return result;
}

template <typename T>
void Deque<T>::PushFront(T value) {
  if (size_ == capacity_) {
    Resize();
  }

  head_ = (head_ - 1) % capacity_;

  buffer_[head_] = value;
  ++size_;
}

template <typename T>
void Deque<T>::PushBack(T value) {
  if (size_ == capacity_) {
    Resize();
  }

  buffer_[tail_] = value;

  tail_ = (tail_ + 1) % capacity_;

  ++size_;
}

template <typename T>
void Deque<T>::Resize() {
  T* new_buffer = new T[capacity_ * 2];

  if (head_ < tail_) {
```

```
            for (size_t i = head_; i < tail_; ++i) {
              new_buffer[i] = buffer_[i];
            }
          } else {
            for (size_t i = head_; i < capacity_; ++i) {
              new_buffer[i] = buffer_[i];
            }

            for (size_t i(0); i < tail_; ++i) {
              new_buffer[i + capacity_] = buffer_[i];
            }

            tail_ += capacity_;
          }
          capacity_ *= 2;

          delete[] buffer_;

          buffer_ = new_buffer;
        }
```

Задача. Поддержка минимума в стеке

Решение: Будем хранить в каждом элементе индекс минимума среди элементов, лежащих ниже него в стеке

Очередь из двух стеков

Входной и выходной стек. InStack и OutStack. Перекладываем из InStack в OutStack, если требуется извлечь элемент, а в выходном стеке закончились элементы.

- Вставка - O(1)
- Извлечение - O(1) или O(n), если выходной стек пуст и требуется перекладывание.

Утверждение. Учетная стоимость извлечения = O(1)

Доказательство:

$AC(n) = \frac{S(n)}{n} = \frac{T_{\text{Enq}}(n)+T_{\text{Deq}}(n)}{n} \leqslant \frac{|Enq|+|Deq|+|Enq|}{n} = O(1)$, где |Enq| - число вставок, |Deq| - число извлечений, n - общее число операций.

То есть вставка просто суммируется, а удаление в худшем случае дает нам перемещение всех элементов в другой стек, после которого n операций идут за O(1) ■

Задача. Поддержка минимума в очереди.

Решение: Построим очередь на двух стеках, каждый из которых поддерживает в себе свой минимум. При запросе на минимум вернем min(min1, min2)

Двоичная куча. АТД Очередь с приоритетом.

СД Двоичная куча

Def. Двоичная куча, пирамида, или сортирующее дерево — такое почти полное двоичное дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения ее потомков
- Глубина листьев (расстояние до корня) отличается не более чем на один
- Последний слой заполняется слева направо

Def. Глубина кучи - максимальное расстояние от корня до листа

Утверждение. Глубина кучи = O(log(n)), где n - количество элементов

Доказательство:

Пусть есть куча из n элементов. Пронумеруем ее слои (начиная с 0). Понятно, что на каждом слое кроме последнего 2^k элементов, где k - номер слоя. Просуммируем количество элементов на каждом слое кроме последнего $\Rightarrow$ количество элементов в дереве кроме последнего слоя $= 2^h - 1$. На последнем слое минимум 1 элемент $\Rightarrow n \geqslant 2^h - 1 + 1 \Rightarrow n \geqslant 2^h$ прологарифмируем неравенство $\Rightarrow h \leqslant log(n) \Rightarrow h = O(log(n))$ ■

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности. Для восстановления этого свойства служат две процедуры Sift Up и Sift Down.

Def. Sift Down - спускает элемент, который меньше дочерних. Если i-й элемент больше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i-й элемент с наибольшим из его сыновей, после чего выполняем Sift Down для этого сына. Функция выполняется за время O(h), где h - высота дерева.

```
template <typename T>
void Heap<T>::SiftDown(int index) {
  while (HasLeftChild(index)) {
    size_t max_child_index = GetIndexOfMaxChild(index);
    if (container_[index] < container_[max_child_index]) {
      std::swap(container_[index], container_[max_child_index]);
      index = max_child_index;
    } else {
      break;
    }
  }
}
```

Def. Sift Up поднимает элемент, который больше родительского.

```
template <typename T>
void Heap<T>::SiftUp(int index) {
  while (index != 0) {
    if (container_[index] > container_[GetParentIndex(index)]) {
      std::swap(container_[index], container_[GetParentIndex(index)]);
      index = GetParentIndex(index);
    } else {
      break;
    }
  }
}
```

Задача. Создать кучу из неупорядоченного массива входных данных

Решение:

Если выполнить SiftDown для всех элементов массива A, начиная с последнего и кончая первым, он станет кучей.

SiftDown(A, i) не делает ничего, если $i \geqslant \frac{n}{2}$.

Достаточно вызвать SiftDown для всех элементов массива A с $(\frac{n}{2})$ по 0-й.

Утверждение. Время работы BuildHeap = O(n)

Доказательство:

Заметим, что число элементов на высоте h (считая от конца) $\leqslant \left[\frac{n}{2^{h+1}}\right]$

Общее время работы:

$$T(n) = \sum_{h=0}^{log(n)} \left[\frac{n}{2^h}\right] C * h = O\left(n \sum_{h=0}^{log(n)} \frac{h}{2^h}\right)$$

Воспользуемся формулой $\sum_{h=0}^{\inf} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

Таким образом $T(n) = O(n)$ ■

Добавление элемента в кучу

1. Добавим элемент в конец кучи
2. Восстановим свойство упорядоченности, проталкивая элемент наверх с помощью SiftUp.

Время работы – O(h), если буфер для кучи позволяет добавить элемент без переаллокации.

Извлечение максимума

1. Сохраним значение корневого элемента для возврата
2. Скопируем последний элемент в корень, удалим последний элемент

3. Вызовем SiftDown для корня

4. Возвратим сохраненный корневой элемент

Время работы - O(h)

Код кучи

```
template <typename ValueType, typename Comparator = std::less<ValueType>>
class Heap {
 public:
  Heap();

  Heap(std::shared_ptr<std::vector<ValueType>> vector);

  size_t GetSize() const;

  bool IsEmpty() const;

  std::optional<ValueType> Pop();

  std::optional<ValueType> GetRootValue() const;

  void Push(ValueType value);

 private:
  bool HasLeftChild(size_t index);

  bool HasRightChild(size_t index);

  size_t GetLeftChildIndex(size_t index);

  size_t GetRightChildIndex(size_t index);

  size_t GetParentIndex(size_t index);

  ValueType& At(size_t index);

  void SiftUp(int index);

  void SiftDown(int index);

  Comparator Comp;

  size_t size_{0};
```

```
  std::shared_ptr<std::vector<ValueType>> container_;
};

template <typename ValueType, typename Comparator>
Heap<ValueType, Comparator>::Heap() :
  container_{std::make_shared<std::vector<ValueType>>()} {}

template <typename ValueType, typename Comparator>
Heap<ValueType, Comparator>::Heap(
    std::shared_ptr<std::vector<ValueType>> vector)
    : container_{vector}, size_{vector->size()} {
  for (int i = GetSize() / 2; i >= 0; --i) {
    SiftDown(i);
  }
}

template <typename ValueType, typename Comparator>
size_t Heap<ValueType, Comparator>::GetSize() const {
  return size_;
}

template <typename ValueType, typename Comparator>
bool Heap<ValueType, Comparator>::IsEmpty() const {
  return size_ == 0;
}

template <typename ValueType, typename Comparator>
std::optional<ValueType> Heap<ValueType, Comparator>::Pop() {
  if (IsEmpty()) {
    return {};
  }

  ValueType root_value = GetRootValue();

  std::swap(At(0), At(GetSize() - 1));
  container_->erase(std::prev(container_->end()));

  --size_;

  SiftDown(0);

  return root_value;
}

template <typename ValueType, typename Comparator>
std::optional<ValueType>
Heap<ValueType, Comparator>::GetRootValue() const {
```

```
  if (IsEmpty()) {
    return {};
  }
  return container_->front();
}

template <typename ValueType, typename Comparator>
void Heap<ValueType, Comparator>::Push(ValueType value) {
  container_->push_back(value);
  ++size_;
  SiftUp(GetSize() - 1);
}

template <typename ValueType, typename Comparator>
bool Heap<ValueType, Comparator>::HasLeftChild(const size_t index) {
  return GetLeftChildIndex(index) < GetSize();
}

template <typename ValueType, typename Comparator>
bool Heap<ValueType, Comparator>::HasRightChild(const size_t index) {
  return GetRightChildIndex(index) < GetSize();
}

template <typename ValueType, typename Comparator>
size_t Heap<ValueType, Comparator>::GetLeftChildIndex(const size_t index) {
  return 2 * index + 1;
}

template <typename ValueType, typename Comparator>
size_t Heap<ValueType, Comparator>::GetRightChildIndex(const size_t index) {
  return 2 * index + 2;
}

template <typename ValueType, typename Comparator>
size_t Heap<ValueType, Comparator>::GetParentIndex(const size_t index) {
  return (index - 1) / 2;
}

template <typename ValueType, typename Comparator>
ValueType& Heap<ValueType, Comparator>::At(size_t index) {
  return (*container_.get())[index];
}

template <typename ValueType, typename Comparator>
void Heap<ValueType, Comparator>::SiftUp(int index) {
  while (index != 0) {
    if (Comp(At(index), At(GetParentIndex(index)))) {
```

```
            std::swap(At(index), At(GetParentIndex(index)));
            index = GetParentIndex(index);
          } else {
            break;
          }
        }
      }

      template <typename ValueType, typename Comparator>
      void Heap<ValueType, Comparator>::SiftDown(int index) {
        while (HasLeftChild(index)) {
          size_t max_or_min_child_index = 0;
          if (HasRightChild(index) &&
              Comp(At(GetRightChildIndex(index)), At(GetLeftChildIndex(index)))) {
            max_or_min_child_index = GetRightChildIndex(index);
          } else {
            max_or_min_child_index = GetLeftChildIndex(index);
          }
          if (Comp(At(max_or_min_child_index), At(index))) {
            std::swap(At(index), At(max_or_min_child_index));
            index = max_or_min_child_index;
          } else {
            break;
          }
        }
      }
```

АТД очередь с приоритетом

Def. Очередь с приоритетом — абстрактный тип данных, поддерживающий три операции:

1. InsertWithPriority — добавить в очередь элемент с назначенным приоритетом.
2. GetNext — извлечь из очереди и вернуть элемент с наивысшим приоритетом. Другие названия: «PopElement», «GetMaximum».

Note. Очередь с приоритетом можно реализовать с помощью кучи

Тема 2. Сортировки и порядковые статистики.

Формулировка задачи. Устойчивость, локальность.

Def. Сортировка - процесс упорядочивания каких-либо сравнимых объектов

Def. Сортировка называется устойчивой, если она не меняет порядок равных с точки зрения операции сравнения элементов

Def. Сортировка называется локальной, если она не использует дополнительную память (M(n) = O(1))

Квадратичные сортировки: сортировка вставками, выбором.

Сортировка вставками

- Массив разделен на 2 части: левая - упорядочена, правая - нет
- На одном шаге:
  1. берем первый элемент правой части
  2. вставляем его на подходящее место в левой части

```
template <typename ValueType, typename Comparator = std::less<ValueType>>
void InsertionSort(std::vector<ValueType>& vector,
                   Comparator Compare = Comparator()) {
  for (int i = 1; i < vector.size(); ++i) {
    ValueType current_value = vector[i];

    int index_for_current_value = i;

    for (int j = i - 1; j >= 0 && Compare(current_value, vector[j]); --j) {
      vector[j + 1] = vector[j];
      index_for_current_value = j;
    }

    vector[index_for_current_value] = current_value;
  }
}
```

Анализ сортировки вставками

- Лучший случай O(n)
  - Массив упорядочен по возрастанию
  - 2(n - 1) копирований
  - n - 1 сравнений
- Худший случай O(n^2)

  - Массив упорядочен по убыванию
  - $2(n-1) + \frac{n(n-1)}{2}$ копирований
  - $\frac{n(n-1)}{2}$ сравнений

- В среднем O(n^2) Для анализа среднего случая нужно посчитать среднее число сравнений, необходимых для определения положения очередного элемента. При добавлении нового элемента потребуется, как минимум, одно сравнение, даже если этот элемент оказался в правильной позиции. i-й добавляемый элемент может занимать одно из i+1 положений. Предполагая случайные входные данные, новый элемент равновероятно может оказаться в любой позиции. Среднее число сравнений для вставки i-го элемента:

$$T_i = \frac{1}{i+1}\left(\sum_{p=1}^{i} p + i\right) = \frac{1}{i+1}\left(\frac{i(i+1)}{2} + i\right) = \frac{i}{2} + 1 - \frac{1}{i+1}$$

  Для оценки среднего времени работы для n элементов нужно просуммировать:

$$T(n) = \sum_{i=1}^{n-1} T_i = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \sum_{i=1}^{n-1}\frac{i}{2} + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1}\left(\frac{1}{i+1}\right)$$

$$T(n) \approx \frac{n^2-n}{4} + (n-1) - (\ln(n) - 1) = O(n^2)$$

Сортировка выбором

- Массив разделен на две части: левая - готова, правая - нет.
- 1. Ищем минимум в правой части
  2. Меняем его с первым элементом правой части
  3. Сдвигаем границу разделения на 1 вправо

$\frac{n(n-1)}{2}$ сравнений, $n-1$ перемещений. $T(n) = \Theta(n^2)$

Сортировка слиянием.

Алгоритм

1. Разбить массив на два
2. Отсортировать каждый(рекурсивно)
3. Слить отсортированные в один

Вариант без рекурсии

1. Разбить на 2^k подмассива, $2^k < n$
2. Отсортировать каждый
3. - Слить 1 и 2, 3 и 4, ..., $2^k - 1$ и 2^k
   - Слить 12 и 34, 56 и 78 ...
   - ...
   - Слить $123 \dots 2^{k-1}$ и $2^{k-1} + 1 \dots 2^k$

Слияние двух отсортированных массивов

- Выберем массив, крайний элемент которого меньше
- Извлечем этот элемент в массив-результат
- Продолжим, пока один из массивов не опустеет
- Копируем остаток второго массива в конец массива-результата

```
template <typename ValueType, typename Iterator,
          typename Comparator = std::greater<ValueType>>
void Merge(Iterator first_begin, Iterator first_end, Iterator second_begin,
           Iterator second_end, Comparator Compare = Comparator()) {
  size_t full_size = (first_end - first_begin) + (second_end - second_begin);
  std::vector<ValueType> merge_vector;
  auto first_iter = first_begin;
  auto second_iter = second_begin;

  while (first_iter != first_end && second_iter != second_end) {
    if (Compare(*second_iter, *first_iter)) {
      merge_vector.push_back(*first_iter);
      first_iter = std::next(first_iter);
    } else {
      merge_vector.push_back(*second_iter);
      second_iter = std::next(second_iter);
    }
  }

  if (first_iter == first_end) {
    for (; second_iter != second_end; second_iter = std::next(second_iter)) {
      merge_vector.push_back(*second_iter);
    }
  } else if (second_iter == second_end) {
    for (; first_iter < first_end; first_iter = std::next(first_iter)) {
      merge_vector.push_back(*first_iter);
```

```
        }
      }

      auto merged_vector_iter = merge_vector.begin();
      for (first_iter = first_begin; first_iter != first_end;
           first_iter = std::next(first_iter)) {
        *first_iter = *merged_vector_iter;
        merged_vector_iter = std::next(merged_vector_iter);
      }

      for (second_iter = second_begin; second_iter != second_end;
           second_iter = std::next(second_iter)) {
        *second_iter = *merged_vector_iter;
        merged_vector_iter = std::next(merged_vector_iter);
      }
    };
```

- Сложность: $T(n, m) = O(n + m)$
- Количество сравнений
  - В лучшем случае $min(n, m)$
  - В худшем случае $n + m - 1$

Сама сортировка

```
    template <typename ValueType, typename Iterator,
              typename Comparator = std::greater<ValueType>>
    void MergeSort(Iterator begin, Iterator end,
                              Comparator Compare = Comparator()) {
      if (end - begin <= 1) {
        return 0;
      }

      auto middle = std::next(begin, container_size / 2);

      MergeSort<ValueType>(begin, middle, Compare);
      MergeSort<ValueType>(middle, end, Compare);
      Merge<ValueType>(begin, middle, middle, end, Compare, &inversion_count);
    }
```

Утверждение. Время работы сортировки слиянием = $O(nlog(n))$

Доказательство:

Рекуррентное соотношение

$$T(n) \leqslant 2 * T\left(\frac{n}{2}\right) + c * n$$

разложим дальше

$$T(n) \leqslant 2 * T\left(\frac{n}{2}\right) + c * n \leqslant 4T\left(\frac{n}{4}\right) + 2cn \leqslant \ldots \leqslant 2^k T(1) + k * c * n$$

$k = log(n)$, следовательно $T(n) = O(nlog(n))$ ■

Пирамидальная сортировка

Remind. Операции в бинарной куче:

- SiftDown O(log(n))
- SiftUp O(log(n))
- Добавление элемента O(log(n))
- Извлечение максимума O(log(n))
- Построение O(n)

Алгоритм пирамидальной сортировки

$T(n) = O(nlog(n))$

1. Строим кучу на исходном массиве.
2. $N-1$ раз достаем максимальный элемент, кладем его на освободившееся место в правой части.

Слияние K отсортированных массивов с помощью кучи.

Задача. Дано k отсортированных массивов $A_1, A_2, \ldots, A_k$. Нужно слить их в один массив.

Решение:

1)Построим min-heap из первых элементов $A_1[0], A_2[0], \ldots, A_k[0]$.
2)Пока куча не пуста:

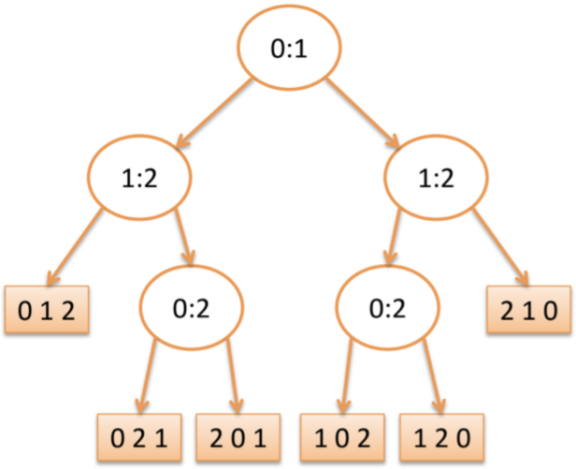
- Скопируем минимум из кучи в результат
- Если минимальный элемент - последний в своем массиве, то извлечем его из кучи, иначе - заменим его следующим из того же массива
- Восстановим свойство кучи - SiftDown(0)

$T(n,k) = O(k + nlog(k)), M(n,k) = O(k)$.

Нижняя оценка времени работы для сортировок сравнением.

Утверждение. Время работы любого алгоритма сортировки, использующего сравнение - $\Omega(Nlog(N)$

Доказательство:



В процессе работы алгоритма сравниваются элементы исходного массива.
Ветвление - дерево
Окончание работы алгоритма - лист
Лист - перестановка исходного массива
Всего листьев в дереве решения не меньше N!
Высота дерева не меньше $log(N!) \approx CNlog(N)$
Следовательно, существует перестановка, на которой алгоритм делает не менее $CNlog(N)$ сравнений ■

Быстрая сортировка. Выбор опорного элемента. Доказательство среднего времени работы.

Идея алгоритма

1. Разделим массив на 2 части таким образом, что элементы в левой части $\leqslant$ элементов в правой части
2. Применим эту процедуру рекурсивно к левой и правой части

Partition

Разделим массив A. Выберем разделяющий элемент - пивот. Пусть пивот лежит в конце массива.

1. Установим 2 указателя: i,j в начало массива

2. Двигаем j вправо, пока не встретим элемент $\leqslant$ пивота
3. Если встретили элемент $\leqslant$ пивота, то меняем A[i] и A[j] местами(если i != j), двигаем i и j вправо
4. Меняем A[i] и A[n-1] (пивот)

Анализ быстрой сортировки

- Если Partition всегда делит пополам, то

$$T(n) \leqslant 2T(\frac{n}{2}) + cn$$

Следовательно, $T(n) = O(nlog(n))$

- Если массив упорядочен, пивот $= A[n-1]$, то массив делится в отношении n-1 : 0.

$$T(n) \leqslant T(n-1) + cn \leqslant T(n-2) + c(n+n-1)$$

Следовательно, $T(n) = O(n^2)$

Утверждение. В среднем $T(n) = O(nlog(n))$

Доказательство:

Пусть X полное количество сравнений элементов с опорным за время работы сортировки. Нам необходимо вычислить полное количество сравнений.

Переименуем элементы массива как $z_1 \dots z_n$, где z_1 наименьший по порядку элемент. Также введем множество $Z_{ij} = \{z_i, z_{i+1} \dots z_j\}$.

Заметим, что сравнение каждой пары элементов происходит не больше одного раза, так как элемент сравнивается с опорным, а опорный элемент после разбиения больше не будет участвовать в сравнении.

Поскольку каждая пара элементов сравнивается не более одного раза, полное количество сравнений выражается как

$X = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} X_{ij}$, где $X_{ij} = 1$ если произошло сравнение z_i и z_j и $X_{ij} = 0$, если сравнения не произошло.

Применим к обеим частям равенства операцию вычисления матожидания и воспользовавшись ее линейностью получим

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr\{z_i \text{ сравнивается с } z_j\}$$

Осталось вычислить величину $Pr\{z_i$ сравнивается с $z_j\}$ вероятность того, что z_i сравнивается с z_j. Поскольку предполагается, что все элементы в массиве различны, то при выборе x в качестве опорного элемента впоследствии не будут сравниваться никакие z_i и z_j для которых $z_i < x < z_j$. С другой стороны, если z_i выбран в качестве опорного, то он будет сравниваться с каждым элементом Z_{ij} кроме себя самого. Таким образом элементы z_i и z_j сравниваются тогда и только тогда когда первым в множестве Z_{ij} опорным элементом был выбран один из них.

$Pr\{z_i$ сравнивается с $z_j\} = Pr\{$первым опорным элементом был z_i или $z_j\} = Pr\{$первым опорным элементом был $z_i\} + Pr\{$первым опорным элементом был $z_j\} =$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

Матожидание времени работы быстрой сортировки будет $O(n \log n)$.

TimSort.

Def. Timsort — гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году Тимом Петерсом.

1. Вычисление minRun - минимального размера подмассива
2. Сортировка вставками каждого run
3. Слияние соседних run

Вычисление minRun

Берутся старшие 6 бит числа n и прибавляется 1, если в младших битах была хоть одна единица

```
int GetMinrun(int n) {
    int r = 0;
    while (n >= 64) {
        r |= n & 1;
        n >>= 1;
    }
    return n + r;
}
```

Вычисление run'ов и их сортировка

Собираем run:

- Ищем максимально отсортированный подмассив, начиная с текущей позиции
- Разворачиваем его, если он отсортирован по убыванию
- Дополняем отсортированный подмассив до minRun

Сортируем вставками каждый run

- Отсортированную часть run'а заново не сортируем, только новые элементы вставляем на свои места

Выполняем слияние соседних run'ов

1. Создается пустой стек пар <индекс начала подмассива, размер подмассива >.
2. Берется первый упорядоченный подмассив.
3. Добавляется в стек пара данных <индекс начала текущего подмассива, его размер >.
4. Пусть X,Y,Z — длины верхних трех интервалов, которые лежат в стеке. Причем X — это последний элемент стека.
5. Повторяем пока выражение $(Z > X + Y \wedge Y > X)$ не станет истинным
   Если размер стека не меньше 3 и $Z \leqslant X + Y$ сливаем Y с $\min(X, Z)$.
   Иначе Если $Y \leqslant X$ сливаем X с Y.
6. Если всего осталось 3 подмассива, которые сейчас в стеке, то сливаем их в правильном порядке, иначе же переходим к шагу 3.

Модификация слияния - галоп

1. Начинается процедура слияния.
2. На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминается, из какого именно подмассива был элемент.
3. Если уже некоторое количество элементов (например, в JDK 7 это число равно 7) было взято из одного и того же массива — предполагается, что и дальше придётся брать данные из него. Чтобы подтвердить эту идею, алгоритм переходит в режим галопа, то есть перемещается по массиву-претенденту на поставку следующей большой порции данных бинарным поиском (массив упорядочен) текущего элемента из второго соединяемого массива.

4. В момент, когда данные из текущего массива-поставщика больше не подходят (или был достигнут конец массива), данные копируются целиком.

Сортировка подсчетом. Карманная сортировка.

Задача. Отсортировать массив $A[0 \dots n-1]$, содержащий неотрицательные целые числа меньшие k.

Решение 1

- Заведем массив $C[0 \dots k-1]$, посчитаем в $C[i]$ количество вхождений элемента i в массиве A
- Выведем все элементы $\{0 \dots k\}$ по $C[i]$ раз

Решение 2(карманная сортировка подсчетом)

- Заведем массив $C[0 \dots k-1]$, посчитаем в $C[i]$ количество вхождений элемента i в массиве A
- Вычислим границы групп элементов для каждого $iin\{0, \dots, k\}$(начальные позиции каждой группы). (Просто массив префиксных сумм)
- Создадим массив для результата B
- Переберем массив A. Очередной элемент A[i] разместим в B в позиции $C[A[i]]$. Сдвинем текущую границу группы.
- Скопируем B в A

Note. Сортировка подсчетом - стабильная, но не локальная

Время работы $T(n,k) = O(n+k)$ // Дополнительная память $T(n,k) = O(n+k)$

Поразрядная сортировка.

Если диапазон значений велик, то сортировка подсчетом не годится. Строки, целые числа можно разложить на разряды. Диапазон значений разряда не велик. Можно выполнять сортировку массива по одному разряду, используя сортировку подсчетом.

MSD, LSD. Сортировка строк.

Note. LSD - Least Significant Digit

Сначала сортируем подсчетом по младшим разрядам, затем по старшим. Ключи с различными младшими разрядами, но одинаковыми старшими не будут перемешаны при сортировки старших разрядов благодаря стабильности поразрядной сортировки.

```
void CountingSort(long long* a, int n, int byte);
void LSDSort(long long* a, int n) {
  for (int r = 0; r < sizeof(long long); ++r) {
    CountingSort(a, n, r);
  }
}
```

Время работы $T(n,k,r) = O(r(n+k))$
Дополнительная память $M(n,k,r) = O(n+k$
где n - размер массива, k - размер алфавита, r - количество разрядов.

Note. MSD - Most Significant Digit

Сначала сортируем подсчетом по старшим разрядам, затем по младшим. Чтобы не перемешать отсортированные старшие разряды, сортируем по младшим только группы чисел с одинаковыми старшими разрядами отдельно друг от друга.

Время работы $T(n,k,r) = O(nrk)$
Дополнительная память $M(n,k,r) = O(n+k)$
где n - размер массива, k - размер алфавита, r - количество разрядов.

Note. MSD хорошо подходит для сортировки строк разной длины, потому что группы с текущим разрядом '\0' можно не сортировать.

Binary QuickSort.

Похожа на MSD по битам

1. Сортируем по старшему биту. Вместо поразрядной сортировки вызываем Partition с фиктивным опорным элементом 100...0 где 1 стоит на месте бита, по которому сортируем.
2. Рекурсивно вызываем от левой и правой части

Время работы $T(n,r) = O(rn)$
Дополнительная память $M(n,r) = O(r)$ - для рекурсии

Note. Эта сортировка не стабильна, но локальна.

Поиск k-ой порядковой статистики методом QuickSelect.

Def. К-ой порядковой статистикой называется элемент, который окажется на К-ой позиции после сортировки массива.

Поиск K-ой порядковой статистики методом «Разделяй и властвуй». KStatDC(A, n, K).

1. Выбираем пивот, вызываем Partition.
2. - Если $P == K$, то пивот является K-ой порядковой статистикой
   - Если $P > K$, то к-я порядковая статистика находится слева. Вызываем $KStatDC(A, n - P, K)$
   - Если $P < K$, то k-я порядковая статистика находится справа, вызываем $KStatDC(A + (P + 1), n - (P + 1), K - (P + 1))$

Время работы

- $T(n) = O(n)$ в лучшем
- $T(n) = O(n)$ в среднем
- $T(n) = O(n^2)$ в худшем

Доказательство:

(Доказательство среднего времени работы)
Будем оценивать количество сравнений. При поиске статистики в массиве размера n функция partition (точнее, одна из распространённых вариаций) совершает не более $n - 1$ сравнений. Далее, в зависимости от k выбирается левая или правая половины (или вообще алгоритм завершает работу). Оценку проводим сверху, то есть, будем считать, что каждый раз выбирается большая половина.

$$T(n) \leqslant \frac{1}{n}\sum_{k=1}^{n}\left(T\left(\max\left\{k-1; n-k\right\}\right)+n-1\right) = n-1+\frac{1}{n}\sum_{k=1}^{n}T(\max\{k-1; n-k\}) = n-1+\frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1}T(k)$$

Предположим, что $T(k) \leqslant ck$ для некоторой константы c и всех $k < n$ (будем доказывать оценку по индукции). Тогда верно неравенство:

$$T(n) = n - 1 + \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck$$

Преобразуем сумму из правой части равенства по формуле суммы арифметической прогрессии и оценим преобразованное выражение:

$$\sum_{k=\lfloor n/2 \rfloor}^{n-1} ck = \frac{1}{2}\left(\left\lceil \frac{n}{2} \right\rceil\right)\left(c\left\lfloor \frac{n}{2} \right\rfloor + c(n-1)\right) \leq \frac{c}{2}\left(\frac{n+1}{2}\right)\frac{3n-2}{2} = c\frac{n+1}{4}\frac{3n-2}{2}$$

Воспользуемся полученной оценкой для оценки исходного выражения. Также, предположим, что $c \geqslant 4$:

$T(n) \leq n - 1 + \frac{2c}{n}\frac{n-1}{4}\frac{3n-2}{2} = n - 1 + c\frac{n-1}{2n}\frac{3n-2}{2} \leqslant \frac{c}{4}(n-1) + \frac{c}{4}\left(\frac{n-1}{n}(3n-2)\right) \leqslant \frac{c}{4}(n-1+3n-2) = \frac{c}{4}(4n-3) \leqslant cn$

Для завершения доказательства необходима проверка базы индукции, но она тривиальна: для выборки порядковой статистики из одного элемента сравнений не требуется: $T(1) = 0 < 4$. Итого, мы доказали, что $T(n) \leqslant 4n$, следовательно, $T(n) = O(n)$

Поиск K-ой порядковой статистики за линейное время. KStatLin(A, n, K).

1. Разобьем массив на пятерки.
2. Сортируем каждую пятерку, выбираем медиану из каждой пятерки.
3. Ищем M – медиану медиан пятерок, вызвав KStatLin(Medians, n/5, n/10).
4. Разделяем по пивоту M, вызывая обычный Partition.
5. Далее аналогично предыдущему алгоритму

Утверждение. Время работы алгоритма $T(n) = O(n)$

Доказательство:

Время работы не больше чем сумма:

1. Времени работы на сортировку групп и разбиение по рассекающему элементу, то есть Cn
2. Времени работы для поиска медианы медиан, то есть $T\left(\frac{n}{5}\right)$
3. времени работы для поиска k-го элемента в одной из двух частей массива, то есть $T(s)$, где s — количество элементов в этой части. Но s не превосходит $\frac{7n}{10}$, так как чисел, меньших рассекающего элемента, не менее $\frac{3n}{10}$ — это $\frac{n}{10}$ медиан, меньших медианы медиан, плюс не менее $\frac{2n}{10}$ элементов, меньших этих медиан. С другой стороны, чисел, больших рассекающего элемента, так же не менее $\frac{3n}{10}$, следовательно $s \leqslant \frac{7n}{10}$, то есть в худшем случае $s = \frac{7n}{10}$.

Тогда получаем, что $T(n) \leqslant T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + Cn$

Покажем, что для всех n выполняется неравенство $T(n) \leqslant 10Cn$.

Докажем по индукции:

1. Предположим, что наше неравенство $T(n) \leqslant 10Cn$ выполняется при малых n, для некоторой достаточно большой константы C.
2. Тогда, по предположению индукции, $T\left(\frac{n}{5}\right) \leqslant 10C\frac{n}{5} = 2Cn$ и $T\left(\frac{7n}{10}\right) \leqslant 10C\frac{7n}{10} = 7Cn$, тогда $T(n) \leqslant T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + Cn = 2Cn + 7Cn + Cn = 10Cn \Rightarrow T(n) \leqslant 10Cn$

Так как $T(n) \leqslant 10Cn$, то время работы алгоритма $O(n)$

Тема 3. Деревья поиска.

Определение дерева, дерева с корнем. Высота дерева, родительские, дочерние узлы, листья. Количество ребер.

Def. Дерево (свободное) – непустая коллекция вершин и ребер, удовлетворяющих определяющему свойству дерева.

Def. Вершина (узел) – простой объект, который может содержать некоторую информацию.

Def. Ребро – связь между двумя вершинами.

Def. Путь в дереве – список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева.

Def. Определяющее свойство дерева – существование только одного пути, соединяющего любые два узла.

Def. (равносильно первому) Дерево (свободное) – неориентированный связный граф без циклов.

Def. Дерево с корнем — дерево, в котором один узел выделен и назначен «корнем» дерева. Существует только один путь между корнем и каждым из других узлов дерева.

Def. Высота (глубина) дерева с корнем — количество вершин в самом длинном пути от корня.

Def. Каждый узел (за исключением корня) имеет только один узел, расположенный над ним. Такой узел называется родительским. Узлы, расположенные непосредственно под данным узлом, называются его дочерними узлами. Узлы, не имеющие дочерних узлов называются листьями.

Утверждение. Любое дерево (с корнем) содержит листовую вершину

Утверждение. Дерево, состоящее из N вершин, содержит $N - 1$ ребро.

Доказательство:

По индукции.

База индукции. $N = 1$. Одна вершина, 0 ребер.

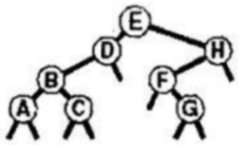
Шаг индукции. Пусть дерево состоит из N + 1 вершины. Найдем листовую вершину. Эта вершина инцидентна ровно одному ребру. Дерево без этой вершины содержит N вершин, а по предположению индукции N – 1 ребро. Следовательно, исходное дерево содержит N ребер. ■

Обходы в глубину. pre-order, post-order и in-order для бинарных деревьев.

Def. Пошаговый перебор элементов дерева по связям между узлами-предками и узлами-потомками называется обходом дерева.

Def. Обходом двоичного дерева в глубину (DFS) называется процедура, выполняющая в некотором заданном порядке следующие действия с поддеревом:

- просмотр (обработка) узла-корня поддерева,
- рекурсивный обход левого поддерева,
- рекурсивный обход правого поддерева.



Def. Прямой обход (сверху вниз, pre-order). Вначале обрабатывается узел, затем посещается левое и правые поддеревья.
Порядок обработки узлов дерева на рисунке: E, D, B, A, C, H, F, G.

Def. Обратный обход (снизу вверх, post-order). Вначале посещаются левое и правое поддеревья, а затем обрабатывается узел.
Порядок обработки узлов дерева на рисунке: A, C, B, D, G, F, H, E.

Def. Поперечный обход (слева направо, in-order). Вначале посещается левое поддерево, затем узел и правое поддерево.
Порядок обработки узлов дерева на рисунке: A, B, C, D, E, F, G, H.

Обход в ширину.

Def. Обход двоичного дерева в ширину (BFS) — обход вершин дерева по уровням (слоям), начиная от корня. BFS – Breadth First Search.

Используется очередь, в которой хранятся вершины, требующие просмотра. За одну итерацию алгоритма:

- если очередь не пуста, извлекается вершина из очереди,
- посещается (обрабатывается) извлеченная вершина,
- в очередь помещаются все дочерние.

Порядок обработки узлов дерева на рис.: E, D, H, B, F, A, C, G.

Дерево поиска.

Def. Двоичное дерево поиска (binary search tree, BST) – это двоичное дерево, с каждым узлом которого связан ключ, и выполняется следующее дополнительное условие:
Ключ в любом узле X больше или равен ключам во всех узлах левого поддерева X и меньше или равен ключам во всех узлах правого поддерева X.

Операции с двочиным деревом поиска:

1. Поиск по ключу
2. Поиск минимального, максимального ключей
3. Вствка
4. Удаление
5. Обход дерева в порядке возрастания ключей

Поиск ключа, вставка, удаление.

Задача. Дан указатель на корень дерева X и ключ K. Нужно проверить, есть ли узел с ключом K в дереве, и если да, то вернуть указатель на этот узел.

Решение:

Если дерево пусто, то узла нет. Иначе сравним K со значением ключа корневого узла X.

- Если $K == X$, то вернуть данный указатель
- Если K < X, рекурсивно искать ключ K в левом поддереве X
- Иначе рекурсивно искать ключ K в правом поддереве X

Время работы O(h), где h - высота дерева

Задача. Дан указатель на корень дерева X и ключ K. Нужно вставить узел с ключем K в дерево.

Решение:

Если дерево пусто, заменить его на дерево с одним корневым узлом K и остановиться. Иначе сравним K со значением ключа корневого узла X.

- Если K $<$ X, рекурсивно искать добавить K в левое поддерево X
- Иначе рекурсивно добавить ключ K в правое поддерево X

Время работы O(h), где h - высота дерева

Задача. Дан указатель на корень дерева X и ключ K. Нужно удалить узел с ключом K из дерева, если такой есть.

Решение:

Если дерево пусто, то остановиться. Иначе сравним K со значением ключа корневого узла X.

- Если K $<$ X, рекурсивно удалить K из левого поддерева X
- Если K $>$ X, рекурсивно удалить K из правого поддерева X
- Если $K == X$, то необходимо рассмотреть три случая:
  1. Дочерних узлов нет. Удаляем узел X, обнуляем ссылку
  2. Одного дочернего нет. Переносим дочерний узел в X, удаляем узел.
  3. Есть оба дочерних узла. Заменяем ключ удаляемого узла на ключ минимульного узла из правого поддерева, удаля последний.

Время работы O(h), где h - высота дерева

Необходимость балансировки. Три типа самобалансирующихся деревьев.

Note. Все перечисленные операции с деревом поиска выполняются за O(h), где h – глубина дерева. Глубина дерева может достигать n. Например, последовательное добавление возрастающих элементов вырождает дерево в цепочку:

Самобалансирующиеся деревья:

- Декартовы деревья
- AVL-деревья
- Красно-черные деревья
- Сплэй-деревья

Декартово дерево. Оценка средней высоты декартового дерева при случайных приоритетах (без доказательства).

Def. Декартово дерево — двоичное дерево, в узлах которого хранится пары (x, y) , где x – это ключ, а y – это приоритет. Все x и все y являются различными. Если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве $y < y_0$.

Th. В декартовом дереве из n узлов, приоритеты которого являются случайными величинами с равномерный распределением, средняя глубина дерева O(log(n)). Без доказательства.

Построение за O(n), если ключи упорядочены.

Будем строить дерево слева направо, то есть начиная с (x_1, y_1) по (x_n, y_n), при этом помнить последний добавленный элемент (x_k, y_k). Он будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой двоичное дерево поиска. При добавлении (x_{k+1}, y_{k+1}), пытаемся сделать его правым сыном (x_k, y_k), это следует сделать если $y_k > y_{k+1}$, иначе делаем шаг к предку последнего элемента и смотрим его значение y. Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе меньше приоритета в добавляемом, после чего делаем (x_{k+1}, y_{k+1}) его правым сыном, а предыдущего правого сына делаем левым сыном (x_{k+1}, y_{k+1}).
Заметим, что каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх (ведь после этого вершина будет лежать в чьем-то левом поддереве, а мы поднимаемся только по правому). Из этого следует, что построение происходит за $O(n)$.

Основные операции над декартовым деревом.

- Основные операции:
  - Разрезание - Split
  - Слияние - Merge
- На основе этих двух операций реализуются операции:
  - Вставка
  - Удаление

Разрезание - Split

Задача. Дано декартово дерево T, ключ K. Нужно разделить дерево T на T_1 и T_2 так, чтобы в T_1 находились все ключи меньшие K, а в T_2 не меньшие.

Решение:

```
template <typename ValueType>
std::pair<std::shared_ptr<Node>,
          std::shared_ptr<Node>>
Treap<ValueType>::Split(std::shared_ptr<Node> node, ValueType value) {
  if (!node) {
    return {nullptr, nullptr};
  }

  if (value > node->value) {
    auto [left, right] = Split(node->right_child, value);
    node->right_child = left;
    return {node, right};
  } else {
    auto [left, right] = Split(node->left_child, value);
    node->left_child = right;
    return {left, node};
  }
}
```

Задача. Дано два декартовых дерева T_1 и T_2. Причем все ключи T_1 меньше ключей T_2. Нужно объединить их в одно декартово дерево.

Решение:

```
template <typename ValueType>
std::shared_ptr<Node> Treap<ValueType>::Merge(
    std::shared_ptr<Node> left_tree,
    std::shared_ptr<Node> right_tree) {
  if (!left_tree) {
    return right_tree;
  }
  if (!right_tree) {
    return left_tree;
  }

  if (left_tree->priority > right_tree->priority) {
    left_tree->right_child = Merge(left_tree->right_child, right_tree);
    return left_tree;
  } else {
    right_tree->left_child = Merge(left_tree, right_tree->left_child);
    return right_tree;
  }
}
```

Задача. Дано декартово дерево T и ключ K. Нужно вставить ключ в дерево.

Решение:

1. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше y.
2. Теперь разрезаем поддерево найденного элемента на T_1 и T_2.
3. Полученные T_1 и T_2 записываем в качестве левого и правого сына добавляемого элемента.
4. Полученное дерево ставим на место элемента, найденного в первом пункте

Задача Дано декартово дерево T и ключ K. Нужно удалить узел с ключом K, если такой есть.

Решение:

1. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска), ища удаляемый элемент
2. Найдя элемент, вызываем слияние его левого и правого сыновей
3. Результат merge ставим на место удаляемого элемента

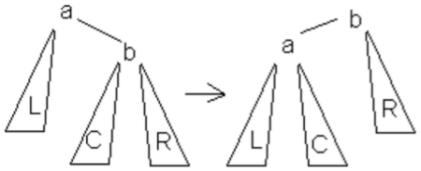
АВЛ-дерево. Вращения.

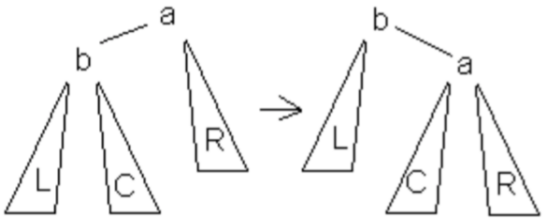
Специальные балансирующие операции, восстанавливающие основное свойство «высоты двух поддеревьев различаются не более чем на 1» – вращения.

- Малое левое вращение
- Малое правое вращение
- Большое левое вращение
- Большое правое вращение

Малое левое вращение

- Используется, когда:
  $H(R) = H(L) + 2$ и $H(C) \leqslant H(R)$
- После операции:
  высота дерева останется прежней, если $H(C) = H(R)$,
  высота дерева уменьшится на 1, если $H(C) < H(R)$.

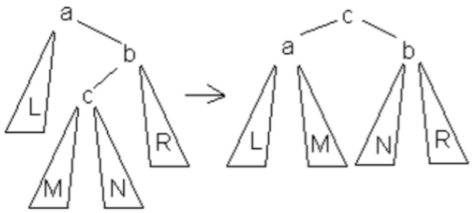




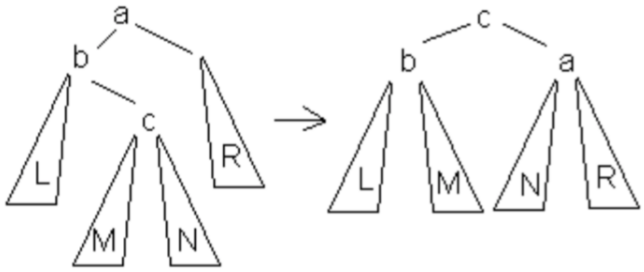
Малое правое вращение

Большое левое вращение

- Используется, когда $H(R) = H(L) + 1$ и $H(C) = H(L) + 2$
- После операции высота дерева уменьшается на 1



Малое правое вращение



Оценка высоты АВЛ-дерева.

Лемма. Пусть m_h минимальное число вершин в AVL-дереве высоты h, тогда $m_h = F_{h+2} - 1$, где $F_h - h$-ое число Фибоначчи.

Доказательство:

Если m_h минимальное число вершин в AVL-дереве высоты h. Тогда, как легко видеть, $m_{h+2} = m_{h+1} + m_h + 1$.
Равенство $m_h = F_{h+2} - 1$ докажем по индукции.
База индукции $m_1 = F_3 - 1$ верно, $m_1 = 1, F_3 = 2$.
Допустим $m_h = F_{h+2} - 1$ верно.
Тогда $m_{h+1} = m_h + m_{h-1} + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$.
Таким образом, равенство $m_h = F_{h+2} - 1$ ■

Th. Высота AVL-дерева с n ключами $h = O(log(n))$

Доказательство:

$F_h = \Omega(\varphi^h)$, $\varphi = \dfrac{\sqrt{5}+1}{2}$. То есть $n \geqslant c_1\varphi^h$
Логарифмируя по основанию φ, получаем

$$\log_\varphi nc_2 \geqslant h$$

Таким образом, получаем, что высота AVL-дерева из n вершин $O(\log n)$. ■

Вставка элемента

1. Вставляем элемент как в обычном двоичном дереве.
2. "Отступаем"назад от добавленной вершины к корню. Проверяем в каждой вершине сбалансированность. Если разность высот поддеревьев равна 2 – выполняем нужное вращение.
3. Время работы $T(n) = O(log(n))$

Удаление элемента

1. Удаляем элемент как в двоичном дереве поиска
2. "Отступаем"назад от удаленной вершины к корню. Проверяем в каждой вершине сбалансированность. Если разность высот поддеревьев равна 2 – выполняем нужное вращение.
3. Время работы $T(n) = O(log(n))$

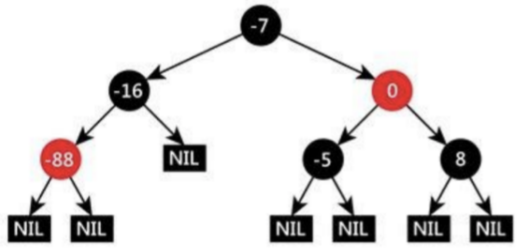
Красно-черное дерево.

Def. Красно-черное дерево – двоичное дерево поиска, у которого каждому узлу сопоставлен дополнительный атрибут – цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом.
2. Корень и конечные узлы (листья) дерева – чёрные.
3. У красного узла родительский узел – чёрный.
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов.

Def. Черная высота вершины x – число черных вершин на пути из x в лист, не учитывая саму вершину x.

Пример: Для вершин «-16» и «-88» черная высота = 1. Для вершин «-7» и «0» черная высота = 2.



Оценка высоты красно-черного дерева.

Лемма. В красно-черном дереве с черной высотой h_b количество внутренних вершин не менее $2^{h_b} - 1$

Доказательство:

Докажем по индукции.

База: для листьев черная высота равна 0, и поддерево действительно содержит $2^{h_b} - 1 = 0$ вершин.

Пусть наше предположение верно для высот до h'. Теперь рассмотрим внутреннюю вершину x с двумя потомками, для которой $hb(x) = h'$. Тогда если ее потомок p черный, то его высота $hb(p) = h' - 1$, а если красный, то $hb(p) = h'$. Но поскольку высота потомка меньше, чем высота узла x, для него выполняется индукционное предположение. В таком случае в поддереве узла x содержится не менее чем $2^{h'-1} - 1 + 2^{h'-1} - 1 + 1 = 2^{h'} - 1$.

Следовательно, утверждение верно и для всего дерева.

Th. Красно-черное дерево с N ключами имеет высоту $h = O(log(n))$

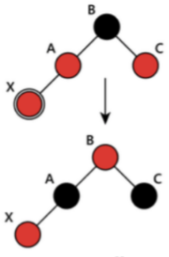
Доказательство:

Обозначим высоту дерева за h. Тогда согласно свойству 3 как минимум половина вершин на пути от корня до листа, не считая корень, составят черные вершины. Тогда черная высота дерева $\geqslant \frac{h}{2}$. По лемме получаем $n \geqslant 2^{\frac{h}{2}} - 1$. Получим $h \leqslant 2log(n+1)$ ■

Операции вставки и удаления в красно-черном дереве.

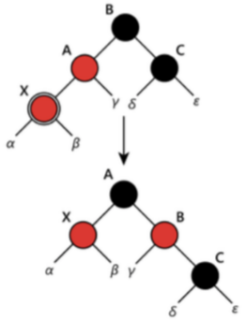
Вставка элемента.

Каждый элемент вставляется вместо листа.
Для выбора места вставки идём от корня в нужную сторону, как в наивном методе построения дерева поиска. До тех пор, пока не остановимся в листе (в фиктивной вершине).
Вставляем вместо листа новый элемент красного цвета с двумя листами-потомками.
Теперь восстанавливаем свойства красно-черного дерева.
Если отец нового элемента черный, то ничего делать не надо.
Если отец нового элемента красный, то достаточно рассмотреть только два случая:
Случай 1.



«Дядя» этого узла тоже красный. Тогда перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» - в красный. Теперь «дед» может нарушать свойство дерева. «Прадед» может быть красного цвета. Так рекурсивно пытаемся восстановить свойства дерева, двигаясь к предкам.
Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет.

Случай 2.



«Дядя» черный и правый. Просто выполнить перекрашивание отца в черный цвет нельзя, чтобы не нарушить постоянство чёрной высоты дерева по ветви с отцом.

1. Если добавленный узел X был правым потомков отца A, то необходимо выполнить левое вращение, которое сделает отца A левым потомком X.

2. Выполняем правый поворот A и B. Перекрашиваем A и B. Больше ничего делать не требуется.

Note. Если дядя левый, то порядок действий симметричен

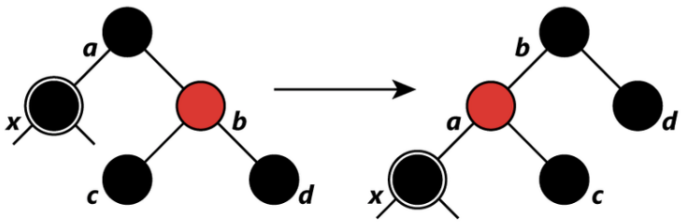
Общее время работы вставки - $O(log(n))$

Удаление вершины
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на nil
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

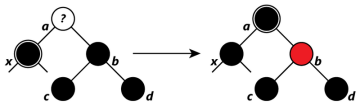
Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

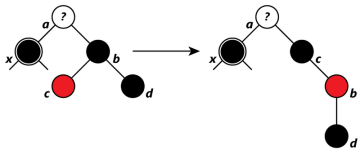


2. Если брат текущей вершины был черным, то получаем три случая:

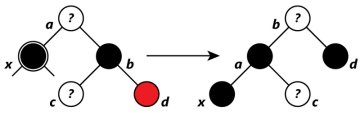
- Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влиянние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.



- Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.



- Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.



Сплей-дерево. Операция Splay.

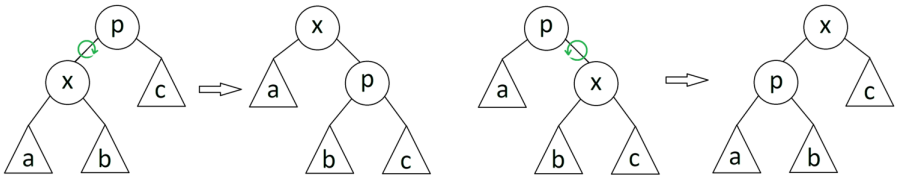
Def. Splay-дерево – двоичное дерево поиска с выделенной операцией Splay, использующейся в операциях Find, Add, Remove.
Операция Splay применяется к узлу x дерева:

- x становится корнем
- время работы $= O(h(x))$, где $h(x)$ - глубина узла x.
- амортизированное время работы $= O(log(n))$

Def. Операция Splay - чередует различные виды поворотов, благодаря чему достигается логарифмическая амортизированная оценка.
Splay делится на 3 случая:

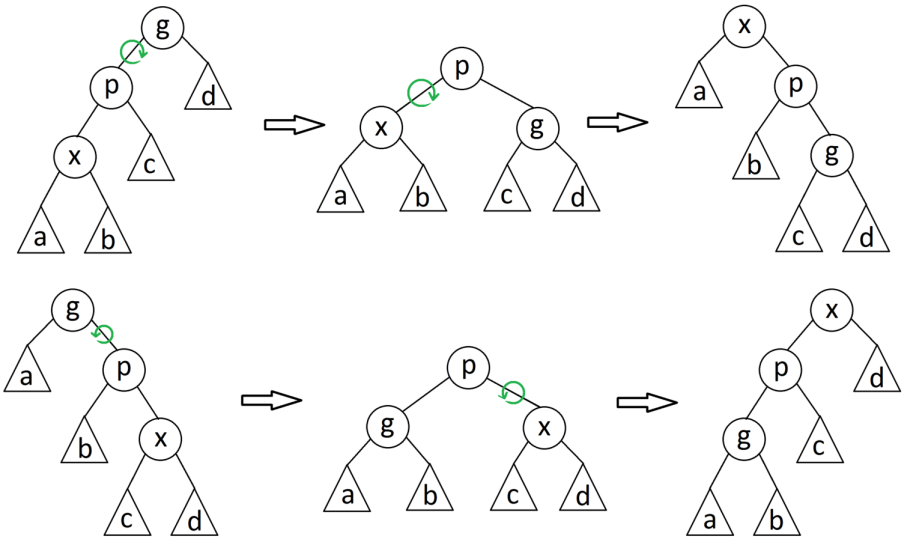
- zig.

  Если p — корень дерева с сыном x, то совершаем один поворот вокруг ребра (x, p), делая x корнем дерева. Данный случай является крайним и выполняется только один раз в конце, если изначальная глубина x была нечетной.



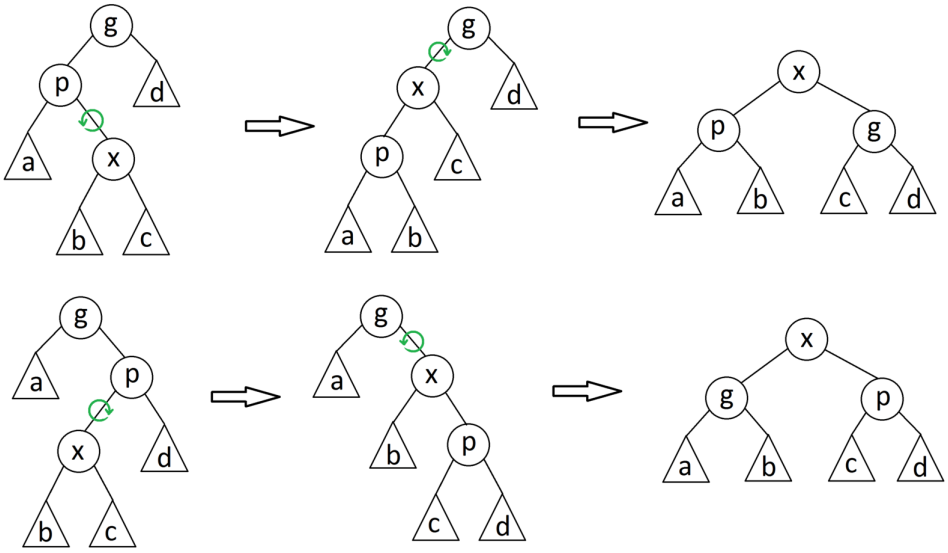
- zig-zig

  Если p — не корень дерева, а x и p — оба левые или оба правые дети, то делаем поворот ребра (p, g), где g отец p, а затем поворот ребра (x, p).



- zig-zag

  Если p — не корень дерева и x — левый ребенок, а p — правый, или наоборот, то делаем поворот вокруг ребра (x, p), а затем поворот нового ребра (x, g), где g — бывший родитель p.



Поиск, вставка, удаление в сплей-дереве.

find

Операция выполняется как для обычного бинарного дерева, только после нее щапускается операция splay

split

Для разделения дерева найдем наименьший элемент, больший или равный x, и сделаем для него Splay. После этого отрезаем у корня левого ребёнка и возвращаем 2 получившихся дерева.

merge

Для слияния деревьев T1 и T2, в которых все ключи T1 меньше ключей в T2, делаем Splay для максимального элемента T1, тогда у корня T1 не будет правого ребенка. После этого делаем T2 правым ребенком T1.

add

Запускаем split(tree, x), который нам возвращает деревья tree1 и tree2. Их подвешиваем к x как левое и правое поддеревья соответственно.

delete

Запускаем splay от элемента x и возвращаем Merge его поддеревьев

Учетная оценка операций в сплей-дереве $= O(logn)$ без доказательства.

Утверждение. Амортизированное время работы операции Splay $= O(log(n))$

Note. из этого утверждения вытекает, что амортизированное время работы любой операции в Splay-

дереве $= O(log(n))$

B-деревья.

Def. B-дерево — сильноветвящееся сбалансированное дерево поиска, позволяющее проводить поиск, добавление и удаление элементов за $O(logn)$.

B-дерево является идеально сбалансированным, то есть глубина всех его листьев одинакова. B-дерево имеет следующие свойства (t — параметр дерева, называемый минимальной степенью B-дерева, не меньший 2.):

- Каждый узел, кроме корня, содержит не менее $t-1$ ключей, и каждый внутренний узел имеет по меньшей мере t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.
- Каждый узел, кроме корня, содержит не более $2t-1$ ключей и не более чем $2t$ сыновей во внутренних узлах
- Корень содержит от 1 до $2t-1$ ключей, если дерево не пусто и от 2 до $2t$ детей при высоте большей 0.
- Каждый узел дерева, кроме листьев, содержащий ключи $k_1, \ldots, k_n$, имеет $n+1$ сына. i-й сын содержит ключи из отрезка $[k_{i-1}; k_i]$, $k_0 = -\infty$, $k_{n+1} = \infty$.
- Ключи в каждом узле упорядочены по неубыванию.
- Все листья находятся на одном уровне.

Note. B-деревья разработаны для использования на дисках (в файловых системах) или иных энергонезависимых носителях информации с прямым доступом, а также в базах данных.

Поиск ключа

Если ключ содержится в текущем узле, возвращаем его. Иначе определяем интервал и переходим к соответствующему сыну. Повторяем пока ключ не найден или не дошли до листа.

Добавление ключа

Ищем лист, в который можно добавить ключ, совершая проход от корня к листьям. Если найденный узел не заполнен, добавляем в него ключ. Иначе разбиваем узел на два узла, в первый добавляем первые t - 1 ключей, во второй — последние t - 1 ключей. После добавляем ключ в один из этих узлов. Оставшийся средний элемент добавляется в родительский узел, где становится разделительной

точкой для двух новых поддеревьев.
Если и родительский узел заполнен — повторяем пока не встретим незаполненный узел или не дойдем до корня. В последнем случае корень разбивается на два узла и высота дерева увеличивается. Добавление ключа в B-дереве может быть осуществлена за один нисходящий проход от корня к листу. Для этого не нужно выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. При проходе от корня к листьям в поисках места для нового ключа будут разбиваться все заполненные узлы, которые будут пройдены (включая и сам лист). Таким образом, если надо разбить какой-то полный узел, гарантируется, что его родительский узел не будет заполнен.

Удаление ключа

Если корень одновременно является листом, то есть в дереве всего один узел, мы просто удаляем ключ из этого узла. В противном случае сначала находим узел, содержащий ключ, запоминая путь к нему. Пусть этот узел — x.
Если x — лист, удаляем оттуда ключ. Если в узле x осталось не меньше $t-1$ ключей, мы на этом останавливаемся. Иначе мы смотрим на количество ключей в следующем, а потом в предыдущем узле. Если следующий узел есть, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и следующим узлом, а на его место ставим первый ключ следующего узла, после чего останавливаемся. Если это не так, но есть предыдущий узел, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и предыдущим узлом, а на его место ставим последний ключ предыдущего узла, после чего останавливаемся. Наконец, если и с предыдущим ключом не получилось, мы объединяем узел x со следующим или предыдущим узлом, и в объединённый узел перемещаем ключ, разделяющий два узла. При этом в родительском узле может остаться только $t-2$ ключей. Тогда, если это не корень, мы выполняем аналогичную процедуру с ним. Если мы в результате дошли до корня, и в нём осталось от 1 до $t-1$ ключей, делать ничего не надо, потому что корень может иметь и меньше $t-1$ ключей. Если же в корне не осталось ни одного ключа, исключаем корневой узел, а его единственный потомок делаем новым корнем дерева.
Если x — не лист, а K — его i-й ключ, удаляем самый правый ключ из поддерева потомков i-го потомка x, или, наоборот, самый левый ключ из поддерева потомков $i+1$-го потомка x. После этого заменяем ключ K удалённым ключом. Удаление ключа происходит так, как описано в предыдущем абзаце.

Тема 4. Хеш-таблицы.

Хеш-функции. Остаток от деления, мультипликативная.

Def. Хеш-функция — преобразование по детерминированному алгоритму входного массива данных произвольной длины (один ключ) в выходную битовую строку фиксированной длины (значение).

Результат вычисления хеш-фукнции называют «хешем».

Метод деления.

$h(k) = kmodM$

M определяет размер диапазона значений: $[0, \ldots, M-1]$

Как выбрать M?

- Если $M = 2^k$, то значение хэш-функции не зависит от старших байтов
- Если $M = 2^8 - 1$, то значение хеш-функции не зависит от перестановки байт.

Обычно в качестве M выбирают простое число, далекое от степеней двойки.

Метод умножения.

$h(k) = [M * \{k * A\}]$

где {} - дробная часть, [] - целая часть,

A - действительное число, $0 < A < 1$.

M определяет размер диапазона значений h: $[0, \ldots, M-1]$.

Кнут предложил следующее A:

$$A = \varphi^{-1} = \left(\frac{\sqrt{5}-1}{2}\right) = 0.6180339887\ldots$$

Такой выбор A дает хорошие результаты хэширования.

Эту функцию вычисляют без использования операция над числами с плавающими точками.

Пусть M - степень двойки. M$= 2^p, p \leqslant 32$.

Вместо действительного числа A берут близкое к нему $A = \frac{s}{2^32} = \frac{2654435769}{2^32}$. То есть $s = 2654435769$.

Тогда $h(k) = \left[2^p * \left\{k * \frac{s}{2^{32}}\right\}\right] = \left[2^p * \left\{\frac{r_1 * 2^{32} + r_0}{2^{32}}\right\}\right] = \left[2^p * \frac{r_0}{2^{32}}\right] = \left[\frac{r_0}{2^{32-p}}\right] = \left[\frac{r_{01}2^{32-p} + r_{00}}{2^{32-p}}\right] = r_{01} =$ старшие p бит r_0.

Итого, $h(k) = (k * smod2^{32}) >> (32 - p)$.

Деление многочленов. CRC.

Каждый ключ K задается числами $(k_0, \ldots, k_{n-1})$

Пусть K(x) $= k_0 + k_1 x + \ldots + k_{n-1}x^{n-1}$

Пусть задан полином $P(x) = p_0 + p_1 x + \ldots + p_m x^m$ - порождающий многочлен.

Определим хэш-функцию как остаток от деления:

$$H_p(K)(x) = K(x)modP(x) = h_0 + h_1 x + \ldots + h_{m-1}x^{m-1}$$

Все коэффициенты в поле Z_p, где p - простое.

CRC - циклически избыточный код, Cyclic redundancy check. Использует немного измененный метод

деления многочленов:

$$H_p(K)(x) = K(x) * x^m mod P(x)$$

коэффициенты в поле Z_2.

Обзор криптографических хеш-функций. CRC*, MD*, SHA*.

Note. Уточняется у лектора, что здесь нужно написать

Полиномиальная. Ее использование для строк. Метод Горнера для уменьшения количества операций умножения при ее вычислении.

Пусть есть строка $s = s_0, \ldots, s_{n-1}$. Есть два варианта полиномиальной функции:

$$h_1(s) = (s_0 + s_1 a + s_2 a^2 + \ldots + s_{n-1} a^{n-1}) mod M$$

$$h_2(s) = (s_0 a^{n-1} + s_1 a^{n-2} + s_2 a^{n-3} + \ldots + s_{n-1}) mod M$$

Как выбрать a и M? Часто M берут как степень двойки.

В идеале при изменении одного символа хэш-функция должна изменяться.

Th. 1)Если a и M не являются взаимно простыми, то $\{s * a mod M, 0 \leqslant s < M\} \neq \{0, \ldots, M-1\}$

2)Если a и M взаимно просты, то $\{s * a mod M, 0 \leqslant s < M\} = \{0, \ldots, M-1\}$

Доказательство:

1) Пусть a и M не являются взаимно простыми. Тогда a и M имеют общий делитель d.

a = d*x, M = d*y

$s * a = M * k + r \Rightarrow r = s * d * x - d * y * k = d(sx - yk)$

2) От противного. Пусть множество $\{s * a mod M, 0 \leqslant s < M\}$ имеет меньше M различных элементов.

Тогда существуют такие i и j, что $ia \equiv ja(mod M), i < j < M \Rightarrow (j-i)a \equiv 0(mod M) \Rightarrow i = j$?!.

Метод Горнера

$$h_2(s) = (((s_0 a + s_1)a + s_2)a + \ldots s_{n-2})a + s_{n-1}$$

Хеш-таблицы. Понятие коллизии.

Def. Коллизией хеш-функции H называется два различных входных блока данных X и Y таких, что $H(x) = H(y)$.

Def. Хеш-таблица – СД, хранящая ключи в таблице (массиве).Индекс ключа вычисляется с помощью хеш-функции.

Пусть хеш-таблица имеет размер M, число элементов в хеш-таблице - N.

Def. Число хранимых элементов, делённое на размер массива (число возможных значений хеш-функции), называется коэффициентом заполнения хеш-таблицы (load factor). Обозначим его $\alpha = \frac{N}{M}$

Метод цепочек (открытое хеширование).

Каждая ячейка массива является указателем на связный список (цепочку). Из-за коллизии появляются цепочки длиной более одного элемента.

Добавление ключа.

1. Вычисляем значение хэш-функции добавляемого ключа - h
2. Находим $A[h]$ - указатель на список ключей
3. Вставляем в начало списка.

Время работы:

- В лучше случае - $O(1)$
- В худшем случае
  - если не требуется проверять наличие дубля, то $O(1)$
  - иначе - $O(N)$

Удаление ключа.

1. Вычисляем значение хэш-функции добавляемого ключа - h
2. Находим $A[h]$ - указатель на список ключей
3. Ищем в списке ключ и удаляем его.

Время работы:

- В лучшем случае - $O(1)$
- В худшем случае - $O(N)$

Th. Среднее время работы операци поиска, вставки (с проверкой на дубликаты) и удаления в хеш-таблице, реализованной методом цепочек – $O(1+\alpha)$, где α – коэффициент заполнения таблицы.

Доказательство:

Среднее время работы - математическое ожидание времени работы в зависимости от исходного ключа.

Время работы для обработки одного ключа $T(k)$ зависит от длины цепочки и равно $1+N_{h(k)}$, где N_i - длина i-й цепочки. Предполагаем, что хеш-функция равномерна, а ключи равновероятны.

Среднее время работы:

$$T(M,N) = M(T(k)) = \sum_{i=0}^{M-1} \frac{1}{M}(1+N_i) = \frac{1}{M}\sum_{i=0}^{M-1}(1+N_i) = \frac{M+N}{M} = 1+\alpha$$

Метод прямой адресации (закрытое хеширование).

Все ключи хранятся в массиве. Каждая запись – ключ или NIL. При поиске проверяем ячейки в некоторой последовательности.

Добавление ключа

1. Если таблица заполненна, выполняем ReHash - увеличение размера таблицы
2. Вычисляем значение хеш-функции ключа – h.
3. Систематически проверяем ячейки, начиная от A[h], до тех пор, пока не находим пустую ячейку.
4. Помещаем вставляемый ключ в найденную ячейку

Def. Последовательность, в которой вы полняется поиск пустой ячейки в пункте 3, называются последовательностью проб

Note. Важно, чтобы для каждого ключа k последовательность проб представляла собой перестановку множества $0, 1, \ldots, M-1$.

Удаление ключа

1. Вычисляем значение хеш-функции ключа – h.
2. Систематически проверяем ячейки, начиная от A[h], до тех пор, пока не находим заполненную ячейку с совпадающим ключем.
3. Помечаем ячейку флагом Deleted

Note. Метод HasKey продолжает поиск при обнаружении "Deleted а метод вставки считает Deleted пустой ячейкой.

Note. Лучше всего использовать три флага - Busy - заполненная ячейка, Deleted - значение из ячейки было удалено, Free - ячейка никогда не была занята.

Линейное пробирование. Проблема кластеризации.

Линейное пробирование.

$$h(k,i) = (hash(k) + i) mod M$$

Основная проблема при таком подходе - кластеризация.
Последовательность подряд идущих занятых элементов таблицы быстро увеличивается, образуя кластер.
Попадание в элемент кластера при добавлении гарантирует «одинаковую прогулку» для различных ключей и проб. Новый элемент будет добавлен в конец кластера, увеличивая его.
Если $h(k_1,i) = h(k_2,j)$, то $\forall r \;\; h(k_1,i+r) = h(k_2,j+r)$

Квадратичное пробирование.

$$h(k,i) = (hash(k) + c_1 i + c_2 i^2) mod M$$

Требуется, чтобы последовательность проб содержала все индексы $0,1,\ldots,M-1$. Для этого нужно подобрать c_1 и c_2.
При $c_1 = c_2 = \frac{1}{2}$, то проба вычисляется рекуррентно:

$$h(k,i+1) = h(k,i) + i + 1 (mod M)$$

Возникает вторичная кластеризация. Проявляется на ключах с одинаковым хеш-значением h'.
Если $h(k_1,0) = h(k_2,0)$, то $\forall i \;\; h(k_1,i) = h(k_2,i)$
Утверждение. Если $c_1 = c_2 = \frac{1}{2}$, а $M = 2^p$, то квадроатичное пробирование дает перестановку $0,1,\ldots,M-1$.

Доказательство:

От противного. Пусть существуют i и j, $0 \leqslant i,j \leqslant M-1$, для которых

$$\frac{i(i+1)}{2} \equiv \frac{j(j+1)}{2} (mod 2^p)$$

Тогда $i^2 + i - j^2 - j = 2^{p+1} D$, $(i-j)(i+j+1) = 2^{p+1} D$
Если i и j одинаковой четности, то $i+j+1$ нечетно, но $i-j$ не может делиться на 2^{p+1}.

Если i и j разной четности, то $i - j$ нечетно, но $i + j + 1$ не может делиться на 2^{p+1}, т.к. $0 < i + j + 1 < 2^{p+1}$?! ■ .

Двойное хеширование.

$$h(h_1(k) + ih_2(k))modM$$

Требуется, чтобы последовательность проб содержала все индексы $0, 1, \ldots, M - 1$. Для этого все значения $h_2(k)$ должны быть взаимно простыми с M

- M может быть степенью двойки, а $h_2(k)$ всегда возвращать нечетные числа
- M простое, а $h_2(M)$ меньше M.

Th. Математическое ожидание количества проб при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $\alpha = \frac{n}{m} < 1$ в предположении равномерного хеширования не превышает $\frac{1}{1-\alpha}$

- Плюсы
  - Основное преимущество метода открытой адресации – не тратится память на хранение указателей списка.
  - Нет элементов, хранящихся вне таблицы.
- Минусы
  - Хеш-таблица может оказаться заполненной. Коэффициент заполнения α не может быть больше 1.
  - При приближении коэффициента заполнения α к 1 среднее время работы поиска, добавления и удаления стремится к N.

Тема 5. Жадные ал-мы, д.п., персистентные С. Д., р-е выражения и мастер-теорема.

Общая идея жадных алгоритмов.

Жадный алгоритм (англ. Greedy algorithm) – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Доказательство оптимальности часто следует такой схеме:

1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
3. Рассуждение завершается по индукции.

Задача о рюкзаке.

Задача о рюкзаке (англ. Knapsack problem) — одна из NP-полных задач комбинаторной оптимизации. Название своё получила от максимизационной задачи укладки как можно большего числа нужных вещей в рюкзак при условии, что общий объём (или вес) всех предметов, способных поместиться в рюкзак, ограничен. Имеется N грузов. Для каждого i-го груза определён вес w_i и ценность c_i. Нужно упаковать в рюкзак ограниченной грузоподъёмности G те грузы, при которых суммарная ценность упакованного была бы максимальной.

Жадный алгоритм

Предметы сортируются по убыванию стоимости единицы каждого (по отношению цены к весу).

1. Помещаем в рюкзак первый предмет из отсортированного массива, который поместится в рюкзак.
2. Помещаем в рюкзак первый из оставшихся предметов отсортированного массива, который поместится в рюкзак. И т.д., пока в рюкзаке остается место или все оставшиеся предметы оказались тяжелее.

Пример, когда жадный алгоритм не работает

Пусть вместимость рюкзака 90. Предметы уже отсортированны. Применяем к ним жадный алгоритм

номер	вес	цена	цена/вес
1	20	60	3
2	30	90	3
3	50	100	2

Кладём в рюкзак первый, а за ним второй предметы. Третий предмет в рюкзак не влезет. Суммарная ценность поместившегося равна 150. Если бы были взяты второй и третий предметы, то суммарная ценность составила бы 190. Видно, что жадный алгоритм не обеспечивает оптимального решения, поэтому относится к приближенным.

С помощью Динамического программирования можно решить задачу о рюкзаке, если:

- В нашем распоряжении неограниченное количество предметов каждого типа
- Веса целочисленные или дискретные

Пусть все веса целочисленные. Определим D(k) - максимальная стоимость рюкзака с грузами общим весом $\leqslant k$
Тогда D(k+1) можно вычислить, пытаясь положить в рюкзак каждый предмет с весом w_i, используя $D(k+1-w_i)$

$$D(k+1) = max(D(k+1-w_i) + c_i)$$

Ответом в задаче является $D(G)$.
Каждое вычисление $D(k+1)$ выполняется за $O(N)$, где N - количество типов грузов.
Общее время работы $= O(GN)$

Общая идея последовательного вычисления зависимых величин. Идея введения подзадач (декомпозиции) для решения поставленной задачи. Восходящее ДП. Нисходящее ДП, кэширование результатов.

Def. Динамическое программирование (ДП) – способ решения сложных задач путём разбиения их на более простые подзадачи. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

Основные принципы ДП:

- Разбить задачу на подзадачи
- Кэшировать результаты решения подзадач
- Удалять более неиспользуемые результаты решения подзадач(опционально)

Есть два подхода динамического программирования:

- Нисходящее динамическое программирование: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений часто встречающихся подзадач.
- Восходящее динамическое программирование: все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи.

Note. Второй способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем. На это есть следующий пример

Пример: $F(\frac{n}{2}) + F(\frac{2n}{3}), F(0) = F(1) = 1$, деление целочисленное

Вычисление чисел Фибоначчи.

Задача. Есть рекуррента $F_n = F_{n-1} + F_{n-2}, F_1 = F_0 = 1$

1)Нисходящая динамика

```
int GetNthFibonacciNumber(int n) {
  if (n == 0 || n == 1) {
      return 1;
  }
  return GetNthFibNumber(n-1) + GetNthFibNumber(n-2);
}
```

2)Восходящая динамика

```
int GetNthFibonacciNumber(int n) {
  if (n == 0) {
    return 1;
  }
  int previous_number = 1;
  int current_number = 1;
  for (int i = 2; i <= n; ++i) {
    int temp = current_number;
    current_number += previous_number;
    previous_number = temp;
  }
    return current_number;
}
```

Нахождение наибольшей возрастающей подпоследовательности за O(N2) и за O(N log N).

За $O(n^2)$

Построим массив d, где $d[i]$ это длина наибольшей возрастающей подпоследовательности, оканчивающейся в элементе, с индексом i. Массив будем заполнять постепенно сначала $d[0]$, потом $d[1]$ и т.д. Ответом на нашу задачу будет максимум из всех элементов массива d.

Заполнение массива будет следующим: если $d[i] = 1$, то искомая последовательность состоит только из числа $a[i]$. Если $d[i] > 1$, то перед числом $a[i]$ в подпоследовательности стоит какое-то другое число. Переберем его: это может быть любой элемент $a[j](j = 0 \dots i-1)$, но такой, что $a[j] < a[i]$. Пусть на каком-то шаге нам надо посчитать очередное $d[i]$. Все элементы массива d до него уже посчитаны. Значит наше $d[i]$ мы можем посчитать следующим образом: $d[i] = 1 + \max\limits_{j=0\dots i-1} d[j]$ при условии, что $a[j] < a[i]$.

Для восстановления ответа найдем такой элемент последовательности a_i, что $d[i]$ будет максимальным. Это будет последний элемент наибольшей возрастающей подпоследовательности.

Теперь найдем предыдущий элемент. Это такой элемент a_j, что $j < i$ и $d[j] = d[i] - 1$. Будем повторять поиск предыдущего элемента до тех пор, пока не дойдем до такого j, что $d[j] = 1$. Это будет первым элементов наибольшей возрастающей подпоследовательности.

За $O(nlog(n))$

Для более быстрого решения данной задачи построим следующую динамику: пусть $d[i](i = 0...n)$ число, на которое оканчивается возрастающая последовательность длины i, а если таких чисел несколько то наименьшее из них. Изначально мы предполагаем, что $d[0] = -\infty$, а все остальные элементы $d[i] = \infty$.

Заметим два важных свойства этой динамики: $d[i-1] \leqslant d[i]$, для всех $i = 1...n$ и каждый элемент $a[i]$ обновляет максимум один элемент $d[j]$. Это означает, что при обработке очередного $a[i]$, мы можем за $O(\log n)$ с помощью двоичного поиска в массиве d найти первое число, которое больше либо равно текущего $a[i]$ и обновить его.

Для восстановления ответа будем поддерживать заполнение двух массивов: pos и $prev$. В $pos[i]$ будем хранить индекс элемента, на который заканчивается оптимальная подпоследовательность длины i, а в $prev[i]$ позицию предыдущего элемента для $a[i]$.

Note. Более подробное объяснение решения

Количество способов разложить число N на слагаемые.

$$P(n,k) = \begin{cases} P(n,k-1) + P(n-k,k), 0 < k \leqslant n \\ 1, n = 0, k = 0 \\ 0,\ n \neq 0, k = 0 \end{cases}$$

Количество способов разложить число N на различные слагаемые.

$$P(n,k) = \begin{cases} P(n,k-1) + P(n-k,k-1), 0 < k \leqslant n \\ 1, n = 0, k = 0 \\ 0,\ n \neq 0, k = 0 \end{cases}$$

Нахождение наибольшей общей подпоследовательности.

Def. Последовательность X является подпоследовательностью Y, если из Y можно удалить несколько элементов так, что получится последовательность X.

Задача. Наибольшая общая подпоследовательность (англ. longest common subsequence, LCS) – задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух).

Решение:

$$f(n_1,n_2) = \begin{cases} 0, n_1 = 0 \vee n_2 = 0 \\ f(n_1-1,n_2-1)+1, s[n_1] = s[n_2] \\ max(f(n_1-1,n_2), f(n_1,n_2-1)), s[n_1] \neq s[n_2] \end{cases}$$

Как восстановить саму подпоследовательность?

		A	B	C	A	B
	0	0	0	0	0	0
D	0	←↑ 0	←↑ 0	←↑ 0	←↑ 0	←↑ 0
C	0	←↑ 0	←↑ 0	↖ 1	← 1	← 1
B	0	←↑ 0	↖ 1	←↑ 1	←↑ 1	↖ 2
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	←↑ 2

Можно хранить в каждой ячейке таблицы «направление» перехода. Переход по диагонали означает, что очередной символ присутствует в обоих последовательностях. Начинаем проход от правого нижнего угла. Идем по стрелкам, на каждый переход по диагонали добавляем символ в начало строящейся подпоследовательности. «Направления» можно не хранить, а вычислять по значениям в таблице.

Методы восстановления ответа в задачах динамического программирования.

Просто запоминаем из каких состояний пришли :)

Расстояние Левенштейна.

Def. Расстояние Левенштейна (также редакторское расстояние или дистанция редактирования) между двумя строками – это минимальное количество следующих операций необходимых для превращения одной строки в другую:

- вставки одного символа
- удаления одного символа
- замены одного символа на другой

Задача. Вычислить расстояние Левенштейна

Решение:

$D(i, j) =$ количество операций, необходимых для приведения одной подстроки $S[0 \dots i]$ к другой $S[0 \dots j]$

Краевые значения:

- $D(i, 0) = i \; \forall \; i \in 0, \dots, |s| - 1$
- $D(0, j) = j \; \forall \; j \in 0, \dots, |T| - 1$

$$D(i,j) = min \begin{cases} D(i-1,j-1) + m(S[i],T[i]) \\ 1 + D(i-1,j) \\ D(i,j-1) + 1 \end{cases}$$

Где $m(S[i],T[i]) = 0$, если $S[i] = T[j]$ и 1 иначе.

Первое выражение соответствует замене i-го символа первой строки на j-ый символ второй строки.

Второе выражение соответствует удалению i-го символа первой строки и получению из $S[0 \dots i-1]$ строки $T[0 \dots j]$.

Третье выражение соответствует получению из строки $S[0 \dots i]$ строки $T[0 \dots j-1]$ и добавлению $T[j]$.

Персистентные структуры данных. Версии, возможность модифицировать любую версию. Примеры использования.

Def. Персистентные структуры данных (англ. persistent data structure) — это структуры данных, которые при внесении в них каких-то изменений сохраняют все свои предыдущие состояния и доступ к этим состояниям.

- В частично персистентных структурах данных к каждой версии можно делать запросы, но изменять можно только последнюю версию структуры данных.
- В полностью персистентных структурах данных можно менять не только последнюю, но и любую версию структур данных, также к любой версии можно делать запросы.

Пример использования:

Персистентные структуры данных используются при решении геометрических задач. Примером может служить Point location problem — задача о местоположении точки. Задачи такого рода решаются в offline и online. В offline-задачах все запросы даны заранее и можно обрабатывать их одновременно. В online-задачах следующий запрос можно узнать только после того, как найден ответ на предыдущий.

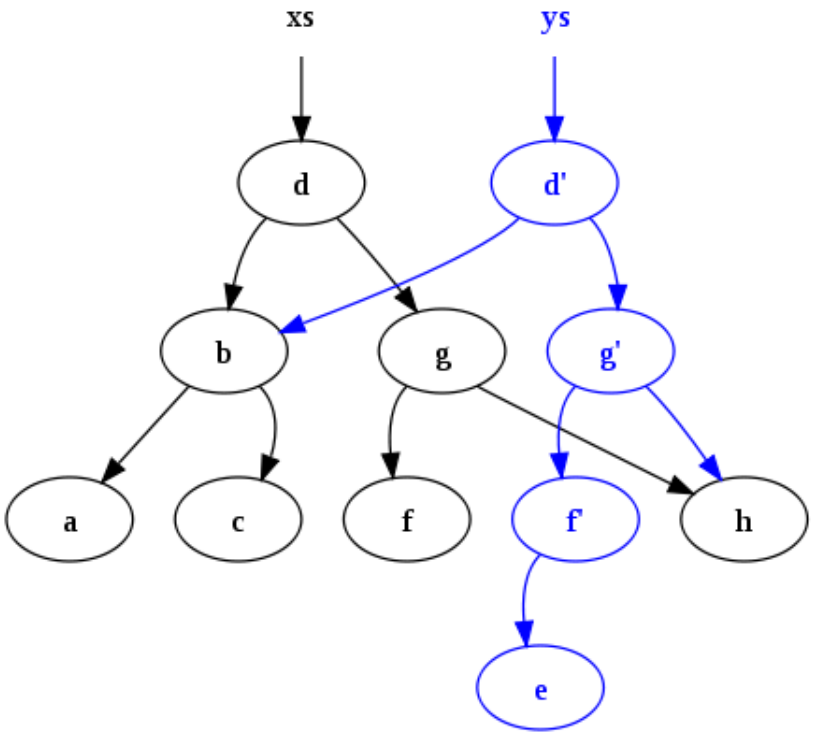
При решении offline-задачи данный планарный граф разбивается на полосы вертикальными прямыми, проходящими через вершины многоугольников. Затем в этих полосах при помощи техники заметающей прямой (scane line) ищется местоположение точки-запроса. При переходе из одной полосы в другую изменяется сбалансированное дерево поиска. Если использовать частично персистентную структуру данных, то для каждой полосы будет своя версия дерева и сохранится возможность делать к ней запросы. Тогда Point location problem может быть решена в online.

Персистентный стек.

Note. Замечательная статья на хабре. Копипастить ее сюда не вижу смысла

Персистентное дерево поиска. Добавление, удаление узла. Повороты.

Пусть есть сбалансированное дерево поиска. Все операции в нем делаются за $O(h)$, где h — высота дерева, а высота дерева $O\ (\log n)$, где n — количество вершин. Пусть необходимо сделать какое-то обновление в этом сбалансированном дереве, например, добавить очередной элемент, но при этом нужно не потерять старое дерево. Возьмем узел, в который нужно добавить нового ребенка. Вместо того чтобы добавлять нового ребенка, скопируем этот узел, к копии добавим нового ребенка, также скопируем все узлы вплоть до корня, из которых достижим первый скопированный узел вместе со всеми указателями. Все вершины, из которых измененный узел не достижим, мы не трогаем. Количество новых узлов всегда будет порядка логарифма. В результате имеем доступ к обоим версиям дерева.



Так как рассматривается сбалансированное дерево поиска, то поднимая вершину вверх при балансировке, нужно делать копии всех вершин, участвующих во вращениях, у которых изменились ссылки на детей. Таких всегда не более трех, поэтому ассимптотика $O\ (\log n)$ не пострадает. Когда балансировка закончится, нужно дойти вверх до корня, делая копии вершин на пути.

Рекуррентные выражения. Способы доказательства оценок: метод подстановки и метод разворачивания суммы.

Оценивая время работы рекурсивной процедуры, мы часто приходим к соотношению, которое оценивает это время через время работы той же процедуры на входных данных меньшего размера. Такого рода соотношения называются рекуррентными.

Пример: Время работы MergeSort

$$T(n) \begin{cases} \Theta(1), \text{ если } n = 1 \\ 2T(n/2) + \Theta(n), \text{ если } n > 1 \end{cases}$$

Способы доказательства

- Метод подстановки. Можно угадать оценку, а затем доказать ее по индукции
- Метод итераций. Можно "развернуть"рекуррентную формулу, получив при этом сумму, которую затем оценить
- Основная теорема о рекуррентных соотношениях

Метод подстановки

Пример: $T(n) = 2T(\left[\frac{n}{2}\right]) + n$

Решение:

Предположим, что $T(n) = O(nlog(n))$, то есть $T(n) \leqslant cnlog(n)$. Тогда

$$T(n) \leqslant 2(c\left[\frac{n}{2}\right] log(\left[\frac{n}{2}\right])) + n \leqslant cnlog(\frac{n}{2}) + n = cnlog(n) - cn + n \leqslant cnlog(n)$$

Метод итераций(суммирование)

Пример: $T(n) = 3T(n/4) + n$

Решение:

$T(n) = 3T(n/4) + n = n + 3n/4 + 9n/16 + 27T(n/64) = \ldots$
Количество итераций $= \log_4 n$. Итого,
$T(n) \leqslant n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \leqslant n\sum\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) = 4n + O(n) = O(n)$

Мастер-теорема(без доказательства)

Th. Пусть $\alpha \geqslant 1$ и $b > 1$ - константы, $f(n)$ - функция, $T(n)$ определено при неотрицательных n формулой

$$T(n) = \alpha T(n/b) + f(n)$$

где под n/b понимается либо $\lceil n/b \rceil$, либо $\lfloor n/b \rfloor$
Тогда:

1. Если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$
2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b \alpha} \log n)$

3. Если $f(n) = \Omega(n^{\log_b \alpha+\varepsilon})$ для некоторого $\varepsilon > 0$ и если $\alpha f(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$.

Доказательство:

Доказательство смотрите в книге Кормена