**1. Нарисуйте дерево, соответствующее префиксному выражению \*a+b\*c+de.**

Префиксная (польская) запись читается слева направо, при этом каждый оператор применим к двум последующим элементам (если это бинарный оператор).

Разберём выражение:

\* a + b \* c + d e

Разбор по уровням:

* \* — корень дерева, левый операнд: a, правый — всё остальное: + b \* c + d e
* + — вторая операция, левая часть: b, правая — \* c + d e
* \* — третья операция, левая часть: c, правая — + d e
* + — последняя операция, с операндами: d и e

Итоговое дерево:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | \* |  |  |  |  |  |  |  |  |
|  | / |  | \ |  |  |  |  |  |  |  |
| a |  |  |  | + |  |  |  |  |  |  |
|  |  |  | / |  | \ |  |  |  |  |  |
|  |  | b |  |  |  | \* |  |  |  |  |
|  |  |  |  |  | / |  | \ |  |  |  |
|  |  |  |  | c |  |  |  | + |  |  |
|  |  |  |  |  |  |  | / |  | \ |  |
|  |  |  |  |  |  | d |  |  |  | e |

**2. Преобразуйте выражение ((a+b)+c\*(d+e)+f)\*(g+h) в префиксную форму**

Разберём выражение по приоритетам:

((a + b) + c \* (d + e) + f) \* (g + h)

1. Вложенные действия:
   * a + b
   * d + e
   * c \* (d + e)
   * (a + b) + (c \* (d + e))
   * ((a + b) + c \* (d + e)) + f
   * вся левая часть: (((a + b) + c \* (d + e)) + f)
   * правая часть: (g + h)
   * всё вместе: (((a + b) + c \* (d + e)) + f) \* (g + h)

Теперь переводим в префиксную форму, начиная с самых вложенных:

a + b → + a b

d + e → + d e

c \* (d + e) → \* c + d e

(a + b) + (c \* (d + e)) → + + a b \* c + d e

+ + a b \* c + d e + f — левая часть полностью

g + h → + g h

Всё выражение:

**\* + + + a b \* c + d e f + g h**

**3. Графы. Обходы графов.**

**Граф** — это структура данных, состоящая из **вершин** (узлов) и **рёбер** (связей между вершинами). Графы могут быть:

* **Ориентированными** (направленные рёбра);
* **Неориентированными** (рёбра без направления);
* **Взвешенными** (каждое ребро имеет вес);
* **Невзвешенными** (веса отсутствуют).

📌 Обходы графа

**Обход** графа — это систематическое посещение всех его вершин, начиная с заданной.

🔹 Обход в глубину (DFS — Depth-First Search)

* Работает по принципу **“в глубину”**: от текущей вершины идёт как можно дальше, пока возможно, затем возвращается назад.
* Использует **рекурсию** или **стек**.
* Применяется для: поиска путей, проверки связности, топологической сортировки, поиска компонент связности.

**Принцип работы:**

1. Посетить вершину.
2. Перейти к первой непосещённой соседней вершине.
3. Повторить шаг 1.
4. Если нет непосещённых — вернуться назад.

🔹 Обход в ширину (BFS — Breadth-First Search)

* Работает по принципу **“в ширину”**: сначала обходятся все вершины, смежные с начальной, затем их соседи и т.д.
* Использует **очередь**.
* Применяется для: поиска кратчайшего пути в невзвешенных графах, проверки двудольности, поиска уровней.

**Принцип работы:**

Поместить стартовую вершину в очередь.

Пока очередь не пуста:

Извлечь вершину.

Добавить в очередь всех её непосещённых соседей.

**4. Графы. Способы реализации.**

Существуют три основных способа представления графов в памяти компьютера:

🔹 1. Матрица смежности

* Представляется в виде квадратной матрицы N x N, где N — число вершин.
* A[i][j] = 1, если существует ребро из i в j, иначе 0.

**Преимущества:**

* Быстрый доступ к информации о наличии ребра O(1).

**Недостатки:**

* Занимает много памяти: O(n²), даже если граф разреженный.

**Применение:** удобно для плотных графов или в задачах с частыми запросами на наличие ребра.

🔹 2. Список смежности

* Каждая вершина хранит **список всех соседей**.
* Например: A: [B, C] означает, что из вершины A есть рёбра в B и C.

**Преимущества:**

* Эффективно по памяти: O(n + m), где n — вершины, m — рёбра.
* Удобно для итерации по соседям.

**Недостатки:**

* Проверка наличия ребра O(k), где k — число соседей.

**Применение:** лучший выбор для разреженных графов.

🔹 3. Список рёбер

* Просто хранятся пары вершин: [(A, B), (B, C), (C, D)].

**Преимущества:**

* Простая структура, легко перебирать все рёбра.

**Недостатки:**

* Неэффективен для частых запросов соседей или проверки смежности.

**Применение:** удобно при алгоритмах, которые работают только с рёбрами (например, Крускала для поиска остовного дерева).

**5. Графы. Построение минимального остовного дерева.**

**Остовное дерево (spanning tree)** — это подграф связного неориентированного графа, который содержит **все вершины исходного графа**, но **минимальное количество рёбер**, не образующих циклов. Если рёбра имеют веса, то целью становится построение **минимального остовного дерева (МОД)** — остовного дерева с наименьшей возможной суммой весов рёбер.

🔹 Алгоритм Крускала (Kruskal’s Algorithm)

* Основан на **жадной стратегии**.
* Работает с **множеством рёбер**, постепенно добавляя наименьшие по весу рёбра, **не создавая циклов**.
* Использует структуру **система непересекающихся множеств (DSU)** для проверки циклов.

**Шаги:**

1. Отсортировать все рёбра по весу.
2. Перебирая рёбра в порядке возрастания:
   * если добавление ребра **не создаёт цикл**, включить его в остов.
3. Повторять, пока не будет n−1 рёбер (где n — число вершин).

**Характеристики:**

* Подходит для **разреженных графов**.
* Сложность: O(m log m) (где m — число рёбер).

🔹 Алгоритм Прима (Prim’s Algorithm)

* Также **жадный алгоритм**, но строит дерево от **одной стартовой вершины**, постепенно расширяя его.
* Всегда добавляет **наименьшее по весу ребро**, которое соединяет вершину внутри дерева с вершиной вне дерева.

**Шаги:**

1. Выбрать любую стартовую вершину.
2. Пока не включены все вершины:
   * выбрать минимальное ребро от дерева к новой вершине;
   * добавить вершину и ребро в остов.

**Характеристики:**

* Эффективен для **плотных графов**.
* Сложность с очередью приоритетов: O(m log n).

📌 Применение МОД:

Построение оптимальной сети (дорог, кабелей, трубопровода);

Минимизация затрат на соединение всех узлов;

Коммуникационные и распределённые системы.

**6. Графы. Кратчайшие расстояния**

Задача: найти путь **с минимальной суммой весов** между двумя вершинами (или от одной ко всем).

🔹 Алгоритм Дейкстры (Dijkstra)

* Работает для графов с **неотрицательными весами рёбер**.
* Использует **приоритетную очередь** (обычно min-heap) для выбора вершины с минимальным текущим расстоянием.

**Шаги:**

1. Установить расстояние до стартовой вершины как 0, до остальных — ∞.
2. Пока есть непосещённые вершины:
   * выбрать вершину с минимальным расстоянием;
   * обновить расстояния до её соседей, если через неё путь короче.

**Сложность:** O(m log n) при использовании очереди с кучей.

🔹 Алгоритм Беллмана-Форда

* Подходит для графов с **отрицательными весами** рёбер.
* Может определить **наличие отрицательных циклов**.

**Шаги:**

1. Установить расстояния аналогично Дейкстре.
2. Повторить n−1 раз (где n — количество вершин):
   * для каждого ребра (u, v) обновить расстояние v = min(v, u + вес)

**Сложность:** O(n \* m)

🔹 Алгоритм Флойда–Уоршелла (Floyd–Warshall)

* Используется для **поиска кратчайших путей между всеми парами вершин**.
* Работает на основе динамического программирования.

**Шаги:**

1. Создать матрицу расстояний D[i][j] с начальными значениями весов рёбер (или ∞).
2. Для каждой вершины k:
   * для каждой пары (i, j):
     + D[i][j] = min(D[i][j], D[i][k] + D[k][j])

**Сложность:** O(n³)

**7. Поиск в линейных структурах.**

**Линейные структуры данных** — это структуры, в которых элементы организованы в последовательность. К ним относятся массивы, списки, стеки, очереди.

**🔹 Последовательный (линейный) поиск**

**Суть**: перебор всех элементов по очереди до тех пор, пока не будет найден нужный или не закончится структура.

Применяется к неотсортированным структурам.

Время поиска — **O(n)**.

**Пример использования:**

Поиск фамилии студента в журнале;

Поиск заданного ключа в неотсортированном списке пользователей.

**Недостатки:**

Низкая эффективность при большом объёме данных.

**🔹 Бинарный поиск**

**Суть**: работает **только с отсортированными структурами**. На каждом шаге делит массив пополам и выбирает, в какой половине искать дальше.

Время поиска — **O(log n)**.

**Пример использования:**

Поиск слова в алфавитно отсортированном словаре;

Поиск имени в базе данных с упорядоченным индексом.

**Условия применения:**

Данные должны быть **отсортированы**;

Эффективен при больших объёмах.

**🔹Поиск Фибоначчи (Fibonacci Search)**

**Суть**: это алгоритм **поиска в отсортированных структурах**, аналогичный бинарному поиску, но использующий **последовательность Фибоначчи** для определения точек деления массива.

В отличие от бинарного поиска, который делит массив **на равные половины**, поиск Фибоначчи делит массив **по "золотому сечению"**, основанному на числах Фибоначчи:

Fn = Fn-1 + Fn-2

Он особенно эффективен в **структурах, где доступ к элементам стоит дорого**, например, в **ленивых структурах или при работе с памятью, чувствительной к кэшированию**.

📌 Как работает:

1. Найти наименьшее число Фибоначчи F[k], которое больше или равно длине массива.
2. Использовать F[k-1] как точку деления.
3. Сравнить элемент в этой позиции с искомым:  
     
   * если элемент меньше — перемещаем границу вправо;
   * если больше — влево;
   * если равен — найдено.
4. Повторять, уменьшая индекс Фибоначчи.

⏱ Сложность:

* В худшем случае — **O(log n)**, как и бинарный поиск;
* Однако требует **меньше вычислений над индексами**, и в некоторых случаях может быть быстрее при неблагоприятном распределении данных.

✅ Примеры использования:

* Поиск в **массиве, размещённом на медленных устройствах** (магнитная лента, HDD);
* Поиск в **структурах с ограниченным доступом к данным по индексу**;
* Алгоритмическая альтернатива бинарному поиску при специфических условиях доступа или оптимизации под кэш.

📌 Сравнение подходов:

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгоритм** | **Данные отсортированы?** | **Сложность** | **Применение** |
| Последовательный | Нет | O(n) | Поиск в любых структурах |
| Бинарный | Да | O(log n) | Быстрый поиск в массиве |
| Фибоначчи | Да | O(log n) | Альтернатива бинарному, оптимизация кэш-доступа |

**8. АТД - стек. Реализация с помощью указателей.**

**Стек** — это **абстрактный тип данных**, работающий по принципу **LIFO (Last In — First Out)**: последний добавленный элемент удаляется первым.

🔹 Операции стека:

* push(x) — добавить элемент на вершину;
* pop() — удалить верхний элемент;
* top() / peek() — вернуть верхний элемент, не удаляя;
* isEmpty() — проверка на пустоту.

🔹 Реализация стека через указатели (на связном списке)

**Принцип**: каждый элемент (узел) содержит значение и **указатель на следующий элемент**.

top → [ значение | указатель ] → [ значение | указатель ] → null

* push: создаём новый узел, указываем на текущий top, затем обновляем top;
* pop: удаляем узел, на который указывает top, и переходим к следующему.

**Преимущества реализации через указатели:**

* Неограниченный размер (в отличие от массива);
* Не требует перераспределения памяти.

🔹 Примеры использования стека

* Обратная польская запись (вычисление выражений);
* Проверка скобочной структуры в коде;
* Реализация возврата назад (например, в браузере);
* Рекурсивные алгоритмы (реализация системного стека вызовов).

**9. АТД - список. Реализация с помощью указателей.**

**Список** — это последовательность элементов, каждый из которых **может быть вставлен, удалён или доступен по ссылке**.

🔹 Виды списков:

* **Односвязный список** — каждый узел содержит значение и указатель на следующий;
* **Двусвязный список** — каждый узел содержит указатель на следующий и предыдущий;
* **Циклический список** — последний элемент указывает на первый.

🔹 Реализация списка через указатели

**Узел списка (Node):**

[ значение | next ]       (односвязный)

[ prev | значение | next ] (двусвязный)

* Для **вставки** — создаём новый узел, меняем указатели;
* Для **удаления** — перенастраиваем указатели так, чтобы "пропустить" нужный узел.

**Пример операций:**

* Вставка в начало: O(1)
* Вставка в конец (если есть указатель): O(1)
* Поиск элемента: O(n)

🔹 Примеры использования списка

Реализация очередей, стеков, деков;

Списки задач в ОС;

Управление динамической памятью (свободные блоки);

Буферы воспроизведения и потока данных.

**10. Нарисуйте все возможные деревья двоичного поиска для элементов 1, 2, 3, 4.**

**Определение**: Дерево двоичного поиска (BST — Binary Search Tree) — бинарное дерево, где:

* все значения **в левом поддереве < корня**,
* все значения **в правом поддереве > корня**.

Нам нужно **перечислить все уникальные BST**, которые можно построить из чисел {1, 2, 3, 4}.

Это задача комбинаторная. Число различных BST для n различных элементов равно **числу Каталана**:

mathematica

КопироватьРедактировать

C₄ = 14

Значит, существует **14 уникальных BST** для 4-х элементов.

Изобразим их в текстовом виде (корень → поддеревья):

Примеры 5 из 14 (полный список слишком объёмный для текстового представления):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |
|  | \ |  |  |  |  |  |
|  |  | 2 |  |  |  |  |
|  |  |  | \ |  |  |  |
|  |  |  |  | 3 |  |  |
|  |  |  |  |  | \ |  |
|  |  |  |  |  |  | 4 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 |  |  |  |  |  |
|  |  | \ |  |  |  |  |
|  |  |  | 2 |  |  |  |
|  |  |  |  | \ |  |  |
|  |  |  |  |  | 4 |  |
|  |  |  |  | / |  |  |
|  |  |  | 3 |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 2 |  |  |  |  |
|  | / |  | \ |  |  |  |
| 1 |  |  |  | 3 |  |  |
|  |  |  |  |  | \ |  |
|  |  |  |  |  |  | 4 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | 3 |  |  |
|  |  |  | / |  | \ |  |
|  |  | 2 |  |  |  | 4 |
|  | / |  |  |  |  |  |
| 1 |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 2 |  |  |  |
|  |  | / |  | \ |  |  |
|  | 1 |  |  |  | 4 |  |
|  |  |  |  | / |  |  |
|  |  |  | 3 |  |  |  |

**11. АТД - очередь. Реализация с помощью указателей.**

**Очередь** — это логическая структура данных, которая работает по принципу:

**"Первым вошёл — первым вышел" (FIFO — First In, First Out)**

Это как **очередь в кассу**: кто пришёл раньше — обслуживается раньше.

🔹 Основные операции очереди

1. **enqueue** — добавление элемента **в конец** очереди.
2. **dequeue** — удаление элемента **из начала** очереди.
3. **peek** или **front** — посмотреть, кто сейчас **первый в очереди**, не удаляя.
4. **isEmpty** — проверить, пуста ли очередь.

🔹 Реализация с помощью указателей (на связном списке)

* Очередь можно построить на **связном списке**, где каждый элемент знает, кто следующий за ним.
* Хранятся два указателя:
  + **на начало** (откуда удаляется элемент);
  + **на конец** (куда добавляется элемент).

**Преимущества такой реализации:**

* **Нет ограничения по размеру** (в отличие от массива);
* **Быстрая вставка и удаление** — каждый из этих процессов занимает **постоянное время**;
* **Не требуется сдвиг элементов** в памяти.

🔹 Где используется очередь в жизни и в программировании

* **Очередь заданий (task queue)** — в операционных системах, когда процессы ждут своей очереди;
* **Очередь принтера** — документы печатаются по порядку;
* **Обработка сообщений** — в сетевых приложениях и мессенджерах;
* **Очередь событий** — в браузерах, игровых движках;
* **Буферы** — при передаче данных (например, потоковое видео).

**12. Преобразуйте выражение ((a+b)+c\*(d+e)+f)\*(g+h) в постфиксную форму**

Исходное выражение:

((a + b) + c \* (d + e) + f) \* (g + h)

Разберём приоритет:

1. Скобки:
   * (a + b)
   * (d + e)
   * (g + h)
2. Умножение выше по приоритету, чем сложение.

Постфиксная (Reverse Polish Notation) форма:

1. a + b → a b +
2. d + e → d e +
3. c \* (d + e) → c d e + \*
4. (a + b) + (c \* (d + e)) → a b + c d e + \* +
5. ((a + b) + c \* (d + e)) + f → a b + c d e + \* + f +
6. g + h → g h +

Всё выражение: (((a + b) + c \* (d + e)) + f) \* (g + h) → **a b + c d e + \* + f + g h + \***