# *LEK.#1 -* Các phương pháp lập trình hiện đại:

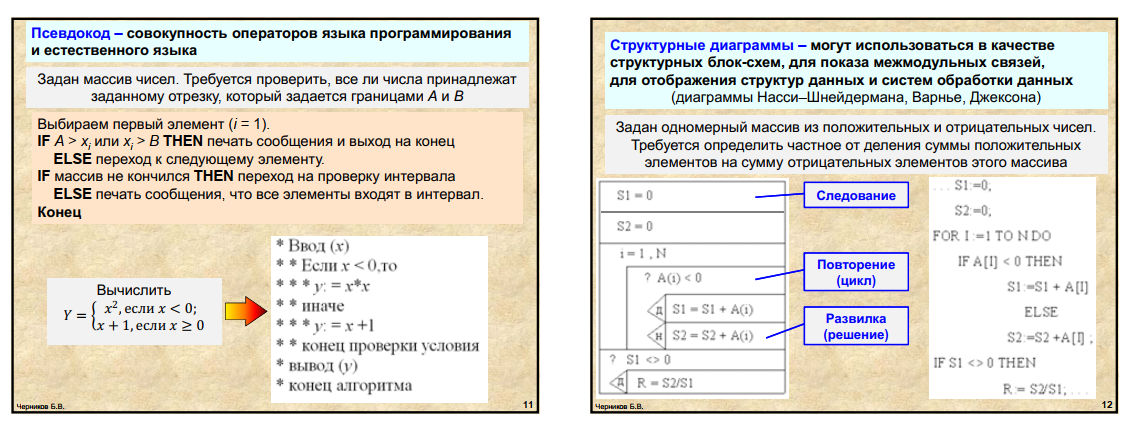
1. Viết thuật toán lập trình.
2. Lựa chọn cấu trúc trình bày dữ liệu.

**Тема 1: Элементы теории алгоритмов**

**(Các yếu tố của lý thuyết thuật toán)**







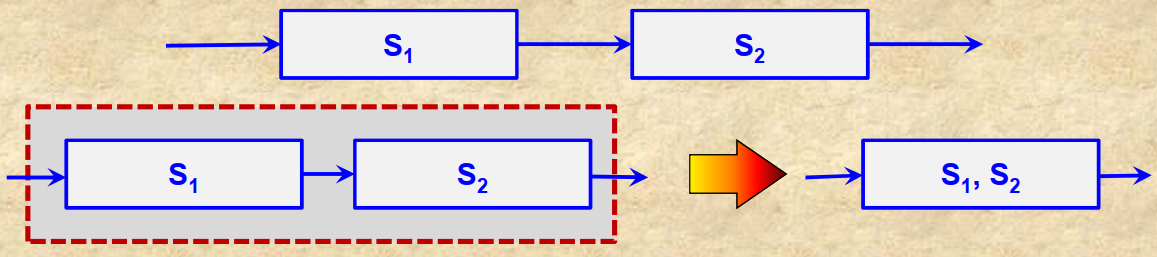
Часть алгоритма, организованная как простое действие, т. е. имеющая один входд (выполнение начинается всегда с одного и того же действия) и один выход (т. е. после завершения данного блока всегда начинает выполняться одно и то же действие), называется функциональным блоком.

Một phần của thuật toán, được tổ chức dưới dạng một hành động đơn giản, nghĩa là có một đầu vào (thực thi luôn bắt đầu bằng cùng một hành động) và một đầu ra (tức là, sau khi hoàn thành khối này, hành động tương tự luôn bắt đầu), được gọi là chức năng khối.

1. Линейный поток управления

Поток управления, в котором выполняются оба свойства, называется линейным – в нем несколько функциональных блоков выполняются последовательно.

Một luồng điều khiển trong đó cả hai thuộc tính được thực thi, gọi là tuyến tính - nó có một số chức năng các khối được thực hiện tuần tự.



1. Ветвление

Phân nhánh

Ветвление организует выполнение одного из двух функциональных блоков в зависимости от значения проверяемого логического условия.

Sự phân nhánh tổ chức thực hiện một trong hai khối chức năng tùy thuộc vào giá trị của điều kiện logic được kiểm tra.

Проверка P выполняется предикатом – функцией, задающей логическое выражение или условие, значением которого может быть истина или ложь.

Kiểm tra P được thực hiện bởi một vị từ - một hàm xác định biểu thức hoặc điều kiện logic có giá trị có thể đúng hoặc sai.

Переключатель организует выбор одного варианта из множества возможных.

Công tắc tổ chức lựa chọn một tùy chọn từ nhiều khả năng.

1. Повторение

Lặp lại

Повторение – многократное выполнение фрагментов алгоритма (программы).

Lặp lại - thực hiện lặp lại các đoạn của một thuật toán (chương trình).

Такой тип потока управления называется циклическим – он организует многократное повторение функционального блока, пока логическое условие его выполнения является истинным.

Loại luồng điều khiển này được gọi là tuần hoàn - nó tổ chức sự lặp lại lặp lại của một khối chức năng, cho đến khi khi điều kiện logic để thực hiện nó là đúng.

Алгоритм называется структурным, если он представляет собой комбинацию трех рассмотренных выше структур (они называются базовыми алгоритмическими структурами).

Một thuật toán được gọi là cấu trúc nếu nó là sự kết hợp của ba cấu trúc được thảo luận ở trên (chúng được gọi là cấu trúc thuật toán cơ bản).

Преимущества структурных алгоритмов

Lợi ích của thuật toán cấu trúc

Понятность и простота восприятия алгоритма (поскольку невелико число исходных структур, которыми он образован)

Sự rõ ràng và đơn giản về nhận thức của thuật toán (vì số lượng cấu trúc ban đầu mà nó được hình thành là nhỏ)

Проверяемость (для проверки любой из основных структур достаточно убедиться в правильности входящих в нее функциональных блоков)

Xác minh (để xác minh bất kỳ cấu trúc chính nào, đủ để xác minh tính chính xác của các khối chức năng có trong nó)

Модифицируемость (состоит в простоте изменения структуры алгоритма, поскольку составляющие блоки относительно независимы)

Tính thay đổi (bao gồm sự đơn giản trong việc thay đổi cấu trúc của thuật toán, vì các khối cấu thành tương đối độc lập)

Структурная теорема Бома – Джакопини

Định lý cấu trúc Bohm - Jacopini

Любой алгоритм может быть сведен к структурному

Иными словами: *“любому неструктурному алгоритму может быть построен эквивалентный ему структурный алгоритм”*

Bất kỳ thuật toán có thể được giảm xuống cấu trúc

Nói cách khác: *“Có thể xây dựng một thuật toán cấu trúc tương đương với bất kỳ thuật toán phi cấu trúc nào”*

# *LEK.#2* **ТЕМА 2 - ОЦЕНКА СТРУКТУРНОЙ И ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМОВ И ПРОГРАММ**

Đánh giá sự phức tạp về cấu trúc và thời gian của các thuật toán và chương trình

## **ВРЕМЕННАЯ СЛОЖНОСТЬ АЛГОРИТМА.**

Độ phức tạp thời gian của thuật toán.

О-символика

Biểu tượng O

**Временная сложность алгоритма зависит от количества входных данных.**

Độ phức tạp thời gian của thuật toán phụ thuộc vào lượng dữ liệu đầu vào

**Обычно говорят, что временная сложность алгоритма имеет порядок T(n) от входных данных размера n.**

Người ta thường nói rằng độ phức tạp thời gian của một thuật toán là theo thứ tự T (n) của dữ liệu đầu vào có kích thước n.

**Точно определить величину T(n) на практике представляется довольно трудно.**

Xác định chính xác giá trị của T (n) trong thực tế có vẻ khá khó khăn.

**Поэтому прибегают к асимптотическим отношениям с использованием O-символики.**

Do đó, họ dùng đến các mối quan hệ tiệm cận bằng cách sử dụng biểu tượng O.

**Если число тактов (действий), необходимое для работы алгоритма, выражается как 11n2 + 19n − log n + 3 n + 4, то это алгоритм, для которого T(n) имеет порядок O(n2)**

Nếu số chu kỳ đồng hồ (hành động) cần thiết để thuật toán hoạt động được biểu thị là 11n2 + 19n - log n + 3 n + 4, thì đây là thuật toán mà T (n) có thứ tự O (n2)

**Фактически, из всех слагаемых оставляется только то, которое вносит наибольший вклад при больших n в этом случае остальными слагаемыми можно пренебречь), и игнорируется коэффициент перед ним**

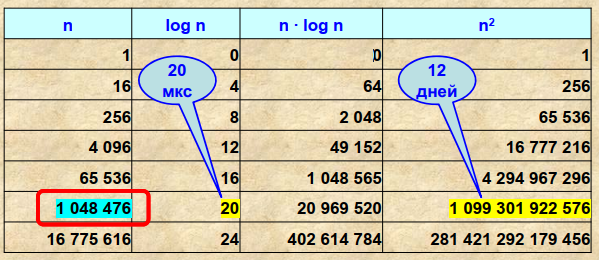
Trong thực tế, trong tất cả các điều khoản, chỉ có một điều khoản đóng góp lớn nhất cho n lớn trong trường hợp này, các điều khoản còn lại có thể bị bỏ qua) và hệ số trước nó bị bỏ qua.

Когда используют обозначение O(), имеют в виду не точное время исполнения, а только его предел сверху, причем с точностью до постоянного множителя

Khi chỉ định O () được sử dụng, chúng không có nghĩa là thời gian thực hiện chính xác, mà chỉ giới hạn trên của nó và tối đa là một yếu tố không đổi.

Когда говорят, например, что алгоритму требуется время порядка O(n2), имеют в виду, что время исполнения задачи растет не быстрее, чем квадрат количества элемент ов.

Ví dụ, khi họ nói rằng một thuật toán yêu cầu thời gian theo thứ tự O (n2), điều đó có nghĩa là thời gian thực hiện của một tác vụ không tăng nhanh hơn bình phương số lượng phần tử.



Если операция выполняется за фиксированное число шагов, не зависящее от количества данных, то принято писать O(1)

Nếu thao tác được thực hiện theo một số bước cố định, không phụ thuộc vào lượng dữ liệu, thì thông thường sẽ viết O (1)

Основание логарифма здесь не пишется

Cơ sở của logarit không được viết ở đây

Пусть есть O(log2n) - Để có O (log2n)

Однако , а , как и любую константу, символ О() не учитывает. Поэтому O(log2n) = O(log3n)

Tuy nhiên, log2n = log3n / log32 và log32, giống như bất kỳ hằng số nào, ký tự O () không được tính. Do đó, O (log2n) = O (log3n)

Время выполнения алгоритма зависит не только от количества входных данных, но и от их значений

Thời gian thực hiện của thuật toán không chỉ phụ thuộc vào lượng dữ liệu đầu vào mà còn phụ thuộc vào giá trị của chúng

Чтобы учитывать этот факт, сохраняя при этом возможность анализировать алгоритмы независимо от данных, различают:

Để tính đến thực tế này trong khi duy trì khả năng phân tích thuật toán bất kể dữ liệu, phân biệt:

* Максимальную сложность Tmax (n) – сложность наиболее неблагоприятного случая, когда алгоритм работает дольше всего

Độ phức tạp tối đa Tmax (n) - độ phức tạp của trường hợp bất lợi nhất, khi thuật toán chạy lâu nhất

* Среднюю сложность Tmid (n) − сложность алгоритма в среднем

Độ phức tạp trung bình Tmid (n) - độ phức tạp thuật toán trung bình

* Минимальную сложность Tmin (n) − сложность в наиболее благоприятном случае, когда алгоритм справляется быстрее всего.

Độ phức tạp tối thiểu Tmin (n) - độ phức tạp trong trường hợp thuận lợi nhất, khi thuật toán xử lý nhanh nhất.

### **Базовые принципы оценки временной сложности алгоритма:**

Các nguyên tắc cơ bản để đánh giá độ phức tạp thời gian của thuật toán:

1. Время выполнения операций присваивания, чтения, записи обычно имеют порядок O(1). Исключение – когда операнды представляют собой массивы или вызовы функций.

Thời gian thực hiện các thao tác gán, đọc, ghi thường là theo thứ tự O (1). Ngoại lệ là khi các toán hạng là mảng hoặc gọi hàm.

1. Время выполнения последовательности операций совпадает с наибольшим временем выполнения операции в ней (правило сумм – если одна операция имеет порядок O(f(n)), а другая − порядок O(g(n)), то общее время будет иметь порядок O(max(f(n), g(n)) ).

Thời gian thực hiện của chuỗi các hoạt động trùng với thời gian thực hiện dài nhất của hoạt động trong đó (quy tắc tổng - nếu một hoạt động có thứ tự O (f (n)) và hoạt động khác là theo thứ tự O (g (n)), thì tổng thời gian sẽ là thứ tự O (tối đa (f (n), g (n))).

1. Время выполнения конструкции ветвления (if−then−else) состоит из времени вычисления логического выражения (обычно имеет порядок O(1) ) и наибольшего из времени, необходимого для выполнения операций, исполняемых при истинном значении логического выражения и при ложном значении логического выражения.

Thời gian thực hiện của cấu trúc phân nhánh (nếu - thì - khác) bao gồm thời gian tính toán của biểu thức logic (thường là theo thứ tự O (1)) và thời gian dài nhất cần thiết để hoàn thành các hoạt động được thực hiện với giá trị đúng của biểu thức logic và giá trị sai của biểu thức logic.

1. Время выполнения цикла состоит из времени вычисления условия прекращения цикла (обычно имеет порядок O(1) ) и произведения количества выполненных итераций цикла на наибольшее возможное время выполнения операций тела цикла.

Thời gian thực hiện của chu trình bao gồm thời gian tính toán điều kiện kết thúc của chu trình (thường là theo thứ tự O (1)) và tích của số lần lặp được thực hiện bởi chu kỳ theo thời gian dài nhất có thể cho các hoạt động của cơ thể.

1. Время выполнения операции вызова процедур определяется как время выполнения вызываемой процедуры

Thời gian thực hiện của một hoạt động gọi thủ tục được định nghĩa là thời gian thực hiện của thủ tục được gọi.

1. При наличии в алгоритме операций безусловного перехода необходимо учитывать изменения последовательности операций, осуществляемых с использованием этих операций безусловного перехода.

Nếu có một hoạt động nhảy vô điều kiện trong thuật toán, cần phải tính đến các thay đổi trong chuỗi các hoạt động được thực hiện bằng cách sử dụng các hoạt động nhảy vô điều kiện này.

## **ПОНЯТИЕ СТРУКТУРНОЙ СЛОЖНОСТИ**



Разнообразие поведения алгоритма (программы) и связей между его входными и результирующими данными определяется набором маршрутов (чередующихся последовательностей вершин и дуг графа управления), по которым он исполняется.

Sự đa dạng về hành vi của thuật toán (chương trình) và mối quan hệ giữa dữ liệu đầu vào và dữ liệu kết quả được xác định bởi tập hợp các tuyến đường (các chuỗi xen kẽ của các đỉnh và cung của biểu đồ điều khiển) dọc theo nó được thực thi.

Сложность программного модуля связана не столько с размером (числом команд) программы, сколько с числом маршрутов ее исполнения и их сложностью.

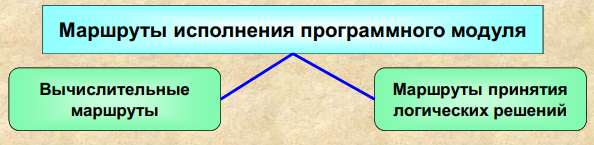
Độ phức tạp của mô-đun phần mềm không liên quan nhiều đến kích thước (số lượng lệnh) của chương trình, nhưng với số lượng tuyến đường để thực hiện và độ phức tạp của chúng.

Маршруты возможной обработки данных определяют сложность разработки программы.

Các tuyến xử lý dữ liệu có thể xác định sự phức tạp của việc phát triển chương trình.

Данную метрику сложности можно использовать для оценки трудоемкости тестирования и сопровождения модуля, а также для оценки потенциальной надежности его функционирования.

Số liệu phức tạp này có thể được sử dụng để đánh giá mức độ phức tạp của kiểm tra và duy trì mô-đun, cũng như để đánh giá độ tin cậy tiềm năng của chức năng của nó.



Сложность вычислительных маршрутов оценивается формулой:

**m** – количество маршрутов исполнения программы.

**l­i** – число данных обрабатываемых в i-ом маршруте.

**νj** – число значений обрабатываемых данных j-го типа (2 ≤ νj ≤ 5).

**Общее число арифметических операций не выходит за пределы 5–10%, поэтому вычислительные маршруты не определяют структурную сложность программ.**

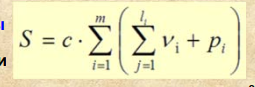
Tổng số phép toán số học không vượt quá 5-10%, do đó, các tuyến tính toán không xác định độ phức tạp về cấu trúc của các chương trình.

**Сложность маршрутов принятия логических решений оценивается формулой:**

**Độ phức tạp của các tuyến ra quyết định hợp lý được ước tính theo công thức:**

**pi** - число ветвлений или число проверяемых условий в i-ом маршруте.

**Общая сложность программы**

****

**с** – коэффициент пропорциональности.

## **КРИТЕРИИ ВЫДЕЛЕНИЯ МАРШРУТОВ**

**Tiêu chí lựa chọn tuyến đường**

Наилучший критерий позволит выделить все реальные маршруты исполнения программы при любых сочетаниях исходных данных.

Tiêu chí tốt nhất sẽ cho phép bạn làm nổi bật tất cả các tuyến thực hiện chương trình thực sự cho bất kỳ sự kết hợp dữ liệu nguồn nào.

1. **Критерий 1**

Граф программы по управлению должен быть покрыт минимальным набором путей, проходящих через каждый оператор ветвления по каждой дуге. Повторная проверка дуг не оценивается и считается избыточной. В процессе проверки гарантируется выполнение всех передач управления между операторами программы и каждого оператора не менее одного раза.

Biểu đồ của chương trình điều khiển phải được bao phủ bằng một tập hợp tối thiểu các đường dẫn đi qua từng toán tử nhánh trên mỗi cung. Kiểm tra lại các cung không được đánh giá và coi là dư thừa. Trong quá trình xác minh, tất cả các lần chuyển kiểm soát giữa các nhà khai thác chương trình và mỗi nhà khai thác được đảm bảo hoàn thành ít nhất một lần.

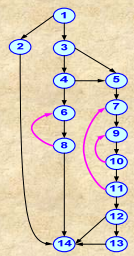
Поток управления – последовательность выполнения различных модулей и операторов программы.

Luồng điều khiển - trình tự thực hiện của các mô-đun và toán tử chương trình khác nhau.

Граф потока управления – ориентированный граф, моделирующий поток управления программы.

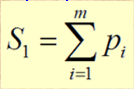
Biểu đồ luồng điều khiển là biểu đồ có hướng mô phỏng luồng điều khiển chương trình.

|  |  |
| --- | --- |
|  | Рассмотрим структурную сложность типичной программы управления, содержащей 100 – 200 команд, граф потока управления которой содержит 14 вершин, 20 дуг и 3 цикла.  Hãy xem xét độ phức tạp về cấu trúc của một chương trình điều khiển điển hình chứa 100 đến 200 lệnh, biểu đồ luồng điều khiển chứa 14 đỉnh, 20 cung và 3 chu kỳ. |

Для полной проверки этой программы по критерию 1 достаточно следующих маршрутов:

Để kiểm tra đầy đủ chương trình này theo tiêu chí 1, các tuyến đường sau là đủ:





Структурная сложность программы по критерию 1

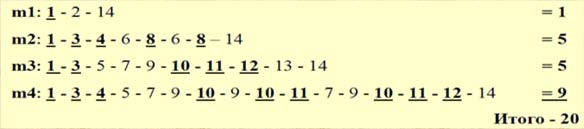
Độ phức tạp về cấu trúc của chương trình theo tiêu chí 1

где pi – количество вершин ветвления в i-том маршруте без учета последней вершины.

Trong đó pi là số đỉnh của nhánh trong tuyến thứ i mà không tính đến đỉnh cuối cùng.

С учетом точек ветвления:

Cho các điểm nhánh:



**Недостаток первого критерия:** не учитывается комбинаторика сочетания условий на разных участках маршрутов (например, при сочетаниях ветвлений в вершинах 3 и 12)

Nhược điểm của tiêu chí thứ nhất: tổ hợp các điều kiện kết hợp trên các phần khác nhau của các tuyến đường không được tính đến (ví dụ: khi kết hợp các nhánh ở đỉnh 3 và 12)

1. **Критерий 2**

Основан на анализе базовых маршрутов в программе, формируемых и оцениваемых на основе цикломатического числа графа потока управления.

Dựa trên phân tích các tuyến cơ bản trong chương trình, được hình thành và đánh giá dựa trên số chu kỳ của biểu đồ luồng điều khiển.

Цикломатическое число исходного графа программы:

Số chu kỳ của biểu đồ nguồn của chương trình:



m – общее число дуг в графе;

m là tổng số cung trong biểu đồ;

n – общее число вершин в графе;

n là tổng số đỉnh trong đồ thị;

p – число связных компонентов графа.

p là số thành phần được kết nối của đồ thị.

**Число связных компонентов графа** p равно количеству дуг, необходимых для превращения исходного графа **в максимально связный (или полносвязный) граф.**

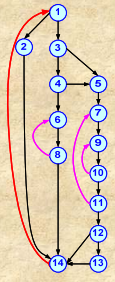
Số lượng các thành phần được kết nối của biểu đồ p bằng số lượng cung cần thiết để biến biểu đồ ban đầu thành biểu đồ được kết nối tối đa (hoặc được kết nối đầy đủ).

**Максимально связным (полносвязным)** называется граф, у которого **любая вершина доступна из любой другой вершины**.

Biểu đồ được kết nối tối đa (kết nối đầy đủ) là một biểu đồ trong đó mọi đỉnh có thể truy cập được từ bất kỳ đỉnh nào khác.

Максимально связный (полносвязный) граф получается из исходного графа путем **замыкания конечных и начальной вершин.**

Một đồ thị được kết nối tối đa (kết nối đầy đủ) được lấy từ biểu đồ ban đầu bằng cách đóng các đỉnh hữu hạn và ban đầu.

Для рассматриваемого графа полносвязный граф получается путем добавления дуги (14, 1)

Đối với biểu đồ đang xem xét, biểu đồ được kết nối đầy đủ có được bằng cách thêm một cung (14, 1)



Критерий 2 (по цикломатическому числу) требует однократной проверки или тестирования **каждого линейно-независимого цикла** и **каждого линейнонезависимого ациклического участка** программы

Tiêu chí 2 (theo số chu kỳ) yêu cầu kiểm tra hoặc kiểm tra từng chu kỳ độc lập tuyến tính và từng phần chu kỳ độc lập tuyến tính của chương trình.

Каждый линейно-независимый ациклический маршрут или цикл **отличается от всех остальных хотя бы одной вершиной или дугой.**

Mỗi tuyến đường hoặc chu kỳ độc lập tuyến tính khác với tất cả các tuyến khác có ít nhất một đỉnh hoặc cung.

При использовании критерия 2 **количество проверяемых маршрутов** равно цикломатическому числу.

Khi sử dụng tiêu chí 2, số lượng tuyến đường được kiểm tra bằng số chu kỳ.

|  |  |
| --- | --- |
| Линейно-независимые **циклические** маршруты  Tuyến đường tuần hoàn độc lập | Линейно-независимые **ациклические** маршруты  Tuyến đường độc lập tuyến tính |
|  |  |

**Определение структурной сложности по второму критерию**

|  |  |
| --- | --- |
|  |  |

Для **правильно структурированных программ** цикломатическое число Z можно определить путем подсчета числа вершин nв, в которых происходит ветвление.

Đối với các chương trình có cấu trúc tốt, số chu kỳ Z có thể được xác định bằng cách đếm số lượng nút nв tại đó phân nhánh diễn ra.



*В нашем примере* ***Z = 7 + 1 = 8***

*(ветвления имеют место в вершинах графа* ***1, 3, 4, 8, 10, 11, 12****)*

Исследования графов реальных программных модулей **с достаточно большим фиксированным числом вершин** показали:

Các nghiên cứu về đồ thị của các mô-đun phần mềm thực với số lượng đỉnh cố định đủ lớn cho thấy:

* Суммарная сложность тестов почти не зависит от детальной структуры графа и в основном определяется числом предикатов – ветвлений графа.

Tổng độ phức tạp của các phép thử gần như không phụ thuộc vào cấu trúc chi tiết của đồ thị và chủ yếu được xác định bởi số lượng vị từ - nhánh đồ thị.

* При неизменном числе вершин **в широких графах** имеется большее количество маршрутов, чем в узких графах, но маршруты в среднем короткие. **В узких графах** число маршрутов сокращается по сравнению с широкими, но маршруты становятся длиннее.

Với cùng số đỉnh, đồ thị rộng có nhiều tuyến hơn đồ thị hẹp, nhưng các tuyến trung bình ngắn. Trong các biểu đồ hẹp, số lượng tuyến đường được giảm so với các tuyến đường rộng, nhưng các tuyến đường trở nên dài hơn.

В результате величина S2 при изменении структуры графов изменяется меньше, чем цикломатическое число и сильнее коррелирована с числом вершин.

Do đó, giá trị S2 khi thay đổi cấu trúc của đồ thị thay đổi ít hơn số chu kỳ và tương quan nhiều hơn với số đỉnh.

В.В. Липаев показал:

* Для Z ≤ 10 модули корректно проверяемы и число ошибок в таких модулях будет минимальным.

Đối với Z ≤ 10, các mô-đun được kiểm tra chính xác và số lỗi trong các mô-đun đó sẽ là tối thiểu.

* Приемлемыми значениями считаются 10 ≤ Z ≤ 30.

Giá trị chấp nhận được là 10 ≤ Z ≤ 30.

* При Z > 30 устранить ошибки в процессе тестирования практически невозможно.

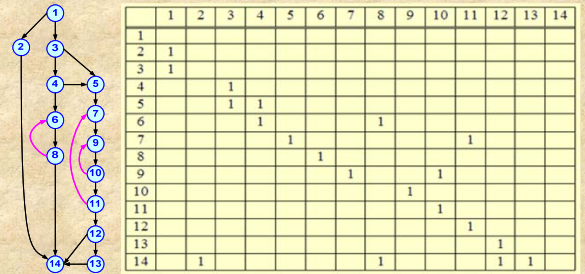
Ở Z>30, gần như không thể loại bỏ lỗi trong quá trình thử nghiệm.

Для **автоматического анализа** **графов** по второму критерию с помощью ЭВМ используются матрицы смежности и достижимости графов

Để tự động phân tích biểu đồ theo tiêu chí thứ hai bằng máy tính, ma trận kề và ma trận khả năng tiếp cận của đồ thị được sử dụng.

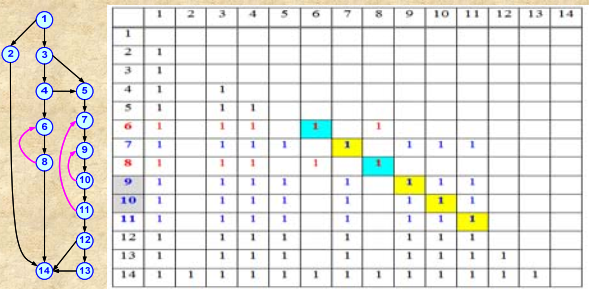
**Матрица смежности** – это квадратная матрица, в которой 1 располагается в позиции (i, j), если в графе имеется дуга (i, j).

Ma trận kề là một ma trận vuông trong đó 1 nằm ở vị trí (i, j) nếu đồ thị có một cung (i, j).



**Матрица достижимости** – это квадратная матрица, в которой 1 располагается в позиции, соответствующей дуге (i, j), **причем 1 ставится для всех вершин, до которых можно «достать».**

Ma trận khả năng tiếp cận là một ma trận vuông trong đó 1 nằm ở vị trí tương ứng với cung (i, j) và 1 được đặt cho tất cả các đỉnh mà bạn có thể "đạt tới".

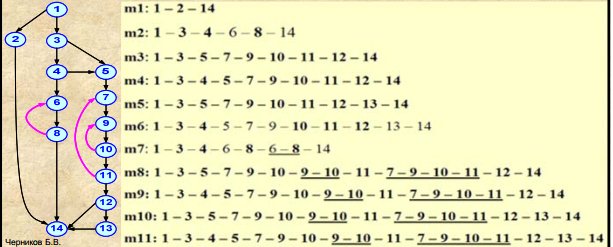


С помощью матрицы достижимости можно сравнительно просто выделить **циклы**, **отмечая диагональные элементы, равные 1**, и идентичные строки (6 – 8, 9 – 10, 7 – 9 – 10 – 11).

1. **Критерий 3**

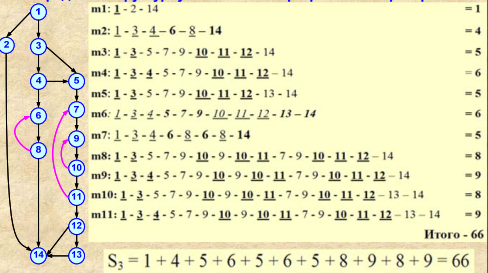
Основан на выделении полного состава базовых структур графа программы и заключается **в анализе хотя бы один раз каждого из реальных ациклических маршрутов исходного графа программы и каждого цикла, достижимого из всех этих маршрутов.**

Nó dựa trên việc làm nổi bật thành phần đầy đủ của các cấu trúc cơ bản của biểu đồ chương trình và bao gồm phân tích ít nhất một lần mỗi tuyến đường chu kỳ thực của biểu đồ chương trình gốc và mỗi chu kỳ có thể tiếp cận từ tất cả các tuyến này.

Если из некоторого ациклического маршрута исходного графа достижимы **несколько элементарных циклов**, то при тестировании должны исполняться все достижимые циклы.

Nếu một số chu kỳ cơ bản có thể truy cập được từ một số tuyến đường chu kỳ của biểu đồ nguồn, thì tất cả các chu trình có thể tiếp cận sẽ được thực hiện trong quá trình thử nghiệm.

Определим структурную сложность программы по критерию 3



Особенностью **четырех последних маршрутов с циклами** и **соответствующих им ациклических маршрутов** является полный перебор сочетаний ветвлений в вершинах 3 и 12.

Một tính năng của bốn tuyến đường cuối cùng với các vòng lặp và các tuyến đường đi vòng tương ứng của chúng là sự liệt kê đầy đủ các tổ hợp các nhánh ở các đỉnh 3 và 12.

### **Выводы по критериям оценки сложности программ**

* Критерии для оценки сложности программных модулей характеризуют в каждом случае минимально необходимые величины проверок по каждому критерию.

Việc đánh giá tính đầy đủ của kiểm tra chương trình khó khăn hơn nhiều, vì ngoài sự phức tạp của cấu trúc, cần phải phân tích độ phức tạp của việc chuyển đổi từng biến trong toàn bộ phạm vi thay đổi của nó và kết hợp với các biến khác.

* Для **проверки реальных программ** это количество проверок может быть недостаточно (например, циклы желательно проверять на одном-двух промежуточных значениях, а также на максимальном и минимальном количествах исполнения циклов).

Để kiểm tra các chương trình thực, số kiểm tra này có thể không đủ (ví dụ: nên kiểm tra các chu kỳ trên một hoặc hai giá trị trung gian, cũng như về số lượng thực hiện chu kỳ tối đa và tối thiểu).

* Оценить **достаточность проверок программы значительно труднее**, так как при этом, кроме сложности структуры, необходимо анализировать сложность преобразования каждой переменной во всем диапазоне ее изменения и при сочетаниях с другими переменными.

Việc đánh giá tính đầy đủ của kiểm tra chương trình khó khăn hơn nhiều, vì ngoài sự phức tạp của cấu trúc, cần phải phân tích độ phức tạp của việc chuyển đổi từng biến trong toàn bộ phạm vi thay đổi của nó và kết hợp với các biến khác.

## **МЕТРИКА МАККЕЙБА**

Основана **на анализе потока передачи управления** от одного оператора к другому, что позволяет учесть логику программы.

Dựa trên phân tích luồng chuyển điều khiển từ toán tử này sang toán tử khác, cho phép bạn tính đến logic của chương trình.

Программа (алгоритм, спецификация) должна быть представлена **в виде управляющего ориентированного графа**.

Chương trình (thuật toán, đặc điểm kỹ thuật) phải được trình bày dưới dạng biểu đồ hướng điều khiển.

**G = (V, Е)**

с V вершинами и E дугами, где **вершины соответствуют операторам, а дуги – переходам от одного оператора к другому.**

với các đỉnh V và các cung E, trong đó các đỉnh tương ứng với các toán tử và các cung tương ứng với các chuyển đổi từ toán tử này sang toán tử khác.

Граф, описывающий программу **в виде вершин-операторов и дуг-переходов**, называют графом управления или управляющим графом программы.

Biểu đồ mô tả chương trình ở dạng đỉnh toán tử và cung tròn chuyển tiếp được gọi là biểu đồ điều khiển hoặc biểu đồ điều khiển của chương trình.

Обычно учитывают только **исполнимые операторы**, исключая операторы описания данных.

Thông thường chỉ xem xét các câu lệnh thực thi, không bao gồm các câu lệnh mô tả dữ liệu.

**Линейные участки программы** можно заменить **одним узлом графа**.

Các phần tuyến tính của chương trình có thể được thay thế bằng một nút duy nhất trong biểu đồ.

Желательно **преобразовать операторы цикла** в эквивалентную последовательность операторов ветвления, добавив операторы суммирования (счетчики) числа повторений цикла с «верхним» или «нижним» окончанием.

Nên chuyển đổi các toán tử vòng lặp thành một chuỗi các toán tử phân nhánh tương đương bằng cách thêm các toán tử tổng hợp (bộ đếm) số lần lặp lại với một kết thúc trên đường trên hoặc trên đường dưới.

**Метрика Маккейба** является цикломатическим числом графа управления программы.

Số liệu McCabe là số chu kỳ của đồ thị điều khiển chương trình.



**m** – количество ребер графа (m là số cạnh của đồ thị)

**n** – количество вершин графа (n là số đỉnh của đồ thị)

Величину М называют цикломатическим числом Маккейба.

Giá trị của M được gọi là số McCabe chu kỳ.

Цикломатическая сложность программы – **структурная** (или топологическая) **мера сложности** программ, используемая для измерения качества программного обеспечения, **основанная на методах статического анализа кода.**

Độ phức tạp của chương trình theo chu kỳ là một phép đo cấu trúc (hoặc cấu trúc liên kết) về độ phức tạp của chương trình được sử dụng để đo lường chất lượng phần mềm, dựa trên các phương pháp phân tích mã tĩnh.

**Цикломатическая сложность** программы равна увеличенному на единицу **цикломатическому числу графа программы**.

Độ phức tạp chu kỳ của chương trình bằng số chu kỳ của đồ thị chương trình tăng thêm một.

*Разработана Томасом Дж. Маккейбом в 1976 году.*

При вычислении цикломатической сложности используется **граф потока управления программы:** узлы графа соответствуют неделимым группам команд программы и ориентированным ребрам, каждый из которых **соединяет два узла** и соответствует **двум командам**, **вторая из которых может быть выполнена сразу после первой.**

Khi tính toán độ phức tạp chu kỳ, biểu đồ luồng điều khiển chương trình được sử dụng: các nút của biểu đồ tương ứng với các nhóm lệnh chương trình và các cạnh được định hướng, mỗi nút kết nối hai nút và tương ứng với hai lệnh, lần thứ hai có thể được thực hiện ngay sau lần đầu tiên.

Эта стратегия тестирования называется основным маршрутом тестирования Маккейба: **тестирование каждого линейного независимого маршрута через программу** – в этом случае **число тестов должно быть равно цикломатической сложности программы**.

Chiến lược thử nghiệm này được gọi là tuyến thử nghiệm chính mccabe: thử nghiệm từng tuyến độc lập tuyến tính thông qua chương trình - trong trường hợp này, số lượng thử nghiệm phải bằng với độ phức tạp chu kỳ của chương trình.

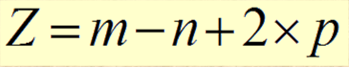
Цикломатическая сложность части программного кода – **счетное число линейно независимых маршрутов** через программный код

Độ phức tạp chu kỳ của một đoạn mã chương trình là số lượng các tuyến độc lập tuyến tính thông qua mã chương trình.



В теории графов цикломатическое число ориентированного графа

Trong lý thuyết đồ thị, số chu kỳ của đồ thị có hướng



**m** – количество ребер (số lượng xương sườn).

**n** – количество вершин (số đỉnh).

**p** – количество компонентов связности графа (số thành phần được kết nối của đồ thị).

Число компонентов связности p можно рассматривать как **минимально необходимое количество ребер, которые** **нужно добавить к графу,** **чтобы** сделать его полносвязным.

Số lượng các thành phần được kết nối p có thể được coi là số cạnh yêu cầu tối thiểu cần được thêm vào biểu đồ để làm cho nó được kết nối đầy đủ.

Справедливо считать, что для любого графа управления программы число компонентов связности равно единице (р = 1).

Thật công bằng khi giả định rằng đối với bất kỳ biểu đồ điều khiển chương trình nào, số lượng các thành phần được kết nối bằng với sự thống nhất (p = 1).

Подстановка p = 1 в формулу определения цикломатического числа дает цикломатическое число Маккейба – определяет **количество независимых контуров в полносвязном графе** и, как следствие, **количество различных путей, ведущих из начальной вершины в конечную.**

Thay thế p = 1 vào công thức xác định số chu kỳ cho số McCabe chu kỳ - nó xác định số lượng đường viền độc lập trong biểu đồ được kết nối đầy đủ và kết quả là, số lượng các đường khác nhau dẫn từ đỉnh ban đầu đến cuối cùng.

При оценке сложности программы с использованием цикломатического числа Маккейба действует правило: **если цикломатическое число Z > 10**, программа обладает излишней сложностью и ее следует разбить на составные части с меньшим значением цикломатического числа.

Khi đánh giá độ phức tạp của chương trình bằng số McCabe chu kỳ, quy tắc được áp dụng: nếu số chu kỳ Z> 10, chương trình phức tạp không cần thiết và nên được chia thành các thành phần có số chu kỳ thấp hơn.

# *LEK.#3* **ТЕМА 3. МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ**

**CÁC PHƯƠNG PHÁP PHÁT TRIỂN ALGORITHM**

## **МЕТОД ДЕКОМПОЗИЦИИ**

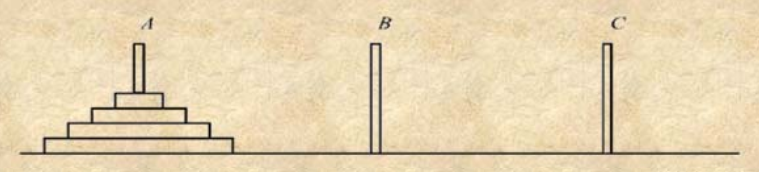
**PHƯƠNG PHÁP KHAI THÁC**

Называется также методом **«разделяй и властвуй»**, или **методом** **разбиения**, и, возможно, является самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов.

Nó cũng được gọi là phương pháp chia và chinh phục, hoặc phương pháp phân vùng, và được cho là phương pháp quan trọng nhất và được áp dụng rộng rãi nhất để thiết kế các thuật toán hiệu quả.

Предполагает такую **декомпозицию (разбиение) задачи на более мелкие задачи**, что на основе решений этих более мелких задач можно легко получить решение исходной задачи.

Nó giả định sự phân rã (phân vùng) của vấn đề thành các vấn đề nhỏ hơn, dựa trên các giải pháp của những vấn đề nhỏ hơn này, thật dễ dàng để có được một giải pháp cho vấn đề ban đầu.



Цель головоломки − перемещать диски (по одному) со стержня на стержень так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы в конце концов **все диски** оказались нанизанными **на стержень B**.

Mục tiêu của câu đố là di chuyển các đĩa (từng cái một) từ thanh này sang que sao cho đĩa có đường kính lớn hơn không bao giờ được đặt cao hơn đĩa có đường kính nhỏ hơn và do đó cuối cùng tất cả các đĩa được xâu vào thanh B

Задачу размера **n** перемещения наименьших дисков со стержня A на стержень B можно представить себе состоящей **из двух подзадач размера n – 1.**

Vấn đề về kích thước n của việc di chuyển các đĩa nhỏ nhất từ ​​thanh A sang thanh B có thể được biểu diễn bao gồm hai nhiệm vụ có kích thước n – 1.

Сначала нужно переместить n − 1 наименьших дисков со стержня A на стержень C, оставив на стержне A n-й наибольший диск. Затем этот диск нужно переместить с A на B. Потом следует переместить n − 1 дисков со стержня C на стержень B.

Trước tiên, bạn cần di chuyển n - 1 đĩa nhỏ nhất từ ​​thanh A sang thanh C, để lại đĩa lớn thứ n trên thanh A. Sau đó, bạn cần di chuyển đĩa này từ A sang B. Sau đó, bạn cần di chuyển n - 1 đĩa từ thành viên C sang thành viên B.

Это перемещение n − 1 дисков выполняется путем рекурсивного применения указанного метода.

Chuyển động này của các đĩa n - 1 được thực hiện bằng cách áp dụng đệ quy phương thức đã chỉ định.

Поскольку диски, участвующие в перемещениях, по размеру меньше тех, которые в перемещении не участвуют, не нужно задумываться над тем, что находится под перемещаемыми дисками на стержнях A, B или C.

Vì các đĩa liên quan đến các chuyển động nhỏ hơn các đĩa không tham gia vào chuyển động, nên bạn không cần phải suy nghĩ về những gì nằm dưới các đĩa trên các thanh A, B hoặc C.

**Легкость разработки алгоритмов по методу декомпозиции обусловила популярность этого метода.**

Việc dễ dàng phát triển các thuật toán bằng phương pháp phân rã đã khiến phương thức này trở nên phổ biến.

**Во многих случаях эти алгоритмы оказываются более эффективными, чем алгоритмы, разработанные традиционными методами.**

Trong nhiều trường hợp, các thuật toán này hiệu quả hơn các thuật toán được phát triển bởi các phương pháp truyền thống.

## **ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

**LẬP TRÌNH ĐỘNG**

Суть метода можно объяснить на простом примере чисел Фибоначчи.

Bản chất của phương pháp có thể được giải thích bằng một ví dụ đơn giản về các số Fibonacci.

Вычислить **N** чисел в последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, ..., **где первые два члена равны единице**, а все остальные представляют собой **сумму двух предыдущих**, причем **N меньше 100.**

Tính N số trong chuỗi Fibonacci: 1, 1, 2, 3, 5, 8, ..., trong đó hai số hạng đầu tiên bằng một và tất cả các số khác là tổng của hai số trước, với N nhỏ hơn 100.

**#prog.1:**

function F(X:integer): longint;

begin

if (X = 1) or (X = 2) then F := 1

else F := F(X-1) + F(X-2)

end;



**#prog.2:**

var D: array[1..100] of longint;

function F(X: integer): longint;

begin

if D[X] = 0 then

if (X=1) or (X=2) then D[X] := 1

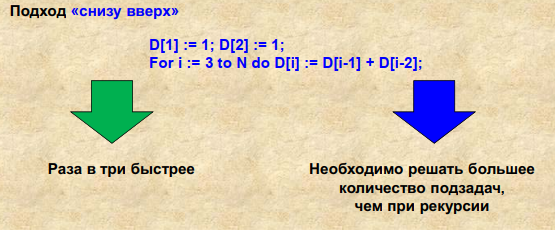
else D[X] := F(X-1) + F(X-2);

F := D[X];

end;

Такой подход динамического программирования называется **подходом «сверху вниз».**

Phương pháp lập trình động này được gọi là phương pháp từ trên xuống.



Часто приходится использовать как промежуточный результат нисходящую форму, а иногда безрекурсивная (итеративная) форма оказывается чрезвычайно сложной и малопонятной.

Thông thường cần phải sử dụng một hình thức giảm dần như một kết quả trung gian, và đôi khi một hình thức đệ quy (lặp lại) là vô cùng phức tạp và không thể hiểu được.

Если алгоритм **превышает** отведенное ему время на тестах большого объема, то необходимо осуществлять доработку этого алгоритма.

Nếu thuật toán vượt quá thời gian được phân bổ cho các bài kiểm tra khối lượng lớn, thì cần phải tinh chỉnh thuật toán này.

## **ПОИСК С ВОЗВРАТОМ**

Когда невозможно применить ни один из известных методов, способных помочь отыскать оптимальный вариант решения, остается прибегнуть **к полному перебору** (иначе – **поиск с возвратом**).

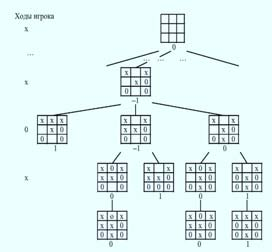
Khi không thể áp dụng bất kỳ phương pháp đã biết nào có thể giúp tìm ra giải pháp tối ưu, nó vẫn phải dùng đến một tìm kiếm hoàn chỉnh (nếu không - tìm kiếm có trả về).

Рассмотрим игру двух лиц, которые обычно описываются множеством «позиций» и совокупностью правил перехода из одной позиции в другую, причем предполагается, что игроки ходят по очереди.

Hãy xem xét một trò chơi hai người, thường được mô tả bởi vô số vị trí trên mạng, và một bộ quy tắc để di chuyển từ vị trí này sang vị trí khác và người ta cho rằng người chơi thay phiên nhau.

Будем считать, что правилами разрешены лишь **конечные последовательности позиций** и что в каждой позиции имеется лишь **конечное число разрешенных ходов.**

Chúng tôi giả định rằng các quy tắc chỉ cho phép các chuỗi vị trí hữu hạn và ở mỗi vị trí chỉ có một số lượng hữu hạn các bước di chuyển được phép.



При таких условиях для каждой позиции р найдется число N(p) такое, что никакая игра, начавшаяся в р, не может продолжаться более N(p) ходов.

Trong các điều kiện này, đối với mỗi vị trí p có một số N (p) sao cho không có trò chơi nào bắt đầu tại p có thể kéo dài hơn N (p) di chuyển.

Терминальными называются **позиции, из которых нет разрешенных ходов**. На каждой из них определена целочисленная функция f(p), задающая выигрыш того из игроков, которому принадлежит ход в этой позиции; выигрыш второго игрока считается равным –f(p).

Vị trí đầu cuối là những vị trí không được phép di chuyển. Trên mỗi trong số chúng, một hàm số nguyên f (p) được xác định, xác định mức chi trả của người chơi sở hữu di chuyển ở vị trí này; người chơi thứ hai giành chiến thắng được coi là ngang bằng vớiff (p).

Если из позиции p имеется d разрешенных ходов р1 , … pd , возникает проблема выбора лучшего из них. Будем называть ход наилучшим, если по окончании игры он приносит наибольший возможный выигрыш при условии, что противник выбирает ходы, наилучшие для него (в том же смысле).

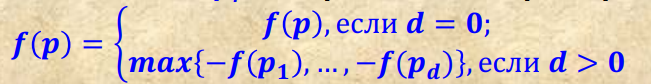
Nếu từ vị trí p có d di chuyển được phép p1, ... pd, thì vấn đề nảy sinh là chọn ra thứ tốt nhất trong số chúng. Chúng tôi sẽ gọi di chuyển là tốt nhất nếu vào cuối trò chơi, anh ta mang lại chiến thắng lớn nhất có thể, với điều kiện đối thủ chọn các nước đi phù hợp nhất với anh ta (theo nghĩa tương tự).

Пусть f(p) есть наибольший выигрыш, достижимый в позиции p игроком, которому принадлежит очередь хода, против оптимальной защиты.

Đặt f (p) là mức tăng lớn nhất có thể đạt được ở vị trí p bởi người chơi sở hữu lượt quay, chống lại phòng thủ tối ưu.

Так как после хода в позицию pi выигрыш этого игрока равен –f(pi), имеем:

Vì sau khi đi đến vị trí pi, mức tăng của người chơi này là Mạnhf (pi), chúng ta có:



Формула позволяет индуктивно определить f(p) для каждой позиции p

Công thức cho phép tự xác định f (p) cho từng vị trí p.

Функция f(p) равна максимуму, который гарантирован, если **оба игрока действуют оптимально.**

Hàm f (p) bằng với mức tối đa được đảm bảo nếu cả hai người chơi hành động tối ưu.

Однако эта функция отражает результаты осторожной стратегии, которая не всегда хороша против плохих игроков или игроков, действующих по иному принципу оптимальности.

Tuy nhiên, chức năng này phản ánh kết quả của một chiến lược thận trọng không phải lúc nào cũng tốt đối với người chơi xấu hoặc người chơi hành động theo nguyên tắc lạc quan khác.

Пусть, например, имеются два хода в позиции р1 и р2 , причем р1 гарантирует ничью (выигрыш 0) и не дает возможности выиграть, в то время как р2 дает возможность выиграть, если противник просмотрит очень тонкий выигрывающий ход.

Ví dụ, giả sử có hai nước đi ở vị trí p1 và p2, trong đó p1 đảm bảo hòa (thắng 0) và không cho cơ hội chiến thắng, trong khi p2 tạo cơ hội chiến thắng nếu đối thủ nhìn thấy nước cờ thắng rất mỏng.

В такой ситуации **можно предпочесть рискованный** ход в р2, если нет уверенности в том, что противник всемогущ и всезнающ.

Trong tình huống như vậy, người ta có thể thích di chuyển rủi ro trong p2, nếu không có gì chắc chắn rằng kẻ thù là toàn năng và toàn tri.

Следующий алгоритм, называемый поиском с возвратом, вычисляет f(p).

Thuật toán sau, được gọi là tìm kiếm trả về, tính toán f (p).

function BackSearch(p: position): integer;

{оценивает и возвращает выигрыш f(p) для позиции р}

var m,i,t,d: integer;

begin

Определить позиции p1,...,pd, подчиненные p;

if d = 0 then BackSearch := f(p)

else

begin

m := −∞;

for i:= 1 to d do

begin

t := − BackSearch(pi);

if t > m then m:= t;

end;

BackSearch := m;

end;

end;

Здесь +∞ обозначает число, которое не меньше abs(f(p)) для любой терминальной позиции р; поэтому −∞ не больше f(p) и −f(p) для всех р. Этот алгоритм вычисляет f(p) на основе «грубой силы» − для каждой позиции он оценивает все возможные продолжения.

Ở đây +∞ biểu thị một số không nhỏ hơn abs (f (p)) cho bất kỳ vị trí đầu cuối p; do đó nhiều nhất là f (p) và −f (p) cho tất cả p. Thuật toán này tính toán f (p) trên cơ sở lực lượng vũ phu, -∞ đối với mỗi vị trí, nó ước tính tất cả các phần tiếp theo có thể.

Для наиболее интересных игр размер дерева является чрезвычайно огромным, порядка WL, где W − среднее количество ходов в позиции, а L − количество уровней дерева.

Đối với các trò chơi thú vị nhất, kích thước cây cực kỳ lớn, theo thứ tự của WL, trong đó W là số lần di chuyển trung bình ở một vị trí và L là số cấp độ của cây.

## **МЕТОД ВЕТВЕЙ И ГРАНИЦ**

Идея метода состоит в том, что можно не искать точную оценку хода, про который стало известно, что он **не может быть лучше**, чем **один из ходов, рассмотренных ранее.**

Ý tưởng của phương pháp là bạn không thể tìm kiếm một ước tính chính xác về di chuyển, về điều mà người ta biết rằng nó không thể tốt hơn một trong những động thái được xem xét trước đó.

Пусть в процессе перебора стало известно, что f(p1) = −10.

Để trong quá trình liệt kê, người ta biết rằng f (p1) = −10.

Отсюда заключаем, что f(p) > 10, и потому не нужно знать точное значение f(p), если мы каким-либо образом узнали, что f(p2) > −10 (поскольку отсюда следует, что –f(p2) ≤ 10).

Từ đó, chúng tôi kết luận rằng f (p)> 10, và do đó chúng tôi không cần biết giá trị chính xác của f (p) nếu bằng cách nào đó chúng tôi đã học được rằng f (p2)> 10 (vì nó tuân theo điều đóf (p2) ≤ 10).

Итак, если p21 − допустимый ход из p2 и f(p21) ≤ 10, можно не исследовать другие ходы из p2.

Vì vậy, nếu p21 là một động thái được chấp nhận từ p2 và f (p21) 10, thì chúng ta không thể điều tra các động thái khác từ p2.

Говорят, что ход в позицию p2 «опровергается» (ходом в p1), если у противника в позиции p2 есть ответ столь же хороший, как его лучший ответ в позиции p1.

Người ta nói rằng việc di chuyển đến vị trí p2 là đã bác bỏ điều đó (bằng cách chuyển sang p1) nếu đối thủ ở vị trí p2 có câu trả lời tốt như câu trả lời hay nhất của anh ta ở vị trí p1.

Ясно, что если ход можно опровергнуть, **можно не искать наилучшее опровержение.**

Rõ ràng là nếu một động thái có thể bị bác bỏ, bạn không thể tìm kiếm sự bác bỏ tốt nhất.

Определим метод «ветвей и границ» как процедуру с двумя параметрами р и bound, вычисляющую f '(p, bound).

Цель алгоритма − удовлетворить следующим условиям:

f(p, bound) = f(p), если f(p) < bound,

f(p, bound) > bound, если f(p) ≥ bound

Идею метода ветвей и границ реализует следующий алгоритм:

Function B&B(p: position, bound: integer): integer;

{оценивает и возвращает выигрыш F'(p) для позиции p}

label done;

var m,i,t,d: integer;

begin

Определить позиции p1, … ,pd, подчиненные p;

if d = 0 then B&B := f(p) else

begin

m := −∞;

for i:= 1 to d do

begin

t := −B&B(p1, −m);

if t > m then m := t;

if m >= bound then goto done;

end;

done:

B&B := m;

end;

end;

## **МЕТОД АЛЬФА-БЕТА ОТСЕЧЕНИЯ**

Метод «ветвей и границ» можно еще улучшить, если ввести не только **верхнюю**, но и **нижнюю границу.**

Có thể cải thiện hơn nữa chi nhánh và phương pháp ràng buộc trên phạm vi bằng cách giới thiệu không chỉ giới hạn trên mà cả ranh giới phía dưới.

Эта идея (ее называют минимаксной альфа-бета процедурой или просто альфа-бета отсечением) является значительным продвижением по сравнению с односторонним методом «ветвей и границ»

Ý tưởng này (được gọi là thủ tục alpha-beta minimax hoặc chỉ cắt alpha-beta) là một bước tiến đáng kể so với phương pháp phân nhánh và liên kết một chiều.

Определим процедуру f" с параметрами p, alpha и beta (причем всегда будет выполнено alpha < beta), которая удовлетворяет следующим условиям:

Chúng tôi xác định thủ tục f "với các tham số p, alpha và beta (hơn nữa, alpha <beta sẽ luôn được đáp ứng), đáp ứng các điều kiện sau:

f"(p, alpha, beta) ≤ alpha, если f(p) < alpha

f"(p, alpha, beta) = f(p), если alpha < f(p) < beta

f"(p, alpha, beta) ≥ beta, если f(p) > beta

function AB(p: position; alpha, beta: integer): integer;

{оценивает и возвращает выигрыш F''(p) для позиции р}

label done;

var m,i,t,d: integer;

begin

{Определить позиции p1, ... ,pd, подчиненные p;}

if d = 0 then AB := f(p)

else

begin

m := alpha;

for i:= 1 to d do

begin

t := −AB(p1, −beta, −m);

if t > m then m := t;

if m >= beta then goto done;

end;

done:

AB := m;

end;

end;

Выгода от альфа-бета отсечения заключается в более раннем выходе из цикла.

Lợi ích của việc cắt alpha-beta là thoát ra khỏi vòng lặp trước đó.

Эти отсечения полностью безопасны (корректны), потому что они гарантируют, что отсекаемая часть дерева хуже, чем основной вариант.

Những đoạn này hoàn toàn an toàn (chính xác), bởi vì chúng đảm bảo rằng phần cắt của cây xấu hơn phiên bản chính.

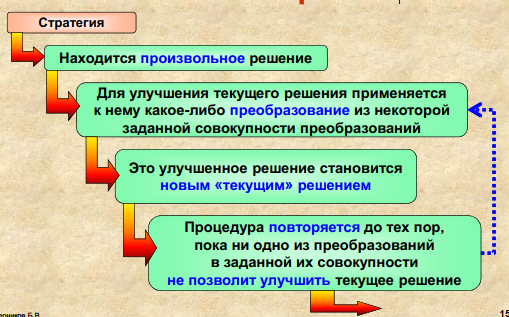
При оптимальных обстоятельствах перебор с альфа-бета отсечением должен просмотреть W(L+1)/2 + W(L/2−1) позицию, где W − среднее количество ходов в позиции, L – количество уровней дерева.

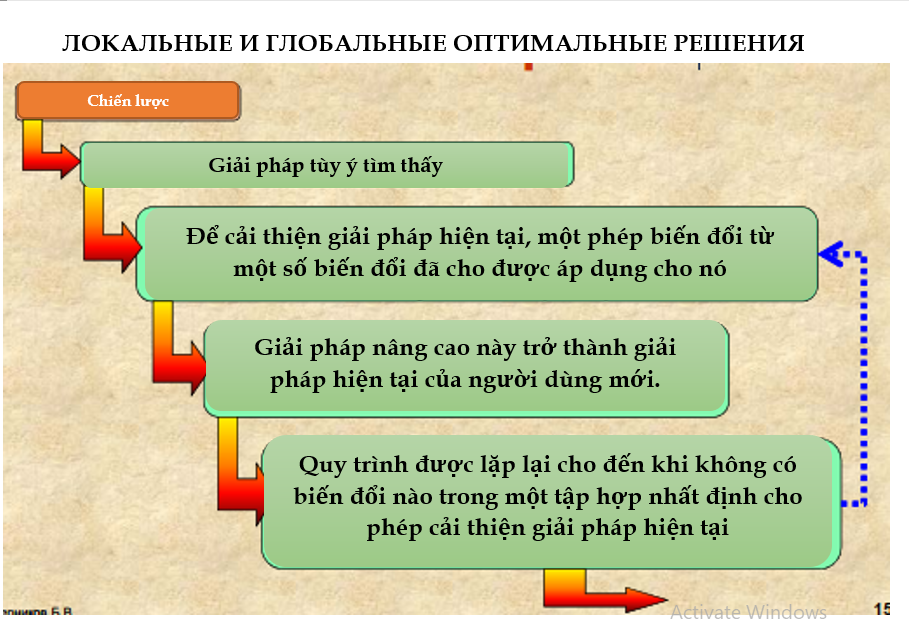
Trong trường hợp tối ưu, tìm kiếm với alpha-beta bằng cách cắt nên nhìn vào vị trí W (L + 1) / 2 + W (L / 2−1), trong đó W là số lần di chuyển trung bình ở một vị trí, L là số cấp độ của cây.

Это **намного меньше, чем перебор с возвратом** − данное отсечение позволяет достигать примерно вдвое большей глубины за то же самое время.

Điều này ít hơn nhiều so với việc quay trở lại với sự quay trở lại - việc cắt này cho phép bạn đạt được độ sâu xấp xỉ gấp đôi trong cùng một lúc.

## **ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ОПТИМАЛЬНЫЕ РЕШЕНИЯ**

****

****

Метод имеет смысл лишь в том случае, когда можно ограничить совокупность преобразований небольшим ее подмножеством, что дает возможность выполнить все преобразования за относительно короткое время: если «размер» задачи равняется n, то можно допустить O(n2) или O(n3) преобразований.

Phương thức này chỉ có ý nghĩa nếu có thể giới hạn tập hợp các phép biến đổi thành một tập con nhỏ của nó, điều này cho phép thực hiện tất cả các phép biến đổi trong một thời gian tương đối ngắn: nếu "kích thước" của vấn đề là n, thì phép biến đổi O (n2) hoặc O (n3) có thể được phép.

Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как «близкие».

Nếu tổng số biến đổi là nhỏ, việc xem xét các giải pháp có thể biến đổi thành một bước khác là một cách tự nhiên.

Такие преобразования называются локальными, а соответствующий метод называется локальным поиском.

Các phép biến đổi như vậy được gọi là cục bộ và phương thức tương ứng được gọi là tìm kiếm cục bộ.

Одной из задач, которую можно решить именно методом локального поиска, является задача нахождения минимального остовного дерева – речь идет о свойстве **MST (minimal spanning tree − минимальное остовное дерево).**

Một trong những nhiệm vụ có thể được giải quyết chính xác bằng phương pháp tìm kiếm cục bộ là vấn đề tìm cây bao trùm tối thiểu - chúng ta đang nói về thuộc tính MST (cây bao trùm tối thiểu - cây bao trùm tối thiểu).

Остовное дерево − **ациклический связный подграф** данного связного неориентированного графа, в который входят **все его вершины.**

Cây bao trùm là một sơ đồ con được kết nối theo chu kỳ của một đồ thị vô hướng được kết nối đã cho, bao gồm tất cả các đỉnh của nó.

В принципе, остовное дерево состоит из некоторого **подмножества ребер** графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим ребрам, и в нем нет циклов, то есть **из любой вершины нельзя попасть в саму себя, не пройдя какое-то ребро дважды.**

Về nguyên tắc, một cây bao gồm một tập hợp con các cạnh của đồ thị, sao cho từ bất kỳ đỉnh nào của đồ thị bạn có thể đến bất kỳ đỉnh nào khác bằng cách di chuyển dọc theo các cạnh này, và không có chu kỳ nào trong đó, từ bất kỳ đỉnh nào bạn có thể tự đi vào mà không đi qua một số xương sườn hai lần.

Минимальное остовное дерево (или минимальное покрывающее дерево) в связанном взвешенном неориентированном графе − это остовное дерево этого графа, имеющее **минимальный возможный вес**, где под весом дерева понимается **сумма весов входящих в него ребер.**

Cây bao trùm nhỏ nhất (hoặc cây bao trùm tối thiểu) trong biểu đồ vô hướng có trọng số được kết nối là cây bao trùm của biểu đồ này có trọng lượng nhỏ nhất có thể, trong đó trọng lượng của cây là tổng trọng số của các cạnh của nó.

Допустим, есть n городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города).

Giả sử có n thành phố cần được kết nối bằng đường bộ, để bạn có thể đi từ bất kỳ thành phố nào đến bất kỳ thành phố nào khác (trực tiếp hoặc thông qua các thành phố khác).

Разрешается строить дороги между заданными парами городов и известна стоимость строительства каждой такой дороги.

Nó được phép xây dựng các con đường giữa các cặp thành phố nhất định và chi phí xây dựng mỗi con đường như vậy được biết đến.

Требуется решить, **какие именно дороги нужно строить**, чтобы **минимизировать общую стоимость** строительства.

Cần phải quyết định những con đường nào cần được xây dựng để giảm thiểu tổng chi phí xây dựng.

Эта задача может быть сформулирована в терминах теории графов как задача о нахождении минимального остовного дерева в графе:

Vấn đề này có thể được hình thành theo lý thuyết đồ thị là vấn đề tìm cây bao trùm tối thiểu trong biểu đồ:

* Вершины графа представляют города

Phần trên của biểu đồ đại diện cho các thành phố

* Ребра − это пары городов, между которыми можно проложить прямую дорогу

Sườn là cặp thành phố mà bạn có thể làm đường trực tiếp

* Вес ребра равен стоимости строительства соответствующей дороги

Trọng lượng của xương sườn bằng với chi phí xây dựng con đường tương ứng.

В графе G = (V, E) рассмотрим U − некоторое подмножество V, такое что U и V \ U не пусты.

Trong biểu đồ G = (V, E) chúng ta xem xét U, một tập hợp con của V sao cho U và V \ U không trống.

Квантор \ означает разность множеств, т.е. V \ U означает множество элементов, принадлежащих V, но не принадлежащих U.

Quantifier \ có nghĩa là sự khác biệt của các bộ, tức là V \ U có nghĩa là tập hợp các phần tử thuộc V, nhưng không thuộc về U.

Пусть (u, v) − ребро наименьшей стоимости, одна вершина которого − u ∈ U, а другая − v ∈ V \ U.

Đặt (u, v) là cạnh có giá trị nhỏ nhất, một đỉnh trong đó là u ∈ U và đỉnh kia là v ∈ V \ U.

Тогда существует некоторое MST, содержащее ребро (u, v).

Sau đó, tồn tại một số MST chứa cạnh (u, v).

|  |  |
| --- | --- |
|  |  |
| Граф | Минимальное остовное дерево графа |

Локальными являются такие преобразования, в ходе которых берется **то или иное ребро, не относящееся к текущему остовному дереву,** оно **добавляется** в это дерево (в результате должен получиться цикл), а затем **убирается** из этого цикла **в точности одно ребро** (предположительно, ребро с наивысшей стоимостью), чтобы образовалось новое дерево.

Biến đổi cục bộ là những biến đổi trong đó một hoặc một cạnh khác không liên quan đến cây bao trùm hiện tại, nó được thêm vào cây này (kết quả phải là một chu kỳ), và sau đó chính xác một cạnh được loại bỏ khỏi chu kỳ này (có lẽ là cạnh có giá trị cao nhất chi phí) để tạo thành một cây mới.

Время, которое занимает выполнение этого алгоритма на графе из n узлов и e ребер, зависит от количества требующихся улучшений решения

Thời gian cần thiết để thực hiện thuật toán này trên đồ thị gồm n nút và cạnh e phụ thuộc vào số lượng cải tiến cần thiết cho giải pháp

Только проверка факта, что преобразования **уже неприменимы**, может занять O(n−e) времени, поскольку для этого необходимо перебрать e ребер, а каждое из них может образовать цикл длиной примерно n.

Chỉ xác minh thực tế rằng các phép biến đổi không còn áp dụng được có thể mất thời gian O (n - e), bởi vì điều này là cần thiết để lặp lại trên các cạnh e và mỗi trong số chúng có thể tạo thành một chu kỳ với độ dài xấp xỉ n.

Алгоритмы локального поиска проявляют себя с наилучшей стороны как **эвристические алгоритмы** для решения задач, точные решения которых **требуют экспоненциальных затрат времени** (относятся к классу EXPTIME)

Các thuật toán tìm kiếm cục bộ thể hiện khả năng tốt nhất của chúng là các thuật toán heuristic để giải quyết các vấn đề mà các giải pháp chính xác của nó đòi hỏi thời gian mở rộng theo cấp số nhân (thuộc về lớp EXPTIME)

Начать следует с ряда произвольных решений, применяя к каждому из них локальные преобразования до тех пор, пока не будет получено локально-оптимальное решение, т. е. такое, которое не сможет улучшить ни одно преобразование

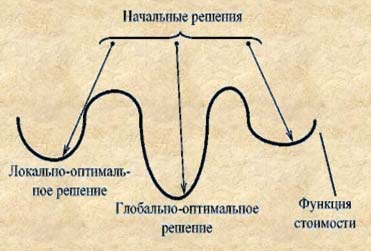
Bạn nên bắt đầu với một số giải pháp tùy ý, áp dụng các phép biến đổi cục bộ cho từng giải pháp cho đến khi có được giải pháp tối ưu cục bộ, nghĩa là, một giải pháp không thể cải thiện bất kỳ chuyển đổi nào

Видно, что на основе большинства произвольных начальных решений могут получаться разные **локально-оптимальные решения.**

Có thể thấy rằng, dựa trên hầu hết các giải pháp ban đầu tùy ý, có thể thu được các giải pháp tối ưu cục bộ khác nhau.

Возможно, одно из таких решений окажется глобально-оптимальным, т. е. **лучше любого другого решения**

Có lẽ một trong những giải pháp này sẽ trở nên tối ưu toàn cầu, tức là tốt hơn bất kỳ giải pháp nào khác.



# *LEK.#4* **ТЕМА 4 - СТРУКТУРЫ ДАННЫХ (ЧАСТЬ 1)**

**ПОНЯТИЕ СТРУКТУРЫ ДАННЫХ**

Структура данных – множество элементов данных и множество связей между ними.

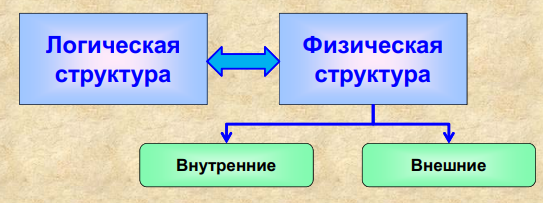
Cấu trúc dữ liệu - nhiều yếu tố dữ liệu và nhiều mối quan hệ giữa chúng.

Физическая структура данных отражает способ **физического представления данных в памяти машины** и называется еще структурой хранения, внутренней структурой или структурой памяти.

Cấu trúc dữ liệu vật lý phản ánh cách thức dữ liệu được thể hiện trong bộ nhớ của máy và còn được gọi là cấu trúc lưu trữ, cấu trúc bên trong hoặc cấu trúc bộ nhớ.

Структура данных **без учета** ее представления в машинной памяти – абстрактная или логическая структура.

Cấu trúc dữ liệu mà không tính đến biểu diễn của nó trong bộ nhớ máy tính là cấu trúc trừu tượng hoặc logic.

****

Элементарные – не могут быть расчленены на составные части, большие, чем биты.

Cấu trúc dữ liệu tổng hợp, các bộ phận cấu thành trong đó là các cấu trúc dữ liệu khác (sơ cấp hoặc, lần lượt, tổng hợp).

Составные – структуры данных, составными частями которых являются **другие структуры данных** (элементарные или, в свою очередь, составные).

Cấu trúc dữ liệu tổng hợp, các bộ phận cấu thành trong đó là các cấu trúc dữ liệu khác (sơ cấp hoặc, lần lượt, tổng hợp).

**Важный признак составной структуры данных** – характер упорядоченности ее частей

Một tính năng quan trọng của cấu trúc dữ liệu tổng hợp là bản chất của thứ tự các bộ phận của nó.

* Линейные (Tuyến tính)
* Нелинейные (phi tuyến)

Изменчивость – изменение числа элементов или связей между составными частями структуры

Tính biến đổi - thay đổi số lượng phần tử hoặc mối quan hệ giữa các bộ phận cấu thành của cấu trúc

* Статические (Tĩnh)
* Динамические (Năng động)

Типы данных <=>Структуры данных:

* Структура хранения данных указанного типа – выделение памяти, представление данных в ней и метод доступа к данным

Cấu trúc lưu trữ dữ liệu của loại được chỉ định - cấp phát bộ nhớ, biểu diễn dữ liệu trong đó và phương thức truy cập dữ liệ

* Множество допустимых значений, которые может иметь тот или иной объект описываемого типа

Tập hợp các giá trị hợp lệ mà một đối tượng thuộc loại được mô tả có thể có

* Набор допустимых операций, которые применимы к объекту описываемого типа

Một tập hợp các hoạt động hợp lệ áp dụng cho một đối tượng của loại được mô tả.

## **ЭЛЕМЕНТАРНЫЕ ДАННЫЕ**

* Представляют собой единое и неделимое целое.

Họ là một tổng thể duy nhất và không thể chia cắt.

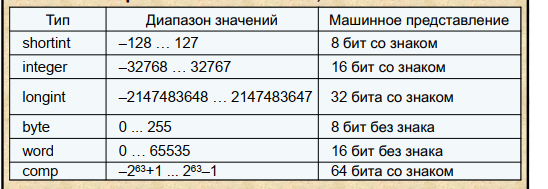
* В каждый момент времени они могут принимать только одно значение.

Tại mỗi thời điểm họ chỉ có thể nhận một giá trị.

**ДАННЫЕ ЦЕЛОЧИСЛЕННОГО ТИПА**

Представляется счетное число объектов, являющихся дискретными по своей природе.

Một số lượng đáng kể các đối tượng rời rạc trong tự nhiên được trình bày.

****Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать 1, 2 или 4 байта.

Phạm vi của các giá trị có thể có của các loại số nguyên phụ thuộc vào biểu diễn bên trong của chúng, có thể mất 1, 2 hoặc 4 byte.

Значения целочисленных типов представляются в памяти ЭВМ абсолютно точно.

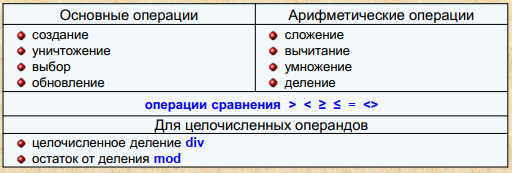
Giá trị của các loại số nguyên được biểu diễn trong bộ nhớ máy tính hoàn toàn đúng.

Значения вещественных типов определяют число лишь **с некоторой конечной точностью**, зависящей **от внутреннего формата вещественного числа.**

Giá trị của loại thực chỉ xác định một số với độ chính xác hữu hạn, tùy thuộc vào định dạng bên trong của số thực.



**Операции над числовыми типами**

****

**ДАННЫЕ СИМВОЛЬНОГО ТИПА**

Тип char (1 байт) – символы из некоторого предопределенного множества (ASCII – American Standard Code for Information Interchange).

Kiểu Char (1 byte) - các ký tự từ một số bộ được xác định trước (ASCII - Mã tiêu chuẩn Mỹ để trao đổi thông tin).

Одна таблица может поддерживать только один национальный алфавит.

Một bảng chỉ có thể hỗ trợ một bảng chữ cái quốc gia.

Этот недостаток преодолен во множестве UNICODE – каждый символ кодируется двумя байтами, что обеспечивает более 216 возможных кодовых комбинаций и дает возможность иметь единую таблицу кодов.

Hạn chế này được khắc phục trong bộ UNICODE - mỗi ký tự được mã hóa bằng hai byte, cung cấp hơn 216 tổ hợp mã có thể và làm cho có thể có một bảng mã duy nhất.

Операции над символьными типами – только операции сравнения.

Hoạt động loại nhân vật - Chỉ hoạt động so sánh.s

**ДАННЫЕ ЛОГИЧЕСКОГО ТИПА**

Значениями логического типа может быть одна из предварительно объявленных констант false (ложь) или true (истина).

Các giá trị Boolean có thể là một trong các hằng số được khai báo trước đó là false (false) hoặc true (true).

Данные логического типа занимают один байт памяти:

Dữ liệu Boolean chiếm một byte bộ nhớ:

* Значению false соответствует **нулевое** значение байта.
* Значению true – любое **ненулевое** значение байта.

Над логическими типами – операции булевой алгебры: НЕ (not), ИЛИ (or), И (and), ИСКЛЮЧАЮЩЕЕ ИЛИ (xor).

Trên các loại logic - hoạt động của đại số Boolean: KHÔNG (không), HOẶC (hoặc), AND (và), ĐỘC QUYỀN HOẶC (xor).

Результаты логического типа получаются при сравнении данных любых типов.

Kết quả loại logic có được bằng cách so sánh bất kỳ loại dữ liệu.

**ДАННЫЕ ТИПА УКАЗАТЕЛЬ**

Тип указателя представляет собой **адрес ячейки памят.**

Loại con trỏ là địa chỉ của ô nhớ.

В языках программирования высокого уровня определена специальная константа nil, которая означает **пустой указатель** или указатель, **не содержащий какой-либо конкретный адрес.**

Trong các ngôn ngữ lập trình cấp cao, một hằng số đặc biệt được xác định, có nghĩa là một con trỏ null hoặc một con trỏ không chứa bất kỳ địa chỉ cụ thể nào.

Указатели применяются (Con trỏ áp dụng):

* При необходимости представить одну и ту же область памяти (одни и те же физические данные) как данные разной логической структуры

Nếu cần, hãy trình bày cùng một vùng nhớ (cùng dữ liệu vật lý) với dữ liệu có cấu trúc logic khác nhau

* При работе с динамическими структурами данных

Khi làm việc với các cấu trúc dữ liệu động

Могут быть типизированными и нетипизированными

Có thể được gõ hoặc gỡ

var pointer

ipt: ^integer;

cpt: ^char;

**Основные операции:**

**Hoạt động cơ bản:**

* Присваивание – **двухместная операция**, оба операнда – **указатели.** (Bài tập là một hoạt động kép, cả hai toán hạng đều là con trỏ.)

Если оба указателя, участвующие в операции присваивания типизированные, то оба должны указывать на данные одного и того же типа.

Nếu cả hai con trỏ liên quan đến hoạt động gán được gõ, thì cả hai phải trỏ đến dữ liệu cùng loại.

* Получение адреса – **одноместная**, ее операнд может иметь **любой тип.** (Lấy một địa chỉ - đơn, toán hạng của nó có thể thuộc bất kỳ loại nào)

Результат – **типизированный указатель** (в соответствии с типом операнда), содержащий адрес операнда.

Kết quả là một con trỏ được gõ (theo loại toán hạng) chứa địa chỉ của toán hạng.

* Выборка – **одноместная**, ее операндом является **типизированный указатель.** (Lựa chọn là đơn, toán hạng của nó là một con trỏ được gõ.)

Результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется **типом указателя.**

Kết quả là dữ liệu được chọn từ bộ nhớ tại địa chỉ được chỉ định bởi toán hạng. Loại kết quả được xác định bởi loại con trỏ.

## **ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ**

Цель описания типа данных и определения некоторых переменных, относящихся к статическим типам, состоит в том, чтобы **зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти.**

Mục đích của việc mô tả kiểu dữ liệu và xác định một số biến liên quan đến kiểu tĩnh là để sửa phạm vi giá trị được gán cho các biến này và theo đó, kích thước của bộ nhớ được phân bổ cho chúng.

Поэтому такие переменные – статические.

Do đó, các biến như vậy là tĩnh.

**МАССИВ**

Массив – поименованная совокупность **однотипных элементов, упорядоченных по индексам**, определяющих положение элемента в массиве.

Mảng - một tập hợp các phần tử có cùng loại, được sắp xếp theo thứ tự xác định vị trí của phần tử trong mảng.

Задание имени массива, типа индекса и типа элементов:

Đặt tên mảng, loại chỉ mục và loại phần tử:

имя: array[ТипИндекса] of ТипЭлемента;

Массив бывает: **одномерным** (вектор), **двумерным** (матрица), **трехмерным** (куб)…

var

Vector: array [1..100] of integer;

Matrix: array [1..100, 1..100] of integer;

Cube: array [1..100, 1..100, 1..100] of integer;

Операции:

* Сравнение на равенство и неравенство массивов.

So sánh về sự bình đẳng và bất bình đẳng của mảng.

* Присвоение для массивов с одинаковым типом индексов и одинаковым типом элементов.

Bài tập cho các mảng có cùng loại chỉ mục và cùng loại phần tử.

Доступ к массивам в этих операциях – через имя массива без указания индексов

Truy cập vào các mảng trong các hoạt động này - thông qua tên của mảng mà không chỉ định các chỉ mục

Можно выполнять операции над **отдельными элементами** массива.

Bạn có thể thực hiện các thao tác trên các phần tử riêng lẻ của một mảng.

Перечень таких операций определяется **типом элементов.**

Danh sách các hoạt động như vậy được xác định bởi loại yếu tố.

Доступ к отдельным элементам массива осуществляется **через имя массива и индексы** элементов:

Truy cập vào các phần tử riêng lẻ của mảng được thực hiện thông qua tên mảng và chỉ số phần tử:

Cube[0,0,10] := 25;

Matrix[10,30] := Cube[0,0,10] + 1;

Элементы массива обычно располагаются **непрерывно**, в соседних ячейках.

Các phần tử của mảng thường được đặt liên tục, trong các ô liền kề.

Размер памяти, занимаемой массивом, есть **суммарный размер** элементов массива.

Kích thước của bộ nhớ chiếm bởi mảng là tổng kích thước của các phần tử của mảng.

**СТРОКА**

Строка – последовательность символов (элементов символьного типа).

Các phần tử của mảng thường được đặt liên tục, trong các ô liền kề.

Количество символов в строке (длина строки) может динамически меняться от 0 до 255:

Kích thước của bộ nhớ chiếm bởi mảng là tổng kích thước của các phần tử của mảng.

var

TTxt: string;

TWrd: string[10];

Каждый символ строки имеет свой индекс: от 1 до заданной длины строки.

Mỗi ký tự của một chuỗi có chỉ mục riêng: từ 1 đến độ dài chuỗi đã cho.

Элемент строки **с индексом 0 недоступен** с использованием индекса и содержит **текущее количество символов** в строке.

Mục hàng có chỉ mục 0 không khả dụng khi sử dụng chỉ mục và chứa số lượng ký tự hiện tại trong dòng.

**Операции:**

* Определенные для символьного типа данных. (Cụ thể cho kiểu dữ liệu ký tự.)
* Присвоение строк в целом. (Phân công các dòng nói chung.)
* Операции сравнения строк <, >, ≥, ≤, =, <>
* Операция сцепления (конкатенации) строк +

Символы строки располагаются **непрерывно**, в соседних ячейках.

Các ký tự dòng được đặt liên tục trong các ô liền kề.

Размер памяти, занимаемой массивом, есть **суммарный размер** элементов массива (включая элемент, содержащий длину строки).

Kích thước của bộ nhớ chiếm bởi mảng là tổng kích thước của các phần tử mảng (bao gồm cả phần tử chứa độ dài chuỗi).

**ЗАПИСЬ**

Запись – агрегат, составляющие которого (поля) имеют **имя** и могут быть **различного типа.**

Một bản ghi là một tập hợp có các thành phần (các trường) có tên và có thể có nhiều loại.

type

TPerson = record

Name: string;

Address: string;

Index:longint;

end;

var Person1: TPerson;

Операция присваивания для записей в целом – записи должны быть одного типа) Можно выполнять операции над отдельным полем.

Thao tác chuyển nhượng cho toàn bộ hồ sơ - các bản ghi phải cùng loại) Bạn có thể thực hiện các thao tác trên một trường riêng biệt.

Person1.Index := 117997;

Person1.Name := 'РЭУ им. Плеханова';

Person1.Adress := 'Москва, Стремянный переулок, д.36';

**МНОЖЕСТВО**

Множество – совокупность **однородных** элементов, объединенных **общим признаком** и представляемых как **единое целое.**

Một tập hợp là một tập hợp các phần tử đồng nhất được hợp nhất bởi một thuộc tính chung và được trình bày dưới dạng một tổng thể duy nhất.

Операции:

* Объединение множеств +
* Пересечение множеств \*
* Разность множеств –
* Проверка элемента на принадлежность к множеству in

var

RGB, YIQ, CMY: set of char;

CMY := ['M', 'C', 'Y'];

RGB := ['R', 'G', 'B'];

YIQ := ['Y', 'Q', 'I'];

Writeln('Пересечение цветовых схем RGB и CMY ',RGB\*CMY);

Writeln('Пересечение цветовых схем YIQ и CMY ',YIQ\*CMY);

**ТАБЛИЦА**

Таблица – одномерный массив (вектор), элементами которого являются **записи**.

Bảng là một mảng một chiều (vectơ) có các phần tử là các bản ghi.

Отдельная запись массива – **строка** таблицы

Nhập mảng riêng - hàng bảng.

Чаще всего используется **простая запись**, т. е. поля – **элементарные данные.**

Thông thường một bản ghi đơn giản được sử dụng, tức là các trường là dữ liệu cơ bản.

Совокупность одноименных полей всех строк – **столбец**, а конкретное поле отдельной строки – **ячейка.**

Tổng các trường giống nhau của tất cả các hàng là một cột và trường cụ thể của một hàng là một ô.

Доступ к элементам таблицы производится не по индексу, а по **ключу**, т. е. по значению одного из полей записи.

Truy cập vào các phần tử bảng được thực hiện không phải bằng chỉ mục, mà bằng khóa, tức là, bằng giá trị của một trong các trường bản ghi.

Ключ таблицы (основной, первичный) – поле, значение которого может быть использовано для **однозначной идентификации** каждой записи таблицы.

Khóa bảng (chính, chính) - một trường có giá trị có thể được sử dụng để xác định duy nhất mỗi mục nhập bảng.

Вторичный ключ – поле таблицы с несколькими ключами, **не** **обеспечивающий** (в отличие от первичного ключа) однозначной идентификации записей таблицы.

Khóa phụ - trường bảng có một số khóa không cung cấp (không giống như khóa chính) một nhận dạng duy nhất của các mục trong bảng.

Если последовательность записей упорядочена относительно какого-либо столбца (поля), то такая таблица называется **упорядоченной**, иначе – таблица **неупорядоченная**.

Nếu chuỗi các bản ghi được sắp xếp theo thứ tự liên quan đến bất kỳ cột (trường) nào, thì một bảng như vậy được gọi là có thứ tự, nếu không - một bảng bị rối loạn.

**Основная операция** – доступ к записи по ключу.

Các hoạt động chính là truy cập vào một bản ghi bằng phím.

Перечень операций над отдельной ячейкой определяется типом ячейки.

Danh sách các hoạt động trên một ô được xác định bởi loại ô.

PersonList[1].Index := 117997;

PersonList[1].Name := 'РЭУ им. Плеханова';

PersonList[1].Adress := 'Москва, Стремянный переулок, д.36';

Ячейки таблицы обычно располагаются построчно, непрерывно, в соседних ячейках.

Các ô bảng thường là hàng, liên tục, trong các ô liền kề.

Размер памяти, занимаемой таблицей, есть суммарный размер ячеек.

Kích thước của bộ nhớ bị chiếm bởi bảng là tổng kích thước của các ô.

**ЛИНЕЙНЫЕ СПИСКИ**

Список – структура данных, представляющая собой **логически** **связанную последовательность** элементов списка

Danh sách - cấu trúc dữ liệu là một chuỗi các mục danh sách được kết nối hợp lý

Динамическая структура данных – структура данных, определяющие характеристики которой могут изменятьсяна **на протяжении ее существования.**

Cấu trúc dữ liệu động là cấu trúc dữ liệu có các đặc điểm xác định có thể thay đổi trong suốt quá trình tồn tại của nó.

Отличие динамических структур от статических:

* В них нельзя обеспечить хранение в заголовке всей информации о структуре, поскольку каждый элемент должен содержать информацию, логически связывающую его с другими элементами структуры.

Không thể đảm bảo rằng tất cả thông tin về cấu trúc được lưu trữ trong tiêu đề, vì mỗi phần tử phải chứa thông tin kết nối hợp lý với phần tử khác của cấu trúc.

* Для них зачастую неудобно использовать единый массив смежных элементов памяти, поэтому необходимо предусматривать какую-либо схему динамического управления памятью.

Việc sử dụng một mảng các phần tử bộ nhớ liền kề thường là bất tiện, do đó cần phải cung cấp một số loại sơ đồ quản lý bộ nhớ động

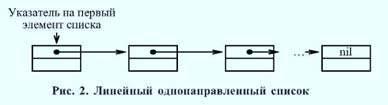
|  |  |
| --- | --- |
| Процедура создания  New(Current); | Процедура освобождения  Dispose(Current); |

#### **Линейный однонаправленный список**

Danh sách đơn hướng tuyến tính

В этом списке любой элемент имеет один указатель, который указывает **на следующий элемент** в списке или является **пустым указателем** у последнего элемента.

Trong danh sách này, bất kỳ mục nào cũng có một con trỏ trỏ đến mục tiếp theo trong danh sách hoặc là một con trỏ trống đến mục cuối cùng.



При выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент.

Khi thực hiện bất kỳ hoạt động nào với danh sách đơn hướng tuyến tính, cần phải đảm bảo định vị một số con trỏ đến phần tử đầu tiên.

**В противном случае часть или весь список будет недоступен.**

Nếu không, một phần hoặc toàn bộ danh sách sẽ không có sẵn.

Операции:

* Вставка элемента (Chèn mục)
* Просмотр (Xem)
* Поиск (Tìm kiếm)
* Удаление элемента (Xóa mục)

type

PElement = ^TypeElement; {указатель на тип элемента}

TypeElement = record {тип элемента списка}

Data: TypeData; {поле данных элемента}

Next: PElement; {поле указателя на следующий элемент}

end;

var

ptrHead: PElement; {указатель на первый элемент списка}

ptrCurrent: PElement; {указатель на текущий элемент}

|  |  |
| --- | --- |
| Процедуры вставки InsFirst\_LineSingleList и Ins\_LineSingleList | |
| Входные параметры | Выходные параметры |
| * Данные для заполнения создаваемого элемента   Dữ liệu để điền vào mục đã tạo   * Указатель на начало списка   Con trỏ đến đầu danh sách | * Указатель на текущий элемент в списке (при необходимости)   Con trỏ tới mục hiện tại trong danh sách (nếu cần)   * Указатель на начало списка указатель текущего элемента (показывает на вновь созданный элемент, при вставке первого элемента указателем на него будет указатель на заголовок списка)   Con trỏ tới đầu con trỏ danh sách đến phần tử hiện tại (trỏ đến phần tử vừa tạo, khi bạn chèn phần tử đầu tiên, con trỏ tới nó sẽ là con trỏ tới tiêu đề danh sách) |

procedure Ins\_LineSingleList(DataElem: TypeData;

var ptrHead, ptrCurrent: PElement);

{Вставка непервого элемента в линейный однонаправленный список}

{справа от элемента, на который указывает ptrCurrent}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

if ptrHead = nil then

begin {список пуст, создаем первый элемент списка}

ptrAddition^.Next := nil;

ptrHead := ptrAddition;

end

else

begin

{список не пуст, вставляем элемент списка справа от

элемента, на который указывает ptrCurrent}

ptrAddition^.Next := ptrCurrent^.Next;

ptrCurrent^.Next := ptrAddition;

end;

ptrCurrent := ptrAddition;

end;

procedure InsFirst\_LineSingleList(DataElem: TypeData;

var ptrHead: PElement);

{Вставка первого элемента в линейный однонаправленный список}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

if ptrHead = nil then

begin {список пуст, создаем первый элемент списка}

ptrAddition^.Next := nil;

end

else

begin {список не пуст - вставляем новый элемент слева

(перед) первым элементом}

ptrAddition^.Next := ptrHead;

end;

ptrHead := ptrAddition;

end;

При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Nếu các con trỏ được xác định lại không chính xác, danh sách có thể bị hỏng hoặc con trỏ đến phần tử đầu tiên có thể bị mất, dẫn đến mất quyền truy cập vào phần hoặc toàn bộ danh sách.

Операция просмотра списка

procedure Scan\_LineSingleList(ptrHead: PElement);

{Просмотр линейного однонаправленного списка}

var ptrAddition: PElement; {вспомогательный указатель}

begin

ptrAddition := ptrHead;

while ptrAddition <> nil do

begin {пока не конец списка}

writeln(ptrAddition^.Data); {Вывод значения элемента}

ptrAddition := ptrAddition^.Next;

end;

end;

**Операция поиска элемента в списке**

|  |  |
| --- | --- |
| Входные параметры | Выходной параметр |
| * Значение, которое должен содержать искомый элемент.   Giá trị mà vật phẩm phải chứa.   * Указатель на список.   Danh sách con trỏ. | * Указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет.   Một con trỏ được đặt trên mục tìm thấy hoặc không thay đổi nếu mục đó không có trong danh sách. |

function Find\_LineSingleList(DataElem: TypeData;

var ptrHead, ptrCurrent: PElement): boolean;

{Поиск элемента в линейном однонаправленном списке}

var ptrAddition:PElement; {Дополнительный указатель}

begin

if (ptrHead <> nil) then

begin {Если существует список}

ptrAddition := ptrHead;

while (ptrAddition <> nil) and (ptrAddition^.Data <> DataElem) do

{пока не конец списка и не найден элемент}

ptrAddition := ptrAddition^.Next;

{Если элемент найден, то результатом работы функции

является true}

if ptrAddition <> nil then

begin

Find\_LineSingleList := true;

ptrCurrent := ptrAddition;

end

else

begin Find\_LineSingleList := false; end;

end

else

begin Find\_LineSingleList:= false; end;

end;

**Операция удаления элемента**

procedure Del\_LineSingleList(var ptrHead, ptrCurrent: PElement);

{Удаление элемента из линейного однонаправленного списка}

var ptrAddition: PElement; {вспомогательный указатель}

begin

if ptrCurrent <> nil then

begin {вх.параметр корректен}

if ptrCurrent = ptrHead then

begin {удаляем первый}

ptrHead := ptrHead^.Next;

dispose(ptrCurrent);

ptrCurrent := ptrHead;

end

else

begin {удаляем непервый}

{устанавливаем вспомогательный указатель на элемент, предшествующий

удаляемому} ptrAddition := ptrHead;

while ptrAddition^.Next <> ptrCurrent do

ptrAddition := ptrAddition^.Next;

{непосредственное удаление элемента}

ptrAddition^.Next := ptrCurrent^.Next;

dispose(ptrCurrent);

ptrCurrent := ptrAddition;

end;

end;

end;

#### **Линейный двунаправленный список**

Danh sách hai chiều tuyến tính

В этом списке любой элемент имеет два указателя, один из которых указывает **на следующий элемент** в списке или является **пустым указателем** у последнего элемента, а второй – на **предыдущий элемент** в списке или является **пустым указателем** у первого элемента.

Trong danh sách này, bất kỳ phần tử nào cũng có hai con trỏ, một trong số đó trỏ đến phần tử tiếp theo trong danh sách hoặc là một con trỏ trống cho phần tử cuối cùng và phần tử thứ hai cho phần tử trước trong danh sách hoặc là một con trỏ trống cho phần tử đầu tiên.



В отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как **благодаря двум указателям** в элементах можно получить **доступ к любому элементу списка** **из любого другого элемента**, осуществляя переходы в прямом или обратном направлении.

Không giống như danh sách đơn hướng, không cần đảm bảo rằng bất kỳ con trỏ nào được định vị chính xác trên phần tử đầu tiên của danh sách, vì nhờ hai con trỏ trong các phần tử, bạn có thể truy cập bất kỳ phần tử nào của danh sách từ bất kỳ phần tử nào khác, thực hiện chuyển tiếp theo hướng tiến hoặc lùi.

**Операции:**

* Вставка элемента
* Просмотр
* Поиск
* Удаление элемента

type

PElement = ^TypeElement; {указатель на тип элемента}

TypeElement = record {тип элемента списка}

Data: TypeData; {поле данных элемента}

Next, {поле указателя на следующий элемент}

Last: PElement; {поле указателя на предыдущий элемент}

end;

var

ptrHead: PElement; {указатель на первый элемент списка}

ptrCurrent: PElement; {указатель на текущий элемент}

Операция вставки реализовывается с помощью двух процедур, аналогичных процедурам вставки для линейного однонаправленного списка:

Hoạt động chèn được thực hiện bằng hai thủ tục tương tự như quy trình chèn cho danh sách đơn hướng tuyến tính:

InsFirst\_LineDubleList и Ins\_LineDubleList

Однако при вставке последующего элемента придется учитывать особенности добавления элемента в конец списка.

Tuy nhiên, khi chèn một phần tử tiếp theo, bạn sẽ phải xem xét các tính năng của việc thêm một phần tử vào cuối danh sách.

procedure Ins\_LineDubleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);

{Вставка непервого элемента в линейный двунаправленный список}

{справа от элемента, на который указывает ptrCurrent}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

if ptrHead = nil then

begin {список пуст - создаем первый элемент списка}

ptrAddition^.Next := nil;

ptrAddition^.Last := nil;

ptrHead := ptrAddition;

end

else

begin

{список не пуст - вставляем элемент списка справа от элемента,}

{на который указывает ptrCurrent}

if ptrCurrent^.Next <> nil then {вставляем не последний}

ptrCurrent^.Next^.Last := ptrAddition;

ptrAddition^.Next := ptrCurrent^.Next;

ptrCurrent^.Next := ptrAddition;

ptrAddition^.Last := ptrCurrent;

end;

ptrCurrent := ptrAddition;

end;

procedure InsFirst\_LineDubleList(DataElem: TypeData; var ptrHead:

PElement);

{Вставка первого элемента в линейный двунаправленный список}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

ptrAddition^.Last := nil;

if ptrHead = nil then

begin {список пуст - создаем первый элемент списка}

ptrAddition^.Next := nil;

end

else

begin

{список не пуст - вставляем новый элемент слева (перед) первым

элементом}

ptrAddition^.Next := ptrHead;

ptrHead^.Last := ptrAddition;

end;

ptrHead := ptrAddition;

end;

**Операция удаления элемента**

procedure Del\_LineDubleList(var ptrHead, ptrCurrent: PElement);

{Удаление элемента из линейного двунаправленного списка}

var ptrAddition: PElement; {вспомогательный указатель}

begin

if ptrCurrent <> nil then

begin {входной параметр корректен}

if ptrCurrent = ptrHead then

begin {удаляем первый}

ptrHead := ptrHead^.Next;

dispose(ptrCurrent);

ptrHead^.Last := nil;

ptrCurrent := ptrHead;

end

else

begin {удаляем непервый}

if ptrCurrent^.Next = nil then

begin {удаляем последний}

ptrCurrent^.Last^.Next := nil;

dispose(ptrCurrent);

ptrCurrent := ptrHead;

end

else

begin {удаляем непоследний и непервый}

ptrAddition := ptrCurrent^.Next;

ptrCurrent^.Last^.Next := ptrCurrent^.Next;

ptrCurrent^.Next^.Last := ptrCurrent^.Last;

dispose(ptrCurrent);

ptrCurrent := ptrAddition;

end;

end;

end;

end;

# *LEK.#5* **ТЕМА 4 - СТРУКТУРЫ ДАННЫХ (ЧАСТЬ 2)**

**ПОНЯТИЕ СТРУКТУРЫ ДАННЫХ**

## **ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ**

### **ЦИКЛИЧЕСКИЕ СПИСКИ**

|  |  |
| --- | --- |
| Линейные списки | Циклические списки |
| * Можно выделить первый и последний элементы.   Bạn có thể chọn các yếu tố đầu tiên và cuối cùng.   * Имеют пустые указатели   Có con trỏ null.   * Обязательно – указатель на начало списка.   Bắt buộc - một con trỏ đến đầu danh sách.   * Алгоритмы вставки и удаления крайних и средних элементов списка отличаются друг от друга.   Các thuật toán để chèn và xóa các phần tử cực trị và trung bình của danh sách khác nhau. | * Нет элементов, содержащих пустые указатели.   Không có phần tử nào chứa con trỏ trống.   * Нельзя выделить крайние элементы.   Bạn không thể chọn các yếu tố cạnh.   * Все элементы являются «средними».   Tất cả các mặt hàng đều trung bình. |

#### **Циклический однонаправленный список**

Danh sách đơn hướng chu kỳ

Похож на линейный однонаправленный список, но его последний элемент содержит указатель, **связывающий его с первым элементом**.

Nó trông giống như một danh sách đơn hướng tuyến tính, nhưng phần tử cuối cùng của nó chứa một con trỏ kết nối nó với phần tử đầu tiên.



Для полного обхода такого списка достаточно иметь указатель на **произвольный элемент.**

Để hoàn toàn bỏ qua một danh sách như vậy, nó là đủ để có một con trỏ đến một phần tử tùy ý.

Полезно выделить некоторый элемент как «первый» путем установки на него специального указателя.

Sẽ rất hữu ích khi làm nổi bật một số phần tử như là First First bằng cách đặt một con trỏ đặc biệt trên nó.

Операции:

* Вставка элемента (Chèn mục)
* Просмотр (Xem)
* Поиск (Tìm kiếm)
* Удаление элемента (Xóa mục)

Объявления данных – те же, что и для линейного однонаправленного списка.

Khai báo dữ liệu giống như đối với danh sách đơn hướng tuyến tính.

type

PElement = ^TypeElement; {указатель на тип элемента}

TypeElement = record {тип элемента списка}

Data: TypeData; {поле данных элемента}

Next, {поле указателя на следующий элемент}

Last: PElement; {поле указателя на предыдущий элемент}

end;

var

ptrHead: PElement; {указатель на первый элемент списка}

ptrCurrent: PElement; {указатель на текущий элемент}

|  |  |
| --- | --- |
| Процедура вставки Ins\_CicleSingleList | |
| Входные параметры | Выходные параметры |
| * Данные для заполнения создаваемого элемента.   Dữ liệu để điền vào mục đã tạo.   * Указатель на начало списка.   Con trỏ đến đầu danh sách.   * Указатель на текущий элемент в списке, после которого осуществляется вставка.   Con trỏ tới mục hiện tại trong danh sách, sau đó thao tác chèn được thực hiện. | * Указатель на начало списка (может измениться).   Con trỏ đến đầu danh sách (có thể thay đổi).   * Указатель текущего элемента (показывает на вновь созданный элемент).   Con trỏ tới mục hiện tại (trỏ đến mục vừa tạo). |

Операция вставки элемента

procedure Ins\_CicleSingleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);

{Вставка элемента в циклический однонаправленный список}

{справа от элемента, на который указывает ptrCurrent}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

if ptrHead = nil then

begin {список пуст – создаем первый элемент списка}

ptrAddition^.Next := ptrAddition; {цикл из 1 элемента}

ptrHead := ptrAddition;

end

else

begin {список не пуст – вставляем элемент списка справа от элемента,}

{на который указывает ptrCurrent}

ptrAddition^.Next := ptrCurrent^.Next;

ptrCurrent^.Next := ptrAddition;

end;

ptrCurrent := ptrAddition;

end;

Порядок следования операторов присваивания процедуры очень важен.

Thứ tự của các toán tử gán thủ tục là rất quan trọng.

При неправильном переопределении указателей возможен **разрыв списка** или **потеря указателя на первый** **элемент**, что приводит к потере доступа к части или всему списку.

Nếu các con trỏ được xác định lại không chính xác, danh sách có thể bị hỏng hoặc con trỏ đến phần tử đầu tiên có thể bị mất, dẫn đến mất quyền truy cập vào phần hoặc toàn bộ danh sách.

**Операция просмотра списка:**

Danh sách hoạt động xem

В отличие от линейного однонаправленного списка здесь признаком окончания просмотра списка будет **возврат к элементу**, выделенному как «первый».

Trái ngược với danh sách đơn hướng tuyến tính, dấu hiệu kết thúc danh sách đang xem ở đây sẽ là sự trở lại với phần tử được đánh dấu là Hồi đầu tiên.

procedure Scan\_CicleSingleList(ptrHead: PElement);

{Просмотр циклического однонаправленного списка}

var ptrAddition: PElement; {вспомогательный указатель}

begin

if ptrHead <> nil do

begin {список не пуст}

ptrAddition := ptrHead;

repeat

writeln(ptrAddition^.Data); {Вывод значения элемента}

ptrAddition := ptrAddition^.Next;

until ptrAddition = ptrHead;

end;

end;

**Операция поиска элемента**

Hoạt động tìm kiếm mục

Последовательный просмотр всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не достигнут «первый» элемент списка **(фиксируется отсутствие искомого элемента в списке, функция принимает значение false).**

Xem tuần tự tất cả các phần tử của danh sách cho đến khi phần tử hiện tại chứa giá trị được chỉ định hoặc cho đến khi phần tử "đầu tiên" của danh sách đạt được (sự vắng mặt của phần tử mong muốn trong danh sách được cố định, hàm sẽ nhận giá trị sai).

|  |  |
| --- | --- |
| Входные параметры | Выходной параметр |
| * Значение, которое должен содержать искомый элемент.   Giá trị mà vật phẩm phải chứa.   * Указатель на список.   Con trỏ tới danh sách. | * Указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет   Một con trỏ được cài đặt trên phần tử tìm thấy hoặc không thay đổi nếu phần tử không có trong danh sách |

function Find\_CicleSingleList(DataElem: TypeData;

var ptrHead, ptrCurrent: PElement): boolean;

{Поиск в циклическом однонаправленном списке}

var ptrAddition: PElement; {вспомогательный указатель}

begin

if ptrHead <> nil do

begin {список не пуст}

ptrAddition := ptrHead;

repeat

ptrAddition := ptrAddition^.Next;

until (ptrAddition = ptrHead) or {прошли весь список}

(ptrAddition^.Data = DataElem) {элемент найден}

if ptrAddition^.Data = DataElem then

begin

Find\_CicleSingleList := true;

ptrCurrent := ptrAddition;

end

else

begin Find\_CicleSingleList := false; end;

end

else

begin

Find\_CicleSingleList := false;

end;

end;

**Операция удаления элемента:**

Thao tác xóa mục:

Удаляет элемент, на который установлен **указатель текущего элемента**

Xóa mục mà con trỏ của mục hiện tại được đặt thành.

После удаления указатель текущего элемента устанавливается на элемент списка, **следующий за удаляемым**

Sau khi xóa, con trỏ của mục hiện tại được đặt thành mục danh sách theo sau mục sẽ bị xóa.

В случае **удаления «первого» элемента** необходимо соответствующий указатель **переместить на следующий элемент.**

Nếu bạn xóa phần tử "đầu tiên", bạn phải di chuyển con trỏ tương ứng sang phần tử tiếp theo.

procedure Del\_CicleSingleList(var ptrHead, ptrCurrent: PElement);

{Удаление элемента из циклического однонаправленного списка}

var ptrAddition: PElement; {дополнительный указатель}

begin

if ptrCurrent <> nil then

begin {входной параметр корректен}

if ptrHead^.Next <> ptrHead then

begin {Если удаляемый элемент не единственный в списке}

{устанавливаем вспомогательный указатель на элемент,

предшествующий удаляемому}

ptrAddition := ptrHead;

while ptrAddition^.Next <> ptrCurrent do

ptrAddition := ptrAddition^.Next;

{непосредственное удаление элемента}

ptrAddition^.Next := ptrCurrent^.Next;

if ptrHead = ptrCurrent then {удаляем первый}

ptrHead := ptrCurrent^.Next;

dispose(ptrCurrent);

ptrCurrent := ptrAddition^.Next;

end

else

begin

ptrHead:=nil;

dispose(ptrCurrent);

ptrCurrent:=nil;

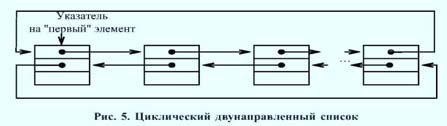
end;

end;

end;

#### **Циклический двунаправленный список**

Любой элемент имеет два указателя, один из которых указывает на **следующий** элемент в списке, а второй – на **предыдущий** элемент.



Объявления данных – те же, что и для

линейного двунаправленного списка

**Операции**:

* Вставка элемента
* Просмотр
* Поиск
* Удаление элемента

type

PElement = ^TypeElement; {указатель на тип элемента}

TypeElement = record {тип элемента списка}

Data: TypeData; {поле данных элемента}

Next, {поле указателя на следующий элемент}

Last: PElement; {поле указателя на предыдущий элемент}

end;

var

ptrHead: PElement; {указатель на первый элемент списка}

ptrCurrent: PElement; {указатель на текущий элемент}

**Операция вставки элемента** Ins\_CicleDubleList

|  |  |
| --- | --- |
| Входные параметры | Выходные параметры |
| * Данные для заполнения создаваемого элемента   Dữ liệu để điền vào mục đã tạo. Указатель на начало списка  Con trỏ đến đầu danh sách.   * Указатель на текущий элемент в списке, после которого осуществляется вставка.   Con trỏ tới mục hiện tại trong danh sách, sau đó thao tác chèn được thực hiện. | * Указатель на начало списка (может измениться).   Con trỏ đến đầu danh sách (có thể thay đổi).   * Указатель текущего элемента (показывает на вновь созданный элемент).   Con trỏ tới mục hiện tại (trỏ đến mục vừa tạo). |

procedure Ins\_CicleDubleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);

{Вставка элемента в циклический двунаправленный список

справа от элемента, на который указывает ptrCurrent}

var ptrAddition: PElement; {вспомогательный указатель}

begin

New(ptrAddition);

ptrAddition^.Data := DataElem;

if ptrHead = nil then

begin {список пуст – создаем первый элемент списка}

ptrAddition^.Next := ptrAddition; {петля из 1 элемента}

ptrAddition^.Last := ptrAddition;

ptrHead := ptrAddition;

end

else

begin {список не пуст – вставляем элемент списка справа от элемента,}

{на который указывает ptrCurrent}

ptrAddition^.Next := ptrCurrent^.Next;

ptrCurrent^.Next := ptrAddition;

ptrAddition^.Last := ptrCurrent;

ptrAddition^.Next^.Last := ptrAddition;

end;

ptrCurrent := ptrAddition;

end;

При неправильном переопределении указателей возможен **разрыв списка** или **потеря указателя на первый элемент**, что приводит к потере доступа к части или всему списку.

Nếu các con trỏ được xác định lại không chính xác, danh sách có thể bị hỏng hoặc con trỏ đến phần tử đầu tiên có thể bị mất, dẫn đến mất quyền truy cập vào phần hoặc toàn bộ danh sách.

**Операция просмотра** и **операция поиска** для циклического двунаправленного списка реализуются абсолютно аналогично соответствующи процедурам для циклического однонаправленного списка.

Hoạt động duyệt và hoạt động tìm kiếm cho danh sách hai chiều tròn được thực hiện theo cách chính xác giống như các quy trình tương ứng cho danh sách đơn hướng vòng tròn.

Просматривать и искать здесь можно в обоих направлениях.

Bạn có thể duyệt và tìm kiếm ở đây theo cả hai hướng.

procedure Del\_CicleDubleList(var ptrHead, ptrCurrent: PElement);

{Удаление элемента из циклического двунаправленного списка}

var ptrAddition: PElement; {дополнительный указатель}

begin

if ptrCurrent <> nil then

begin {входной параметр корректен}

if ptrHead^.Next <> ptrHead then

begin {Если удаляемый элемент, не единственный в списке}

ptrAddition := ptrCurrent^.Next;

ptrCurrent^.Last^.Next := ptrCurrent^.Next;

ptrCurrent^.Next^.Last := ptrCurrent^.Last;

if ptrHead = ptrCurrent then {удаляем первый}

ptrHead := ptrCurrent^.Next;

dispose(ptrCurrent);

ptrCurrent := ptrAddition;

end

else

begin

ptrHead:=nil;

dispose(ptrCurrent);

ptrCurrent:=nil;

end;

end;

end;

Использование двух указателей позволяет **ускорит операции**, связанные с передвижением по списку за счет двунаправленности этого движения.

Việc sử dụng hai con trỏ cho phép bạn tăng tốc các hoạt động liên quan đến việc di chuyển xung quanh danh sách do tính hai chiều của chuyển động này.

### **РАЗРЕЖЕННЫЕ МАТРИЦЫ**

Разреженная матрица – **двумерный массив**, большинство элементов которого **равны** между собой, так что хранить в памяти достаточно лишь небольшое число значений, **отличных от основного** (фонового) значения остальных элементов.

Ma trận thưa thớt là một mảng hai chiều, hầu hết các phần tử của chúng đều bằng nhau, do đó chỉ một số lượng nhỏ các giá trị khác với giá trị chính (nền) của các phần tử còn lại là đủ để lưu trữ trong bộ nhớ.

Разреженные матрицы

Ma trận thưa thớt

* Матрицы, в которых местоположения элементов со значениями, отличными от фонового, могут быть математически описаны.

Ma trận trong đó vị trí của các phần tử có giá trị khác với nền có thể được mô tả bằng toán học.

* Матрицы со случайным расположением элементов.

Ma trận với sự sắp xếp ngẫu nhiên của các yếu tố.

#### **Матрицы с математическим описанием местоположения элементов**

В расположении элементов есть какая-либо закономерность.

Có một số mô hình trong việc sắp xếp các yếu tố.

Элементы, значения которых **являются фоновыми**, называют нулевыми. – Các phần tử có giá trị là nền được gọi là null.

Элементы, значения которых **отличны от фонового**, называют ненулевыми. – Các phần tử có giá trị khác với nền được gọi là khác không.

(фоновое значение не всегда равно нулю - giá trị nền không phải luôn luôn bằng không).

Ненулевые значения хранятся, как правило, **в одномерном массиве (векторе),** а связь между местоположением в разреженной матрице и в новом, одномерном, описывается математически с помощью формулы, преобразующей индексы матрицы в индексы вектора.

Các giá trị khác không được lưu trữ, theo quy tắc, trong một mảng một chiều (vectơ) và mối quan hệ giữa vị trí trong ma trận thưa thớt và trong một mảng mới, một chiều được mô tả bằng toán học bằng cách sử dụng một công thức chuyển đổi các chỉ số ma trận thành các chỉ số vectơ.

Для работы разрабатываются функции (Chức năng được phát triển cho công việc)

* Для преобразования индексов матрицы в индекс вектора.

Để chuyển đổi các chỉ số ma trận thành chỉ mục vector.

* Для получения значения элемента матрицы из ее упакованного представления по двум индексам (строка, столбец).

Để có được giá trị của một phần tử ma trận từ biểu diễn được đóng gói của nó bằng hai chỉ số (hàng, cột).

* Для записи значения элемента матрицы в ее упакованное представление по вум индексам.

Để ghi giá trị của một phần tử ma trận vào biểu diễn đóng gói của nó bằng các chỉ số wum.

**Пример:**

Имеется двумерная разреженная матрица, в которой все ненулевые элементы расположены в шахматном порядке, начиная со второго элемента.

Có một ma trận thưa thớt hai chiều trong đó tất cả các phần tử khác không được đặt so le, bắt đầu từ phần tử thứ hai.

Формула вычисления индекса элемента в линейном представлении:

Công thức tính chỉ số của một phần tử trong biểu diễn tuyến tính:

Где:

**l** – индекс в линейном представлении;

**x, у** – индексы соответственно строки и столбца в двумерном представлении;

**Xm** – количество элементов в строке исходной матрицы

#### **Матрицы со случайным расположением элементов**

Матрицы, у которых местоположение элементов со значениями, отличными от фонового, не могут быть математически описано – в их расположении **нет какой-либо закономерности.**

Ma trận trong đó vị trí của các phần tử có giá trị khác với nền không thể được mô tả bằng toán học - không có mẫu nào ở vị trí của chúng.

Пусть в матрице A размерности 5x7 из 35 элементов только 10 отличны от нуля.

Giả sử rằng trong một ma trận A có kích thước 5x7 trong số 35 phần tử chỉ có 10 phần tử là khác không.

|  |  |
| --- | --- |
|  |  |

Способ хранения – запоминание ненулевых элементов в одномерном массиве записей – **последовательное представление разреженных матриц.**

Phương pháp lưu trữ - lưu trữ các phần tử khác 0 trong một mảng các bản ghi - biểu diễn tuần tự các ma trận thưa thớt.

Для ускорения поиска элементы матрицы обязательно запоминаются в порядке **возрастания номеров строк.**

Để tăng tốc tìm kiếm, các phần tử ma trận nhất thiết phải được lưu trữ theo thứ tự tăng dần của số dòng.

Включение и исключение новых элементов матрицы вызывает необходимость перемещения большого числа существующих элементов.

Việc bao gồm và loại trừ các yếu tố ma trận mới làm cho cần phải di chuyển một số lượng lớn các yếu tố hiện có.

Если включение новых элементов и их исключение осуществляется часто, то можно использовать метод связанных структур – переводит **статическую структуру** матрицы в **динамическую**, реализуемую в виде циклических списков.

Nếu việc bao gồm các phần tử mới và loại trừ chúng thường được thực hiện, thì bạn có thể sử dụng phương pháp của các cấu trúc liên quan - chuyển cấu trúc ma trận tĩnh thành một cấu trúc động, được thực hiện dưới dạng danh sách tuần hoàn.

type

PElement = ^TypeElement; {указатель на тип элемента}

TypeElement = record {тип элемента списка}

Left: PElement; {указатель на предыдущий элемент в строке}

Up: PElement; {указатель на предыдущий элемент в столбце}

Value: TypeData; {значение элемента матрицы}

Row: integer; {индекс строки матрицы}

Colum: integer; {индекс столбца матрицы}

end;



### **СТЕК**

Стек – структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала

Используется принцип «последним пришел – первым вышел» (LIFO: Last Input – First Output)

Стек можно реализовывать

* Как статическую структуру данных в виде одномерного массива
* Как динамическую структуру – в виде линейного списка

# *LEK.#6* **ТЕМА 5 - АЛГОРИТМЫ ПОИСКА**

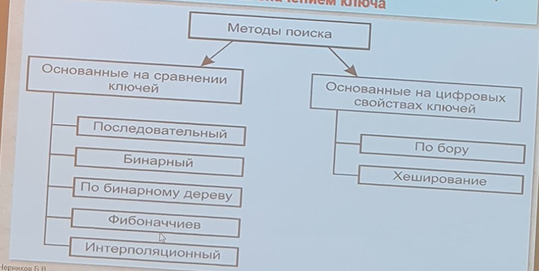
**Общее знание**

**Поиск** – процесс **отыскания информации во множестве данных** (обычно предсталяющих собой записи) путем просмотра специального поля в каждой записи, называемого ключом.

**Tìm kiếm** là quá trình tìm kiếm thông tin trong một tập hợp dữ liệu (thường đại diện cho các bản ghi) bằng cách xem một trường đặc biệt trong mỗi bản ghi được gọi là **khóa**.

**Целю поиска** является отыскание записи (если она есть) с данным значением ключа. – Mục đích của tìm kiếm là tìm một bản ghi (nếu có) với một **giá trị khóa đã cho.**

Методы поиска:

****

**Предметы (объекты), составляющие множество**, называются его элементами.

Элемент множества будет называться ключом, и обозначаться латинской буквой **ki**, с индексом, указывающим номер элемента.

Пусть задано множество ключей {k1, k2, k3, ..., kn}

**Массив фиксированной длиный:** (Mảng có độ dài cố định) – Это множество данных, в котором происходит поиск.

A: array[1..n] of ItemType;

## **ПОСЛЕДОВАТЕЛЬНЫЙ (ЛИНЕЙНЫЙ) ПОИСК**

(Tìm kiếm tuần tự (tuyến tính))

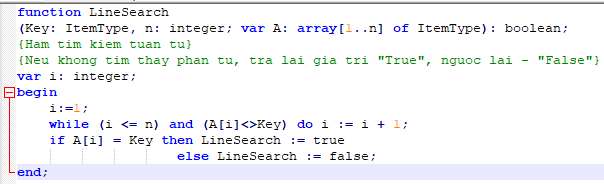
Множество элементов просматривается последовательно в порядке, гарантирующем просмотр их. - Nhiều yếu tố được quét liên tục để đảm bảo rằng chúng được xem đầy đủ.

Алгоритм можно применять тогда, когда нет никакой дополнительной информации о расположении данных в рассматриваемой последовательности. - Thuật toán có thể được áp dụng khi không có thông tin bổ sung về vị trí của dữ liệu trong chuỗi được đề cập.

Tóm tắt các bước:

1. Đặt i = 1.
2. k = ki => thuật toán kết thúc thành công.
3. Tăng i thêm một đơn vị.
4. Nếu i <= N, quay trở lại bước 2. Nếu không, thuật toán kết thúc mà không tìm ra kết quả.

**Ex:**



В среднем требует n/2 итераций цикла - Trung bình, yêu cầu lặp lại n / 2 vòng lặp

Это означает временную сложность алгоритма поиска, пропорциональную O(n) - Điều này có nghĩa là độ phức tạp thời gian của thuật toán tìm kiếm, tỷ lệ với O (n)

Никаких ограничений на порядок элементов в массиве алгоритм не накладывает

Thuật toán không áp đặt bất kỳ hạn chế nào đối với thứ tự các phần tử trong mảng

При наличии в массиве нескольких элементов со значением Key алгоритм находит только первый из них (с наименьшим индексом)

Nếu có một số phần tử trong mảng có giá trị Khóa, thuật toán chỉ tìm thấy phần tử đầu tiên trong số chúng (có chỉ số thấp nhất)

Модификация поиска - оптимизация цикла: (Sửa đổi tìm kiếm - Tối ưu hóa chu kỳ)

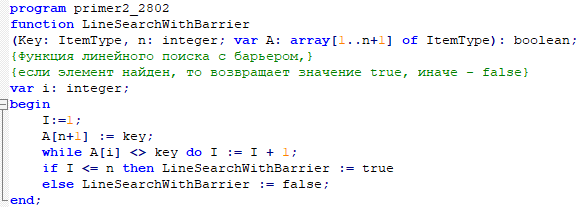
В цикле while происводятся два сравнения: (i<=n) и (A[i] <> Key).

Hai phép so sánh được thực hiện trong vòng lặp while: (i <= n) và (A [i] <> Key).

Избавимся от первого, введя в массив так называемый " барьер", положив

A[n+1] := Key

Chúng tôi loại bỏ cái đầu tiên bằng cách đưa cái gọi là "rào cản" phạm vi vào mảng, đặt A [n + 1]: = Key



Работает быстрее, но временная сложность алгоритма остается такой же - O(n)

## **БИНАРНЫЙ ПОИСК**

Этот алгоритм поиска предполагает, что множество хранится, как некоторая упорядоченная (например, по возрастанию) последовательность элементов, к которым можно получить прямой доступ посредством индекса.

Thuật toán tìm kiếm này giả định rằng tập hợp được lưu trữ dưới dạng một số chuỗi các phần tử được sắp xếp (ví dụ: tăng dần) có thể được truy cập trực tiếp thông qua chỉ mục.

Массив отсортирован - каждый последующий ключ больше (во всяком случае - не меньше) предыдушего:

Mảng được sắp xếp - mỗi khóa tiếp theo lớn hơn (trong mọi trường hợp, không ít hơn) của khóa trước:



**Суть метода:** отыскиваемый ключ сравнивается с центральным элементом множества, если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае - в правом.

Bản chất của phương thức: khóa tìm kiếm được so sánh với phần tử trung tâm của tập hợp, nếu nó nhỏ hơn trung tâm, thì tìm kiếm tiếp tục trong tập hợp con bên trái, nếu không - ở bên phải.

Медианой последовательности из n элементов считается элемент, значение которого меньше (или равно) половины n элементов и больше (или равно) другой половины.

Trung vị của một chuỗi gồm n phần tử là một phần tử có giá trị nhỏ hơn (hoặc bằng) một nửa của n phần tử và lớn hơn (hoặc bằng) nửa còn lại.

В алгоритме Хоара для нахождения медианы используется операция разделения с L = 1, R = n и с a[k], выбранным в качестве разделяющего значения x.

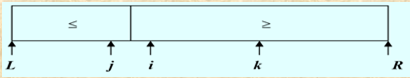
Trong thuật toán Hoare, thao tác tách với L = 1, R = n và với [k], được chọn làm giá trị tách của x, được sử dụng để tìm trung vị.

Получаются значения индексов i и j, такие, что:

Các giá trị của các chỉ số i và j thu được, sao cho:

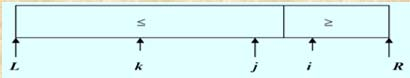
1. Разделяющее значение x было слишком мало – в результате граница между двумя частями **ниже** искомого значения k. Процесс разделения следует повторить для элементов a[i],… , a[R]

Giá trị chia của x quá nhỏ - kết quả là ranh giới giữa hai phần thấp hơn giá trị mong muốn của k. Quá trình phân tách phải được lặp lại cho các phần tử a [i], ..., a [R].



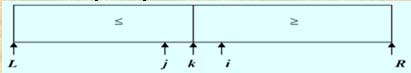
2. Выбранная граница x была слишком велика – граница **выше** искомого значения k. Операцию разбиения следует повторить на подмассиве a[L],…, a[j]

Ranh giới được chọn x quá lớn - ranh giới cao hơn giá trị tìm kiếm của k. Thao tác chia tách phải được lặp lại trên phân đoạn con [L], ..., a [j].



3. Значение k лежит в интервале j < k < i – элемент a[k] разделяет массив в заданной пропорции

Giá trị của k nằm trong khoảng j <k <i - phần tử a [k] chia mảng theo tỷ lệ đã cho



**Пример 1**. Во множестве элементов отыскать ключ равный **653**.

Поиск ключа К = 653 осуществляется за четыре шага:

**[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]**

061 087 154 170 275 426 503 509 **[512 612 653 677 703 765 897 908]**

061 087 154 170 275 426 503 509 **[512 612 653]** 677 703 765 897 908

061 087 154 170 275 426 503 509 512 612 **[653]** 677 703 765 897 908

**Пример 2**. Дано упорядоченное множество элементов.

**{7,8,12,16,18,20,30,38,49,50,54,60,61,69,75,79,80,81,95,101,123,198}**

Найти во множестве ключ K = 61

**Шаг 1.** Nэл-та = [n / 2]+1 = [22 / 2] + 1 = 12

**{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}**

**K∨k12 -> 61 > 60 ->** **Дальнейший поиск в правом подмножестве.**

**Шаг 2.** Nэл-та эл-та = [n / 2] +1 = [10 / 2] +1 = 6

**{61, 69, 75, 79, 80, 81, 95, 101, 123, 198}**

**K∨k18 -> 61 < 81** **-> Дальнейший поиск в левом подмножестве.**

**Шаг 3.** Nэл-та эл-та = [n/2] +1 = [5/2] +1 = 3

**{61, 69, 75, 79, 80}**

**K∨k15 -> 61 < 75** **-> Дальнейший поиск в левом подмножестве.**

**Шаг 4.** Nэл-та эл-та = [n/2] +1 = [2/2] +1 = 2

**{61, 69}**

**K∨k14 -> 61 < 69** **-> Дальнейший поиск в левом подмножестве.**

**Шаг 5.** **{61}**

**K∨k13 -> 61=61** **-> Искомый ключ найден под номером 13.**

Имеем массив **А = {1, 2, 2, 3, 5}**

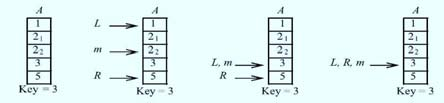
Сначала область поиска – часть массива (L, R), где L = 1, а R = n

Индекс среднего элемента: m := (L+R) div 2

Если Key > Am **->** Key находится в одном из элементов с индексами

от L + m до R, следовательно, можно присвоить L := m + 1

В противном случае можно положить R := m



На каждом шаге область поиска будет **сокращаться вдвое**

Ở mỗi bước, khu vực tìm kiếm sẽ được giảm một nửa.

Как только L станет равно R, т. е. область поиска **сократится** **до** **одного** **элемента**, можно будет проверить этот элемент на равенство искомому и сделать вывод о результате поиска.

Ngay khi L trở thành bằng R, tức là, khu vực tìm kiếm bị giảm xuống một yếu tố, có thể kiểm tra yếu tố này để tìm sự bình đẳng với yếu tố tìm kiếm và đưa ra kết luận về kết quả tìm kiếm.

function BinarySearch

(Key: ItemType, n: integer; var A: array[1..n] of ItemType): boolean;

{Функция двоичного поиска,}

{если элемент найден, то возвращает значение true, иначе – false}

var L, m, R: integer;

begin

L := 1; R := n;

while (L <> R) do

begin

m := (L+R) div 2;

if Key > A[m] then L := m+1 else R := m;

end;

if A[L]= Key then BinarySearch := true

else BinarySearch := false;

end;

Область поиска на каждом шаге сокращается вдвое, а это означает временную сложность алгоритма, пропорциональную **O(log n).**

Vùng tìm kiếm ở mỗi bước được giảm một nửa, có nghĩa là độ phức tạp thời gian của thuật toán, tỷ lệ với O (log n).

## **ФИБОНАЧЧИЕВ ПОИСК**

Анализируются элементы, находящиеся в позициях, равных **числам Фибоначчи**: {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …}

Các phần tử ở các vị trí bằng với số Fibonacci được phân tích: {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...}.

Поиск продолжается до тех пор, пока не будет найден **интервал** **между двумя позициями** с расположением **отыскиваемого ключа.**

Việc tìm kiếm tiếp tục cho đến khi một khoảng được tìm thấy giữa hai vị trí với vị trí của khóa được tìm kiếm.

Фибоначчиев поиск предназначается для поиска аргумента K среди расположенных отсортированных ключей K1 < K2 < … < Kn

Truy xuất Fibonacci nhằm tìm kiếm đối số K trong số các khóa được sắp xếp nằm K1 <K2 <... <Kn.

Предположим, что n+1 есть число Фибоначчи Fk+1

Giả sử n + 1 là số Fibre Fk + 1

1. **Начальная установка**. Установить i:=Fk, p:=Fk-1 , q:=Fk-2. (p и q обозначают последовательные числа Фибоначчи)

Cài đặt ban đầu. Đặt i: = Fk, p: = Fk-1, q: = Fk-2. (p và q biểu thị các số Fibonacci liên tiếp)

1. **Сравнение**. Если K < Ki, то перейти **на шаг 3**; если K > Ki, то перейти **на шаг 4**; если K = Ki, алгоритм заканчивается удачно.

So sánh. Nếu K <Ki, sau đó chuyển sang bước 3; nếu K> Ki, sau đó chuyển sang bước 4; nếu K = Ki, thuật toán kết thúc thành công.

1. **Уменьшение** i. Если q = 0, алгоритм заканчивается неудачно. Если q ≠ 0, то установить i := iq, заменить (p, q) на (q, pq) и вернуться на **шаг 2**.

Giảm i. Nếu q = 0, thuật toán thất bại. Nếu q ≠ 0, sau đó đặt i: = iq, thay thế (p, q) bằng (q, pq) và quay lại bước 2.

1. **Увеличение** i. Если p = 1, алгоритм заканчивается неудачно. Если p ≠ 1, установить i := i + q, p := pq, q := qp и вернуться **на шаг 2.**

Tăng i. Nếu p = 1, thuật toán thất bại. Nếu p ≠ 1, đặt i: = i + q, p: = pq, q: = qp và quay lại bước 2.

**Пример**. Дано исходное множество ключей

**{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}**

Пусть отыскиваемый ключ равен 42 (K = 42).

Đặt khóa tìm kiếm là 42 (K = 42).

Последовательное сравнение отыскиваемого ключа будет проводиться с элементами исходного множества, расположенных в позициях, равных числам Фибоначчи: **{1, 2, 3, 5, 8, 13, 21,…}**

Một so sánh tuần tự của khóa được tìm kiếm sẽ được thực hiện với các phần tử của tập hợp gốc nằm ở các vị trí bằng với các số Fibonacci: {1, 2, 3, 5, 8, 13, 21, ...}

**Шаг 1.** K∨k1 ->42>3 -> отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи. (khóa tìm kiếm được so sánh với khóa ở vị trí bằng với số Fibonacci)

**Шаг 2.** K∨k2 ->42>5 -> сравнение продолжается с ключом, стоящимVв позиции, равной следующему числу Фибоначчи. (sự so sánh tiếp tục với một khóa nằm ở vị trí V bằng với số Fibonacci tiếp theo)

**Шаг 3.** K∨k3 ->42>8 -> сравнение продолжается.

**Шаг 4.** K∨k5 ->42->11 -> сравнение продолжается.

**Шаг 5.** K∨k8 ->42>19 -> сравнение продолжается.

**Шаг 6.** K∨k13 ->42>35 -> сравнение продолжается.

**Шаг 7.** K∨k18 ->42<52 -> найден интервал, в котором находится отыскиваемый ключ, т.е отыскиваемый ключ может находится в исходном множестве **между 13 и 18 позициями.** (đã tìm thấy khoảng thời gian mà khóa tìm kiếm được đặt, tức là, khóa được tìm kiếm có thể nằm trong bộ ban đầu giữa 13 và 18 vị trí)

т.е. **{35, 37, 42, 45, 48, 52}**

В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

Trong khoảng thời gian tìm thấy, việc tìm kiếm lại được tiến hành ở các vị trí bằng với số Fibonacci.

## **ИНТЕРПОЛЯЦИОННЫЙ ПОИСК**

Исходное множество – **упорядочено по возрастанию** весов.

Các tập ban đầu được sắp xếp bằng cách tăng trọng lượng.

Первоначальное сравнение осуществляется на расстоянии шага d, который определяется по формуле.

Việc so sánh ban đầu được thực hiện ở khoảng cách bước d, được xác định theo công thức.



**i –** номер первого элемента; số phần tử đầu tiên;

**j –** номер последнего элемента; số phần tử cuối cùng;

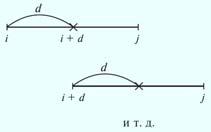
**K –** ключ;

**Ki, Kj** − значения ключей в **i** и **j** позициях; các giá trị chính ở vị trí i và j;

**[ ]** – целая часть от числа - Là phần nguyên của số

Идея: шаг поиска d меняется по формуле после каждого этапа

Ý tưởng: bước tìm kiếm d thay đổi theo công thức sau mỗi giai đoạn

Алгоритм **заканчивает** работу при d = 0, при этом **анализируются соседние элементы**, после чего принимается окончательное решение о результатах поиска.

Thuật toán kết thúc tại d = 0, trong khi các yếu tố lân cận được phân tích, sau đó quyết định cuối cùng về kết quả tìm kiếm được đưa ra.

Метод прекрасно работает, если исходное множество – **арифметическая прогрессия** или множество, приближенное к ней.

Phương thức này hoạt động rất tốt nếu tập gốc là một tiến trình số học hoặc một tập gần với nó.

**Пример 1**. Дано множество ключей:

**{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}**

Пусть искомый ключ K = 70.

**Шаг 1**. Определим шаг d для исходного множества (i = 1; j = 18):

d = [(18 – 1)·(70 – 2) / (76 – 2)] = 15

Сравниваем ключ, стоящий под номером 16 с искомым ключом:

**k16 ∨ K -> 70 = 70** -> **ключ найден**

**Пример 2**. Дано множество ключей:

**{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102}**

Требуется отыскать ключ К = 90.

**Шаг 1.** Определим шаг d для исходного множества (i = 1 ; j = 17):

d = [(17 – 1)·(90 – 4) / (102 – 4)] = 14

Сравниваем ключ, стоящий под номером 15 с ключом:

**k15 ∨ K -> 95 > 90** -> следовательно, сужается область поиска

**{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95}**

**Шаг 2**. Определим шаг d для множества ключей (i = 1 ; j = 15):

d = [(15 – 1) ·(90 – 4) / (95 – 4)] = 13

Сравниваем ключ, стоящий под порядковым номером 14 с отыскиваемым ключом:

**k14 ∨ K ->k14 = K -> 90 = 90** -> **ключ найден**

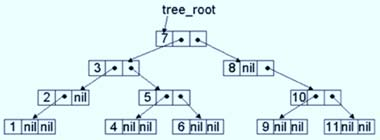
## **ПОИСК ПО БИНАРНОМУ ДЕРЕВУ**

Дерево двоичного поиска для множества чисел S – это **размеченное бинарное дерево**, каждой **вершине** которого сопоставлено **число** из множества S, a все пометки удовлетворяют правилу: если больше – направо, если меньше – налево

Cây tìm kiếm nhị phân cho một tập hợp số S là cây nhị phân được đánh dấu, mỗi đỉnh được liên kết với một số từ tập S và tất cả các dấu thỏa mãn quy tắc: nếu nhiều hơn - bên phải, nếu ít hơn - bên trái

**Пример**. Для набора чисел **{7, 3, 5, 2, 8, 1, 6, 10, 9, 4, 11}** получится бинарное дерево.

Một ví dụ. Đối với một tập hợp các số {7, 3, 5, 2, 8, 1, 6, 10, 9, 4, 11}, chúng ta có được một cây nhị phân.



Чтобы правильно учесть **повторения** чисел, можно ввести **дополнительное поле**, которое будет хранить количество вхождений для каждого числа.

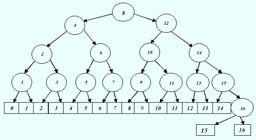
Để tính toán chính xác cho việc lặp lại các số, bạn có thể nhập một trường bổ sung sẽ lưu trữ số lần xuất hiện cho mỗi số.

Бинарное дерево для бинарного поиска среди n записей можно построить следующим образом: при n = 0 дерево сводится **к узлу 0**.

Cây nhị phân để tìm kiếm nhị phân trong số n bản ghi có thể được xây dựng như sau: for n = 0, cây giảm xuống nút 0.

В противном случае корневой узел – [n / 2], левое поддерево соответствует бинарному дереву с [n / 2] – 1 узлами, а правое – дереву с [n / 2] узлами и числами в узлах, увеличенными на [n / 2]

Mặt khác, nút gốc là [n / 2], cây con bên trái tương ứng với cây nhị phân có [n / 2] - 1 nút và cây con bên phải tương ứng với cây có các nút [n / 2] và số trong các nút tăng thêm [n / 2]



*Бинарное дерево для n=16*

**Число порождений отдельного узла** (число поддеревьев данного корня) − его степень.

Số lượng sinh sản của một nút (số lượng cây con của một gốc nhất định) - mức độ của nó.

**Узел с нулевой степенью** называют листом или концевым узлом

Một nút có độ không được gọi là nút lá hoặc nút cuối.

**Максимальное значение степени** всех узлов дерева – степень дерева

Giá trị tối đa của mức độ của tất cả các nút của cây là mức độ của cây.

**Алгоритм поиска по бинарному дереву**: вначале аргумент поиска сравнивается с ключом, находящимся в корне. Если аргумент **совпадает** с ключом, поиск заканчивается. Если не совпадает, то в случае, когда аргумент **меньше ключа**, поиск продолжается в левом поддереве, когда больше ключа – в правом поддереве.

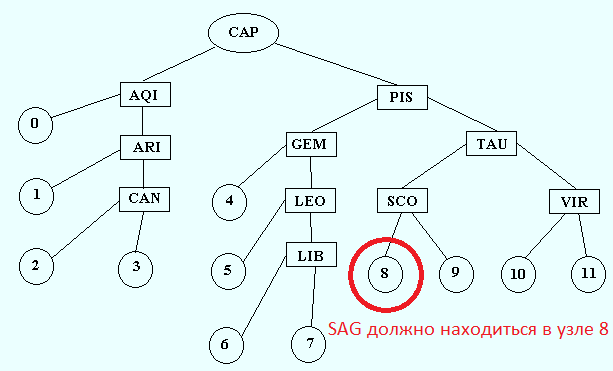
Thuật toán tìm kiếm cây nhị phân: đầu tiên, đối số tìm kiếm được so sánh với khóa nằm trong thư mục gốc. Nếu đối số khớp với khóa, tìm kiếm kết thúc. Nếu nó không khớp, thì trong trường hợp khi đối số nhỏ hơn khóa, tìm kiếm tiếp tục ở cây con bên trái, khi nhiều hơn khóa - trong cây con bên phải.

Увеличив уровень на 1, повторяют сравнение, считая текущий узел корнем.

Tăng cấp độ 1, lặp lại so sánh, coi nút hiện tại là gốc.

Дано бинарное дерево поиска. Требуется отыскать ключ **SAG**

Cho một cây tìm kiếm nhị phânTìm khóa SAG cần thiết.



1. По первой букве латинского алфавита – название **SAG** больше чем **САР** -> поиск будем вести в правой ветви.

Bằng chữ cái đầu tiên của bảng chữ cái Latinh - tên SAG nhiều hơn so với chữ viết hoa -> chúng tôi sẽ tìm kiếm ở nhánh bên phải.

1. **SAG** больше **PIS** -> вправо.

SAG lớn hơn PIS -> ở bên phải.

1. **SAG** меньше **TAU** -> влево.

SAG ít hơn TAU -> trái.

1. **SAG** меньше **SCO** -> узел 8.

SAG nhỏ hơn SCO -> nút 8.

**Пример.** Дано исходное множество ключей

**{2, 4, 5, 6, 7, 9, 12, 14, 18, 21, 24, 25, 27, 30, 32, 33, 34, 37, 39}**

Множество ключей должно быть **упорядочено по возрастанию.**

Nhiều khóa nên được sắp xếp theo thứ tự tăng dần.

Переходим от линейного списка к построению бинарного дерева поиска, где **корень дерева** – центральный элемент множества:

Chúng ta chuyển từ danh sách tuyến tính sang việc xây dựng cây tìm kiếm nhị phân, trong đó gốc của cây là thành phần trung tâm của tập hợp:

Nц = [ N / 2 ] + 1, где N – количество элементов множества

Nц = [N / 2] + 1, trong đó N là số phần tử trong tập hợp

Вершиной по **левой** ветке является **центральный элемент левого** **подмножества**, а **правой** – **правого** подмножества, и т. д.

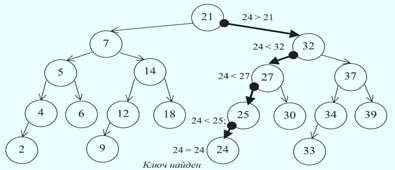
Phần trên cùng của nhánh bên trái là phần tử trung tâm của tập hợp con bên trái và bên phải là tập hợp con bên phải, v.v.

Отыскиваемый ключ K = 24.

Khóa tìm kiếm là K = 24.

Исходное множество имеет 19 элементов, Nц = [ 19 / 2 ] + 1 = 10

Tập ban đầu có 19 phần tử, Nц = [19/2] + 1 = 10



### **ПРЕОБРАЗОВАНИЕ ПРОИЗВОЛЬНОГО ДЕРЕВА В БИНАРНОЕ**

CHUYỂN ĐỔI MỘT CÂY TRỒNG

Упорядоченные деревья **степени 2** – бинарные деревья.

Cây theo thứ tự cấp 2 là cây nhị phân.

Бинарное дерево состоит из конечного множества элементов (узлов), каждый из которых либо **пуст**, либо состоит из **корня** (узла), **связанного с двумя** различными **бинарными деревьями**, называемыми **левым** и **правым поддеревом корня.**

Cây nhị phân bao gồm một tập hợp các phần tử (nút) hữu hạn, mỗi phần tử trống hoặc bao gồm một gốc (nút) được liên kết với hai cây nhị phân khác nhau, được gọi là cây con bên trái và bên phải của gốc.

Деревья, у которых d > 2, называются d-арными.

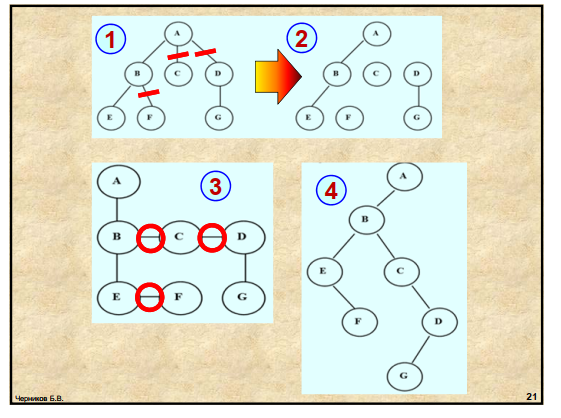
Cây có d> 2 được gọi là d-ary.

**Преобразование произвольного дерева с упорядоченными узлами в бинарное дерево:**

Chuyển đổi một cây tùy ý với các nút được đặt hàng thành cây nhị phân:

1. В каждом узле исходного дерева **вычеркиваем все ветви**, кроме самых левых ветвей.

Trong mỗi nút của cây nguồn, gạch bỏ tất cả các nhánh trừ các nhánh ngoài cùng bên trái.

1. Соединяем **горизонтальными ветвями** узлы одного уровня, которые являются «**братьями**» в исходном дереве (если несколько узлов имеют общего предка, то такие узлы – «братья»).

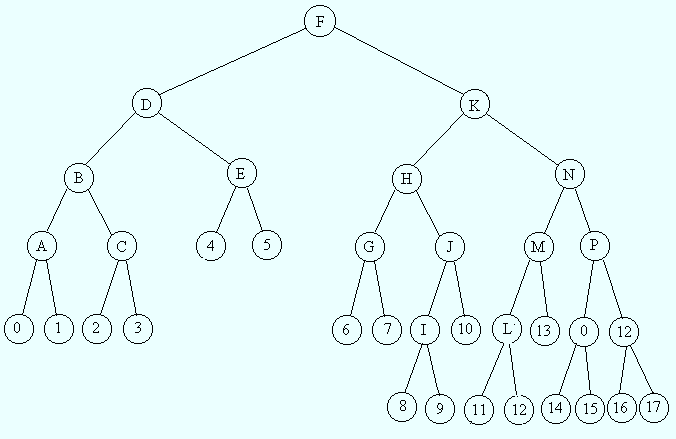
Kết nối các nhánh ngang của các nút có cùng cấp độ, đó là các anh em của họ trong cây nguồn (nếu một số nút có tổ tiên chung, thì các nút đó là các anh em Riết).

1. Левым потомком каждого узла x считается непосредственно **находящийся под ним узел** (если он есть), а в качестве правого потомка – **соседний справа «брат**» для x, если таковой имеется.

Hậu duệ bên trái của mỗi nút x được coi là nút ngay bên dưới nút đó (nếu có một), và là hậu duệ bên phải, anh em nhà Riết liền kề bên phải cho x, nếu có.

### **СБАЛАНСИРОВАННОЕ БИНАРНОЕ ДЕРЕВО**

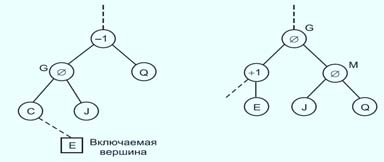
Cây nhị phân cân bằng

Бинарное дерево – сбалансированное (В-balanced), если высота **левого** поддерева каждого узла отличается от высоты **правого** не более чем на 1.

Cây nhị phân được cân bằng (cân bằng B) nếu chiều cao của cây con bên trái của mỗi nút khác với chiều cao của cây bên phải không quá 1.

**Показатель сбалансированности** узла, то есть разность высот правого и левого поддерева обозначается В = +1; 0; –1.

Chỉ báo cân bằng của nút, nghĩa là chênh lệch chiều cao của cây con bên phải và bên trái được ký hiệu là B = +1; 0; -1.



При восстановлении баланса дерева по высоте учитывается показатель В.

Khi khôi phục lại sự cân bằng của cây theo chiều cao, chỉ số B được tính đến.

**Символы указывают:**

* Левое поддерево **выше** правого **(+1)**

Cây con bên trái phía trên bên phải (+1)

* Поддеревья **равны по высоте (∅)**

Các cây con có chiều cao bằng nhau (∅)

* Правое поддерево **выше** левого **(–1)**

Cây con bên phải phía trên bên trái (1.000)

## **ПОИСК ХЕШИРОВАНИЕМ**

### **ФУНКЦИЯ ХЕШИРОВАНИЯ**

В рассмотренных методах поиска число итераций в лучшем случае было пропорционально O (log n).

Trong các phương pháp tìm kiếm được xem xét, số lần lặp tốt nhất tỷ lệ thuận với O (log n).

Надо найти такой метод поиска, при котором число итераций **не зависело бы от размера таблицы**, а в идеальном случае поиск сводился бы **к одному шагу.**

Cần tìm một phương thức tìm kiếm trong đó số lần lặp sẽ không phụ thuộc vào kích thước của bảng và trong trường hợp lý tưởng, tìm kiếm sẽ giảm xuống một bước.

Идеально быстрый поиск - **таблица прямого доступа** - (tìm kiếm nhanh hoàn hảo - bảng truy cập trực tiếp).

При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями.

Khi tạo bảng, bộ nhớ được phân bổ để lưu trữ toàn bộ bảng và chứa đầy các bản ghi trống

Затем записи вносятся в таблицу - каждая на свое место, определяемое ее ключом.

Sau đó, các mục được thực hiện trong bảng - mỗi mục ở vị trí của nó, được xác định bởi khóa của nó.

При поиске ключ используется как адрес и по этому адресу выбирается запись.

Khi tìm kiếm, khóa được sử dụng làm địa chỉ và bản ghi được chọn tại địa chỉ này.

Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице.

Nếu bản ghi được chọn trống, thì bản ghi với khóa này hoàn toàn không có trong bảng.

Пространство ключей – множество всех **теоретически возможных** значений ключей записи.

Không gian khóa là tập hợp tất cả các giá trị lý thuyết có thể có của các phím ghi.

Пространство записей – множество тех **ячеек памяти**, которые выделяются для хранения таблицы.

Một không gian ghi là tập hợp các ô nhớ được phân bổ để lưu trữ bảng.

Из соображений экономии памяти целесообразно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно.

Vì lý do tiết kiệm bộ nhớ, nên đặt kích thước của không gian ghi bằng với kích thước của bộ bản ghi thực tế hoặc vượt quá nó một chút.

Необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, т. е. преобразование ключа в адрес записи: а := h(k), где а – адрес, k – ключ.

Cần có một số chức năng cung cấp ánh xạ của một điểm từ không gian khóa sang một điểm trong không gian ghi, tức là, việc chuyển đổi khóa thành địa chỉ ghi: a: = h (k), trong đó a là địa chỉ, k là khóa.

Такая функция называется функцией хеширования (другие ее названия – **функция перемешивания, функция рандомизации).**

Hàm như vậy được gọi là hàm băm (các tên khác của nó là hàm trộn, hàm ngẫu nhiên).

Идеальной хеш-функцией является такая функция, которая для любых **двух неодинаковых ключей** дает **неодинаковые адреса**:

Hàm băm lý tưởng là một hàm như vậy, đối với bất kỳ hai khóa khác nhau sẽ cung cấp các địa chỉ khác nhau:

к1 ≠ k2 -> h(k1) ≠ h(k2).

Ситуация, при которой **разные ключи отображаются в один и тот же адрес записи**, называется коллизией, или переполнением, а такие ключи называются синонимами.

Một tình huống trong đó các khóa khác nhau được ánh xạ tới cùng một địa chỉ bản ghi được gọi là xung đột hoặc tràn và các khóa như vậy được gọi là từ đồng nghĩa.

Коллизии составляют основную проблему для хеш-таблиц.

Va chạm là một vấn đề lớn đối với bảng băm.

Если хеш-функция, преобразующая ключ в адрес, может порождать коллизии, то однозначной обратной функции: k := h’(a), позволяющей восстановить ключ по известному адресу, существовать не может -> **ключ должен обязательно входить в состав записи хешированной таблицы как одно из ее полей.**

Nếu hàm băm chuyển đổi khóa thành địa chỉ có thể gây ra xung đột, thì hàm nghịch đảo duy nhất: k: = h '(a), cho phép bạn khôi phục khóa thành địa chỉ đã biết, không thể tồn tại -> phải bao gồm khóa trong mục nhập bảng băm như một trong những lĩnh vực của cô.

**Требования к хеш-функции:**

Yêu cầu hàm băm:

* Должна обеспечивать **равномерное распределение** отображений фактических ключей по пространству записей.

Phải đảm bảo rằng ánh xạ của các phím thực tế được phân bổ đều trên không gian bản ghi.

* Должна порождать **как можно меньше коллизий** для данного фактического множества записей.

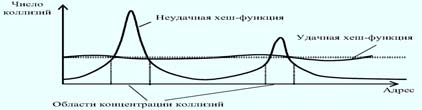
Nên tạo ra càng ít va chạm càng tốt cho một bộ hồ sơ thực tế nhất định.

* **Не должна отображать какую-либо связь** между значениями ключей в связь между значениями адресов;

Không được ánh xạ bất kỳ mối quan hệ nào giữa các giá trị chính với mối quan hệ giữa các giá trị địa chỉ;

* Должна быть **простой** и **быстрой для вычисления.**

Phải đơn giản và nhanh chóng để tính toán.



1. Деление по модулю числового значения ключа на размер пространства записи.

Việc chia modulo giá trị số của khóa theo kích thước của không gian ghi.

Результат интерпретируется как адрес записи.

Kết quả được hiểu là địa chỉ hồ sơ.

Плохо соответствует первым трем требованиям. Обычно применяется как последний шаг в более сложных функциях хеширования для приведения результата к размеру пространства записей.

Đáp ứng kém ba yêu cầu đầu tiên. Nó thường được sử dụng như bước cuối cùng trong các hàm băm phức tạp hơn để đưa kết quả về kích thước của không gian ghi.

1. Функция середины квадрата.

Hàm số giữa của hình vuông.

Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Giá trị khóa được chuyển đổi thành một số, số này sau đó được bình phương, một số chữ số ở giữa được chọn từ nó và được hiểu là địa chỉ bản ghi.

1. Функция свертки.

Hàm tích chập.

Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине адреса. Над частями производятся какие-то арифметические или поразрядные логические операции, результат которых интерпретируется как адрес.

Biểu diễn kỹ thuật số của khóa được chia thành các phần, mỗi phần có độ dài bằng với độ dài của địa chỉ. Các phép toán số học hoặc bitwise được thực hiện trên các phần, kết quả của nó được hiểu là một địa chỉ.

1. Функция преобразования системы счисления.

Hàm chuyển đổi của hệ thống số.

Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q > P. Обычно выбирают Q = P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

Khóa được viết dưới dạng số trong một số hệ thống số P được hiểu là một số trong hệ thống số Q> P. Thông thường, Q = P + 1 được chọn. Số này được chuyển từ hệ thống Q trở lại hệ thống P, giảm xuống kích thước của không gian ghi và được hiểu là một địa chỉ.

### **АЛГОРИТМ ПОИСКА ХЕШИРОВАНИЕМ**

В основе поиска лежит переход от исходного множества к множеству хеш-функций h(k)

Tìm kiếm dựa trên sự chuyển đổi từ bộ ban đầu sang bộ hàm băm h (k)

Хеш-функция имеет следующий вид: h(k) = k mod (m),

где k – ключ; m – целое число; mod – целочисленный остаток от деления.

Trong đó k là chìa khóa; m là một số nguyên; mod - số nguyên còn lại của phép chia.

**Пример**. Пусть дано множество **{9, 1, 4, 10, 8, 5}.**

Определим для него хеш-функцию **h(k)= k mod (m).**

Chúng tôi xác định hàm băm h (k) = k mod (m) cho nó.

1. Пусть **m = 1 -> h(k) = {0, 0, 0, 0, 0, 0}** -> Множество хеш–функций состоит из нулей - Tập hợp các hàm băm bao gồm các số không.

2. Пусть **m = 20 -> h(k) = {9, 1, 4, 10, 8, 5}** -> Множество хеш-функций повторяет исходное множество - Tập hợp các hàm băm lặp lại bộ ban đầu.

3. Пусть **m равно половине максимального ключа**

Đặt m bằng một nửa khóa tối đa

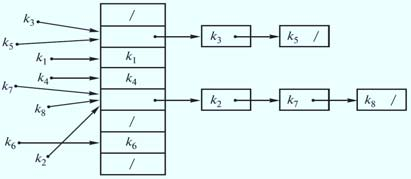
**m = [Kmax / 2] -> m = [10 / 2] = 5 -> h(k) = {4, 1, 4, 0, 3, 0}**

Хеш-функция указывает адрес, по которому **следует отыскивать ключ.**

Hàm băm cho biết địa chỉ để tìm khóa.

Для разных ключей хеш-функция может принимать **одинаковые значения**, такая ситуация называется коллизией.

Поиск хешированием заключается **в устранении коллизий методом цепочек**.



**Пример 1.** Дано множество ключей **{7,13,6,3,9,4,8,5}.** Найти ключ K=27

Хеш-функция равна h(k) = K mod (m);

m = [13/2] = 6 (т.к. 13 – максимальный ключ)

h(k) = {1, 1, 0, 3, 3, 4, 2, 5}

Попарным сравнением множества хеш-функций и множества исходных ключей, заполняем таблицу.

При этом хеш-функция указывает адрес, по которому следует отыскивать ключ

Если отыскивается ключ K = 27, тогда h(k) = 27 mod 6 = 3

Это значит, что ключ K = 27 может быть только **в 3-й строке.**

Но его там нет -> данный ключ отсутствует в исходном множестве.

**Пример 2.** Дано множество ключей

**{7, 1, 8, 5, 14, 9, 16, 3, 4}**

 Найти ключ K = 14.

Хеш-функция равна h(k) = K mod (m);

m = [16/2] = 8 (т.к. 16 – максимальный ключ)

h(k) = {7, 1, 0, 5, 6, 1, 0, 3, 4}

Поиск осуществляется по таблице: K = 14

h(k) = 14 mod 8 = 6 -> ключ K = 14 может быть только в 6-й строке.

## **ПОИСК ПО БОРУ**

Бор (trie, луч, нагруженное дерево) – структура данных для хранения **набора строк**, представляющая из себя «подвешенное» дерево с символами на ребрах (по одному на каждом ребре).

Boron (trie, chùm, cây được tải) - một cấu trúc dữ liệu để lưu trữ một chuỗi các chuỗi, đó là một cây "lơ lửng" với các ký tự ở các cạnh (một trên mỗi cạnh).

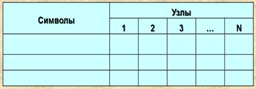
Строки получаются прохождением из корня по ребрам, записывая соответствующие им символы, до терминальной вершины.

Các dòng có được bằng cách chuyển từ gốc dọc theo các cạnh, viết các ký tự tương ứng vào đỉnh cuối.

**Размер бора** линейно зависит от **суммы длин всех строк**, а поиск в бору занимает **время**, **пропорциональное длине образца.**

Kích thước của lỗ khoan tuyến tính phụ thuộc vào tổng độ dài của tất cả các chuỗi và việc tìm kiếm trong lỗ khoan mất thời gian tỷ lệ thuận với chiều dài của mẫu.

Бор представляет собой **m-арное дерево**. Каждый узел уровня h представляет **множество всех ключей**, начинающихся с **определенной последовательности из h** **литер**. Узел определяет **m-путевое разветвление** в зависимости **от (h + 1)-ой литеры.**

Boron là một cây m-ary. Mỗi nút của cấp h đại diện cho tập hợp tất cả các khóa bắt đầu bằng một chuỗi chữ h nhất định. Nút xác định phân nhánh m-way tùy thuộc vào chữ cái (h + 1).

* В первом узле записывается **первая буква или цифра** ключа.

Chữ cái hoặc số đầu tiên của khóa được ghi trong nút đầu tiên.

* Во втором узле к ней **добавляется** еще один символ и т.д.

Trong nút thứ hai, một ký tự khác được thêm vào nó, v.v.

* Если слово, начинающееся с определенной буквы (цифры), – **единственное**, то оно сразу записывается в первом узле.

Nếu từ bắt đầu bằng một chữ cái (số) nhất định là từ duy nhất, thì từ đó được viết ngay lập tức trong nút đầu tiên.

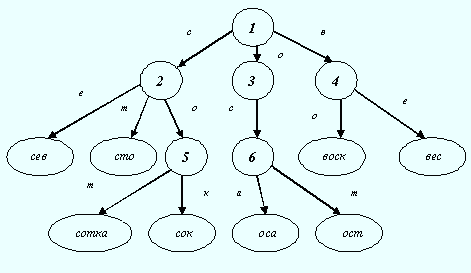
В графическом виде бор представляет собой граф – пример бора для хранения словаря **{he, she, his, hers}.**

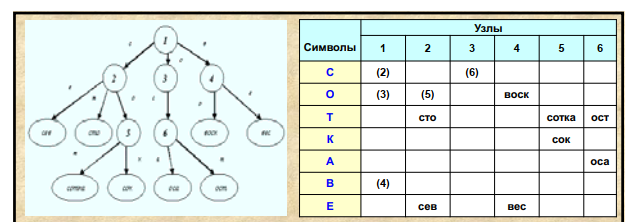
Về mặt đồ họa, boron là một biểu đồ - một ví dụ về boron để lưu trữ từ điển {anh, cô, của anh, của cô}.

**Пример.** Пусть задано множество слов:

Một ví dụ. Hãy để rất nhiều từ được đưa ra:

**{воск, сок, оса, сто, сев, век, ост, сотка}**





Пусть нужно найти слово ост

1. **Слово начинается на букву «о»** -> смотрим пересечение первого столбца со строкой, обозначенной буквой **«о».**

Từ bắt đầu bằng chữ cái o o - - chúng tôi nhìn vào giao điểm của cột đầu tiên với hàng được biểu thị bằng chữ cái

1. **Записана ссылка (3)** – обращаемся к третьему столбцу «бора».

Tài liệu tham khảo được ghi lại (3) - chúng tôi chuyển sang cột thứ ba của Bor boron.

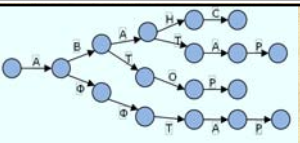
1. **Вторая буква слова ост – буква «с»** -> на пересечении третьего столбца и строки с буквой «с» записана ссылка **(6)**, которая отправляет нас на просмотр **шестого столбца** бора.

Chữ cái thứ hai của chữ Ost - chữ cái Cv -> ở giao điểm của cột thứ ba và dòng có chữ cái chữ cái c, một liên kết (6) được viết, để chúng ta xem cột thứ sáu của boron.

1. В этом столбце видим два слова: ост и оса. Поскольку третьей буквой искомого слова является буква «т», то на строке, обозначенной этой буквой, находим слово ост.

Trong cột này, chúng ta thấy hai từ: las và ong bắp cày. Vì chữ cái thứ ba của từ được tìm kiếm là chữ "t", nên trên dòng được ký hiệu bằng chữ này, chúng tôi tìm thấy các từ.

Бор – классическая структура данных для **хранения набора строк**.

Bor là một cấu trúc dữ liệu cổ điển để lưu trữ một chuỗi các chuỗi.

Размер бора **линейно зависит** **от суммы длин** всех строк, а поиск в бору занимает время, **пропорциональное длине образца.**

Kích thước của lỗ khoan tuyến tính phụ thuộc vào tổng độ dài của tất cả các chuỗi và việc tìm kiếm trong lỗ khoan mất thời gian tỷ lệ thuận với chiều dài của mẫu.

Сжатый бор – отличается от бора следующим: если у какой-то вершины **исходящая степень 1**, то эту вершину, ребро, входящее в нее, и ребро, исходящее из нее, можно **объединить в одно ребро с более чем одним символом.**

Boron nén - khác với boron theo cách sau: nếu một số đỉnh có độ ra là 1, thì đỉnh này, một cạnh đi vào nó và một cạnh ra khỏi nó, có thể được kết hợp thành một cạnh với nhiều hơn một ký hiệu.

l-слабый бор – улучшение сжатого бора - Boron yếu - cải thiện boron nén

Глубина вершины – длина пути от корня до этой вершины (глубина корня равна 0). l-слабый бор имеет структуру сжатого бора в вершинах с глубиной, меньшей l.

Độ sâu đỉnh - chiều dài của đường dẫn từ gốc đến đỉnh này (độ sâu gốc là 0). Boron yếu có cấu trúc của boron nén ở các đỉnh có độ sâu nhỏ hơn l.

На уровне l ветвление заканчивается, 2-слабый бор и все суффиксы **дописываются на** **ребрах**, исходящих из вершин глубины l.

Ở cấp độ l, các nhánh kết thúc, một rừng thông yếu 2 và tất cả các hậu tố được thêm vào các cạnh phát ra từ các đỉnh của độ sâu l.

В частности, из вершины глубины l могут выходить ребра, помеченные строками, начинающимися **с одной буквы.**

Cụ thể, các cạnh được đánh dấu bằng các dòng bắt đầu bằng một chữ cái có thể đi ra từ một đỉnh có độ sâu l.

## **АЛГОРИТМЫ ПОИСКА СЛОВЕСНОЙ ИНФОРМАЦИИ**

NHỮNG VẤN ĐỀ ĐỂ TÌM KIẾM THÔNG TIN ĐỊA PHƯƠNG

Пусть задан массив Txt из N элементов, называемый **текстом**, и массив Wrd из M элементов, называемый **словом (0 < M < N).**

Đặt một mảng Txt gồm các phần tử N, được gọi là văn bản và một mảng Wrd của các phần tử M, được gọi là một từ (0 <M <N).

Описать их можно как строки.

Bạn có thể mô tả chúng như các chuỗi.

Поиск слова обнаруживает первое вхождение Wrd в Txt.

Một tìm kiếm từ phát hiện sự xuất hiện đầu tiên của Wrd trong Txt.

**Прямой поиск строки**

Tìm kiếm chuỗi trực tiếp

Посимвольное сравнение текста со словом

So sánh từng ký tự của một từ

* **Начальный момент** -> сравнение первого символа текста с первым символом слова, второго символа текста со вторым символом слова и т. д.

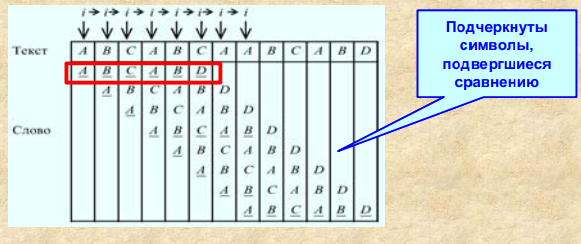
Điểm bắt đầu -> so sánh ký tự đầu tiên của văn bản với ký tự đầu tiên của từ, ký tự thứ hai của văn bản với ký tự thứ hai của từ, v.v.

* **Cовпадение всех символов** -> фиксируется факт нахождения слова.

Khớp tất cả các ký tự -> thực tế tìm từ được ghi lại.

* **В противном случае** -> «сдвиг» слова на одну позицию вправо и повторяется посимвольное сравнение, т. е. сравнивается второй символ текста с первым символом слова, третий символ текста со вторым символом слова и т. д.

Mặt khác -> Chuyển dịch từ một vị trí sang phải và so sánh từng ký tự được lặp lại, nghĩa là ký tự thứ hai của văn bản được so sánh với ký tự đầu tiên của từ, ký tự thứ ba của văn bản với ký tự thứ hai của từ, v.v.



Подчеркнуты символы, подвергшиеся сравнению

Nhân vật nhấn mạnh so sánh

Такие «сдвиги» слова повторяются, пока конец слова не достигнет конца текста или не произошло полное совпадение символов слова с текстом (т. е. слово найдено).

Các dịch chuyển như vậy Các từ của từ được lặp đi lặp lại cho đến khi hết từ này đến cuối văn bản hoặc có sự trùng khớp hoàn toàn giữa các từ của từ với văn bản (nghĩa là từ được tìm thấy).

function DirectTxtSearch(var Wrd: TWrd;

var Txt: TText;

var Position: integer): boolean;

{Функция поиска слова Wrd в тексте Txt,}

{если слово найдено, то возвращает значение true}

{и позицию Position начала первого слова Wrd,}

{иначе – false и Position не изменяется}

var i: integer; {Индекс начала слова в тексте}

j: integer; {Индекс текущего символа слова}

begin

i := 0;

repeat

j := 1; i := i + 1;

{Осуществляем посимвольное сравнение}

while (j <= M) and (Txt[i+j-1] = Wrd[j]) do j := j+1;

until (j = M+1) or {Совпало все слово}

(i >= N-M+1); {Конец слова за концом текста}

{Оценка результатов поиска}

if j = M+1 then

begin DirectTxtSearch := true; Position := i; end

else begin DirectTxtSearch := false; end;

end;

В худшем случае алгоритм будет малоэффективен: его сложность будет пропорциональна O((N – М)·М).

Trong trường hợp xấu nhất, thuật toán sẽ không hiệu quả: độ phức tạp của nó sẽ tỷ lệ thuận với O ((N - M) · M).

### **АЛГОРИТМ КНУТА – МОРРИСА – ПРАТТА (КМП)**

В 1970 г. Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм (КМП-алгоритм), фактически требующий только **O(N)** сравнений даже **в самом плохом случае.**

Năm 1970, D. Knut, D. Maurice và W. Pratt đã phát minh ra một thuật toán (thuật toán KMP), trên thực tế chỉ cần so sánh O (N) ngay cả trong trường hợp xấu nhất.

**Идея**: После частичного совпадения начальной части слова с символами текста известна пройденная часть текста и можно «вычислить» некоторые сведения (на основе самого слова), с помощью которых затем быстро продвинуться по тексту.

Ý tưởng: Sau một sự trùng hợp một phần của phần đầu của từ với các ký tự của văn bản, phần được thông qua của văn bản được biết đến và một số thông tin có thể được tính toán, dựa trên chính từ đó, với sự trợ giúp của nó sẽ nhanh chóng chuyển qua văn bản.

Если j определяет позицию в слове, содержащую первый несовпадающий символ (как в алгоритме прямого поиска), то величина сдвига **Shift** определяется как j – LenSuff – 1.

Nếu j xác định vị trí trong từ có chứa ký tự không khớp đầu tiên (như trong thuật toán tìm kiếm trực tiếp), thì giá trị dịch chuyển Shift được xác định là j - LenSuff - 1.

Значение LenSuff определяется как размер самой длинной последовательности символов слова, **непосредственно** **предшествующих позиции j (суффикс),** которая **полностью совпадает с началом слова.**

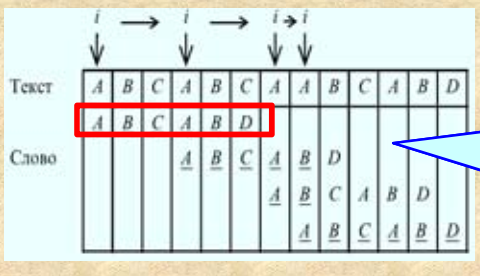
Giá trị LenSuff được định nghĩa là kích thước của chuỗi ký tự dài nhất trong từ ngay trước vị trí j (hậu tố), hoàn toàn trùng khớp với phần đầu của từ.

LenSuff зависит только от слова и не зависит от текста.

LenSuff là từ chỉ và văn bản độc lập.

Для каждого j будет своя величина сдвига.

Mỗi j sẽ có giá trị thay đổi riêng.

При каждом несовпадении пары символов слово сдвигается на переменную величину, и меньшие сдвиги не могут привести к полному совпадению.

Với mỗi sự không khớp của một cặp ký tự, từ được thay đổi bởi một biến và các ca nhỏ hơn không thể dẫn đến một kết hợp hoàn chỉnh.

Алгоритм получает на вход слово

X = x[1]x[2] ... x[n]

и просматривает его слева направо буква за буквой, заполняя приэтом массив натуральных чисел L[1] ... L[n], где L[i] = длина слова L(x[1]...х[i]).

và quét nó từ chữ này sang chữ khác từ trái sang phải, điền vào mảng các số tự nhiên L [1] ... L [n], trong đó L [i] = độ dài của từ L (x [1] ... x [i]).

Таким образом, L[i] есть длина наибольшего начала слова x[1]...x[i], одновременно являющегося его концом.

Do đó, L [i] là độ dài bắt đầu lớn nhất của từ x [1] ... x [i], cũng là kết thúc của nó.

**Пример**. Используя алгоритм КМП, определить, является ли слово A подсловом слова B?

Một ví dụ. Sử dụng thuật toán KMP, xác định xem từ A có phải là từ phụ của từ B không?

**Решение**. Применим алгоритм КМП к слову A#B, где # – специальная буква, **не встречающаяся** ни в A, ни в B.

Giải pháp. Chúng tôi áp dụng thuật toán KMP cho từ A # B, trong đó # là một chữ cái đặc biệt không tìm thấy trong cả A hoặc B.

Слово A является подсловом слова B тогда и только тогда, когда среди чисел в массиве L будет число, равное длине слова A.

Từ A là một từ phụ của từ B khi và chỉ khi trong số các số trong mảng L sẽ có một số bằng độ dài của từ A.

Предположим, что первые i значений L[1]...L[i] уже найдены.

Giả sử rằng các giá trị i đầu tiên của L [1] ... L [i] đã được tìm thấy.

Читается очередная буква слова (т.е. x[i+1]) и вычисляется L[i+1].

Chữ cái tiếp theo của từ được đọc (tức là x [i + 1]) và L [i + 1] được tính.



Определить начала Z слова x[1]...x[i+1], одновременно являющиеся его концами – из них следует выбрать самое длинное.

Xác định Z bắt đầu của từ x [1] ... x [i + 1], đồng thời là kết thúc của nó - từ đó nên chọn từ dài nhất.

Рассмотрим все начала слова x[1]...x[i], являющиеся одновременно его концами. Из них выберем подходящие – те, за которыми следует буква x[i+1]. Из подходящих выберем самое длинное. Приписав в его конец х[i+1], получим искомое слово Z. НО!!! Все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции L.

Hãy xem xét tất cả các bắt đầu của từ x [1] ... x [i], cả hai đều là kết thúc của nó. Chúng tôi sẽ chọn những người phù hợp từ họ - những người theo sau là chữ x [i + 1]. Từ phù hợp chúng tôi chọn lâu nhất. Ghi x [i + 1] vào cuối, chúng ta có được từ Z. NHƯNG mong muốn !!! Tất cả các từ đồng thời là bắt đầu và kết thúc của một từ nhất định có thể thu được bằng các ứng dụng lặp lại của hàm L.

i:=1; l[1]:=0;

{таблица l[1]..l[i] заполнена правильно}

while i <> n do

begin

len:= l[i]

{len – длина начала слова x[1]..x[i], которое является

его концом; все более длинные начала оказались

неподходящими}

while (x[len+1]<>х[i+1]) and (len>0) do

begin {начало не подходит, применяем к нему функцию l}

len:=l[len];

end;

{нашли подходящее слово или убедились в его отсутствии}

if x[len+1]=x[i+1] then do

begin {х[1]..x[len] – самое длинное подходящее начало}

l[i+1]:=len+1;

end

else begin l[i+1]:= 0; end; {подходящих нет}

i:=i+1;

end;

**Пример**. Запишем алгоритм, проверяющий, является ли слово

X = x[1]...x[n] подсловом слова Y = y[1]...y[m].

**Решение**. Вычисляем таблицу L[1]...L[n]

j:=0; len:=0;

{len – длина максимального начала слова X,

одновременно являющегося концом слова y[1]..j[j]}

while (len<>n) and (j<>m) do

begin

while (x[len+1]<>у[j+1]) and (len>0) do

begin {начало не подходит, применяем к нему функцию l}

len: = l[len];

end;

{нашли подходящее слово или убедились в его отсутствии}

if x[len+1]=y[j+1] then do

begin {x[1]..x[len] – самое длинное подходящее начало}

len:=len+1;

end

else begin len:=0; end; {подходящих нет}

j:=j+1;

end;

{если len=n, то слово X встретилось;

иначе мы дошли до конца слова Y, так и не встретив X}

### **АЛГОРИТМ БОУЕРА – МУРА (БМ)**

BOWER ALGORITHM - MURA (BM)

КМП-алгоритм дает выигрыш тогда, когда неудаче предшествовало некоторое число совпадений – тогда слово сдвигается более чем на единицу.

Thuật toán KMP mang lại lợi ích khi thất bại xảy ra trước một số lượng khớp nhất định - sau đó từ được dịch chuyển nhiều hơn một.

Метод, предложенный Р. Боуером и Д. Муром в 1975 году (БМ-алгоритм), не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях.

Phương pháp được đề xuất bởi R. Bower và D. Moore năm 1975 (thuật toán BM) không chỉ cải thiện việc xử lý trường hợp xấu nhất, mà còn mang lại lợi ích trong các tình huống trung gian.

**Идея:** **Сравнение символов идет с конца слова, а не с начала**

Ý tưởng: So sánh các ký tự từ cuối của một từ, không phải từ đầu

Перед поиском на основе слова формируется некоторая таблица.

Trước khi tìm kiếm, một bảng được hình thành trên cơ sở của từ.

Пусть для каждого символа х из алфавита величина Shiftx – расстояние от самого правого в слове вхождения х до правого конца слова.

Đặt Shiftx là khoảng cách từ sự xuất hiện ngoài cùng bên phải của x trong từ đến cuối bên phải của từ cho mỗi ký tự x từ bảng chữ cái.

Пусть обнаружено расхождение между словом и текстом, причем символ в тексте, который не совпал – х.

Đặt sự khác biệt giữa một từ và một văn bản, với một ký hiệu trong văn bản không khớp - x.

Тогда слово сразу можно сдвинуть вправо так, чтобы самый правый символ слова, равный x, оказался бы в той же позиции, что и символ текста х.

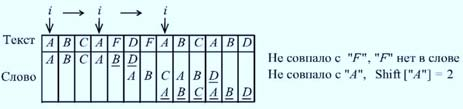
Sau đó, từ ngay lập tức có thể được dịch chuyển sang phải sao cho ký tự bên phải nhất của từ, bằng x, ở cùng vị trí với ký tự của văn bản x.

Этот сдвиг будет на число позиций, большее единицы.

Sự thay đổi này sẽ là một số vị trí lớn hơn một.

Если несовпадающий символ текста х в слове вообще не встречается, то сдвиг становится даже больше: сдвигаем вправо так, чтобы ни один символ слова не накладывался на символ х.

Nếu một ký tự không khớp x hoàn toàn không xuất hiện trong từ đó, thì sự thay đổi thậm chí còn lớn hơn: chúng ta dịch nó sang phải để không một ký tự nào của từ đó trùng với ký tự x.



function

BMTxtSearch(var Wrd: TWrd; var Txt: TText; var Position: integer): boolean;

{Функция поиска слова Wrd в тексте Txt, если слово найдено,

{то возвращает значение true и позицию Position начала первого слова Wrd, иначе –

false и Position не изменяется}

var

i, {Индекс начала слова в тексте}

j: integer; {Индекс текущего символа слова}

ch: char;

Shift: array[' '..'я'] of integer; {Массив смещений}

begin

{Заполнение массива Shift}

for ch:=' ' to 'я' do Shift[ch] := M;

for j:=1 to M do Shift[Wrd[j]] := M–j;

{Поиск слова Wrd в тексте Txt}

i := 1; {Начало слова совпадает с началом текста}

repeat

j := M+1; {Сравнивать будем с последнего символа}

{Посимвольное сравнение слова, начиная с правого символа}

repeat j := j–1; until (j < 1) or (Wrd[j] <> Txt[i+j–1]);

if j >= 1 then i := i + (j + Shift[Txt[i+j-1]] – M); {Сдвиг слова вправо}

until (j < 1) or (i > N–M+1);

{Оценка результатов поиска}

if j < 1 then begin BMTxtSearch := true; Position := i; end

else begin BMTxtSearch := false; end;

end;

Этот алгоритм в типичной ситуации **читает лишь небольшую часть всех букв слова**, в котором ищется заданный образец.

Trong một tình huống điển hình, thuật toán này chỉ đọc một phần nhỏ của tất cả các chữ cái của từ trong đó một mẫu đã cho được tìm kiếm.

**Пример**. Отыскивается образец abcd.

Một ví dụ. Một mẫu abcd được tìm thấy.

Посмотрим на четвертую букву слова в тексте: если, например, это буква e, то нет никакой необходимости читать первые три буквы (в образце буквы e нет, поэтому он может начаться не раньше пятой буквы)

Chúng ta hãy xem chữ cái thứ tư của từ trong văn bản: ví dụ, nếu đó là chữ e, thì không cần phải đọc ba chữ cái đầu tiên (không có chữ e trong mẫu, vì vậy nó không thể bắt đầu sớm hơn chữ cái thứ năm)

Почти всегда, кроме специально построенных примеров, данный алгоритм требует значительно меньше O(N) сравнений.

Hầu như luôn luôn, ngoại trừ các ví dụ được xây dựng đặc biệt, thuật toán này yêu cầu so sánh O (N) ít hơn đáng kể.

В самых благоприятных обстоятельствах, когда последний символ слова **всегда попадает на несовпадающий символ текста**, число сравнений пропорционально **O(N / M).**

Trong trường hợp thuận lợi nhất, khi ký tự cuối cùng của một từ luôn rơi vào một ký tự không khớp trong văn bản, số lượng so sánh tỷ lệ thuận với O (N / M).

# *LEK.#7* **ТЕМА 6 - СЛОВ**

## **ПРЯМОЙ ПОИСК:**

1) Первым действием мы производим сравнение первого символа текста с первым символом слова, второй символ текста со вторым символом слова и т.д.

1) Bằng hành động đầu tiên, chúng ta so sánh ký tự đầu tiên của văn bản với ký tự đầu tiên của từ, ký tự thứ hai của văn bản với ký tự thứ hai của từ, v.v.

Первый символ текста - это «п», а первый символ слова - это «э». Мы наблюдаем не совпадение символов.

Ký tự đầu tiên của văn bản là Tiếng p, Tiết và ký tự đầu tiên của từ này là Tiếng e. Chúng tôi quan sát không có sự trùng hợp của các nhân vật.

2) Вторым действием делаем «сдвиг» слова на одну позицию вправо, и повторяется посимвольное сравнение, т.е. сравнивается второй символ текста с первым символом слова, третий символ текста со вторым символом слова и т.д.

2) Hành động thứ hai là chuyển sang dịch vụ Thay đổi từ một vị trí sang phải và việc so sánh từng ký tự được lặp lại, tức là so sánh ký tự thứ hai của văn bản với ký tự đầu tiên của từ, ký tự thứ ba của văn bản với ký tự thứ hai của từ, v.v.

Как мы видим, вторым символом текста является «а», а первым символом слова – «э». Таким образом, опять наблюдаем несовпадение символов, поэтому производим «сдвиг» до тех пор, пока не «столкнемся» с символом «э». В данном тексте первый символ «э» встречается как раз в искомом слове. Посимвольно сравнивая слова «экспорт» с искомым словом, мы получает положительный результат.

Như chúng ta thấy, ký tự thứ hai của văn bản là một người khác, và ký tự đầu tiên của từ này là sự khác biệt. Do đó, một lần nữa, chúng tôi quan sát thấy một sự không phù hợp của các biểu tượng, do đó, chúng tôi thực hiện một sự thay đổi của người khác cho đến khi chúng tôi va chạm với biểu tượng của mình. Trong văn bản này, ký hiệu đầu tiên "e" được tìm thấy chỉ trong từ tìm kiếm. So sánh các từ biểu tượng xuất khẩu trực tuyến theo ký tự với từ tìm kiếm, chúng tôi nhận được một kết quả tích cực.

Количество шагов в таком поиске составило 350.

Số lượng các bước trong một tìm kiếm như vậy là 350.

## **АЛГОРИТМ КНУТА – МОРРИСА – ПРАТТА:**

1) Производим сравнение первого символа текста с первым символом слова.

1) Chúng tôi so sánh ký tự đầu tiên của văn bản với ký tự đầu tiên của từ.

Таким образом, видим несовпадение п<>э.

Như vậy, chúng ta thấy sự không phù hợp n <> e.

2) Вводим переменную J, которая определяет позицию в слове, содержащую первый несовпадающий символ (как в алгоритме прямого поиска).

2) Chúng tôi giới thiệu biến J, xác định vị trí trong từ có chứa ký tự không khớp đầu tiên (như trong thuật toán tìm kiếm trực tiếp).

Величина сдвига Shift определяется как j – LenSuff – 1

Giá trị thay đổi được định nghĩa là j - LenSuff - 1

Величина LenSuff определяется как размер самой длинной последовательности символов слова, непосредственно предшествующих позиции j, которая полностью совпадает с началом слова.

Giá trị LenSuff được định nghĩa là kích thước của chuỗi ký tự dài nhất của từ ngay trước vị trí j, hoàn toàn trùng khớp với phần đầu của từ.

Нужно учитывать, что для каждого j будет своя величина сдвига.

Phải xem xét rằng với mỗi j sẽ có giá trị dịch chuyển riêng.

В моем случае поиск слова «экспорт» будет аналогичным с прямым поиском слова, так как символы будут не совпадать до первого встречающегося символа «э». Он встречается в искомом слове «экспорт». Поэтому «сдвиг» всегда будет не более чем на один символ.

Trong trường hợp của tôi, việc tìm kiếm từ xuất khẩu xuất hiện tương tự như tìm kiếm trực tiếp từ này, vì các ký tự sẽ không khớp cho đến khi ký tự xuất hiện đầu tiên là E e. Nó được tìm thấy trong từ tìm kiếm "xuất khẩu." Do đó, "ca" sẽ luôn không quá một ký tự.

Делаем вывод, что количество шагов такого способа поиска так же составило 350.

Chúng tôi kết luận rằng số bước trong phương pháp tìm kiếm này cũng lên tới 350.

Алгоритм поиска:

## **АЛГОРИТМ БОУЭРА – МУРА**

1) Этот способ поиска отличается от двух предыдущих. Сравнение символов идет с конца слова, а не с начала. Последний символ искомого слова «т». Посимвольно ищем первый встречающийся символ «т». Далее сравниваем предыдущие буквы перед символами «т». Замечаем в первом случае, что а<>р.

1) Phương pháp tìm kiếm này khác với hai phương pháp trước. So sánh nhân vật đến từ cuối của một từ, không phải từ đầu. Ký tự cuối cùng của từ tìm kiếm "t". Chúng tôi đang tìm kiếm biểu tượng "t" xuất hiện đầu tiên theo từng ký tự. Tiếp theo, so sánh các chữ cái trước đó trước các ký tự "t". Chúng tôi lưu ý trong trường hợp đầu tiên rằng một <p.

2) Делаем «сдвиг» минимум на целое слово из-за несовпадения символов, т.к. этот алгоритм подразумевает запоминание проверяемых символов. Таким образом «сдвигаемся» до следующего символа «т».

2) Tạo một sự thay đổi của hoàng tử Ít nhất là toàn bộ một từ do sự không khớp của các ký tự, bởi vì thuật toán này liên quan đến việc lưu trữ các ký tự cần kiểm tra. Do đó, chúng tôi chuyển sang sử dụng biểu tượng tiếp theo.

3) Алгоритм заканчивает работу при нахождении слова «экспорт», т.е. при полном совпадении всех символов текста и всех символов искомого слова.

3) Thuật toán kết thúc khi tìm thấy từ xuất xuất xuất hiện, tức là với sự trùng hợp đầy đủ của tất cả các ký tự của văn bản và tất cả các ký tự của từ được tìm kiếm.

## **ВЫВОД**

Вывод: Самым эффективным способом в данном случае является алгоритм Боуэра – Мура.

Kết luận: Cách hiệu quả nhất trong trường hợp này là thuật toán Bower-Moore.

Количество шагов в таком поиске составило 45 шагов, что гораздо меньше количества шагов в двух первых способах.

Số bước trong tìm kiếm như vậy là 45 bước, ít hơn nhiều so với số bước trong hai phương thức đầu tiên.