1. Понятие о СД. Уровни представления СД. Классификация СД на логическом и физическом уровне

Структура данных (СД) — общее свойство информационного объекта, с которым взаимодействует та или иная программа. Это общее свойство характеризуется: множеством допустимых значений данной структуры;
набором допустимых операций; характером организованности.
Различают следующие уровни описания СД: абстрактный уровень; логический уровень;физический уровень.
Абстрактный уровень определяет характер организованности СД. Организованность СД может быть представлена множеством элементов, каждый из которых представляет собой СД, и отношениями между ними, свойства которых определяют различные типы СД.
Основные типы СД:
Множество - совокупность независимых элементов, отношения между которыми не заданы.
Последовательность - множество элементов, над которыми определены отношения линейного порядка, т.е. для каждого элемента, может быть за исключением первого и последнего, имеется один предыдущий и один последующий.
Дерево - множество элементов, над которыми определены отношения иерархического порядка, т.е. «один ко многим».
Граф - множество элементов, на которых определены отношения бинарного порядка, т.е. “многие ко многим”.
Логический уровень — представление СД на языке программирования. Простейшие СД, представляющие собой один элемент, определяются простыми типами. В языке Pascal для переменных простого типа определены множества допустимых значений и набор допустимых операций. Характер организованности — простейший.
СД которые полностью определяются типами данных языка программирования, называются встроенными. Для других СД, например, список, стек, очередь, дерево, таблица и др., нет соответствующих типов, определяющих организованность этих данных и допустимые операции. Такие СД называются производными и реализуются непосредственно программистами
Физический уровень — отображение на память ЭВМ информационного объекта в соответствии с логическим описанием. На этом уровне определяются область и объем памяти, необходимый для хранения экземпляра СД, форматы и интерпретация внутреннего представления. На физическом уровне, в памяти ЭВМ, СД могут иметь последовательную (прямоугольную) или связанную схему хранения, располагаться в статической или динамической памяти

2. СД типа «массив»: вычисление адреса для многомерного массива

Массив — последовательность элементов одного типа, называемого базовым. На абстрактном уровне массив представляет собой линейную структуру. На физическом уровне массив реализован последовательной (прямоугольной) схемой хранения. Располагаться он может в статической или динамической памяти. Размер памяти, выделяемый под массив, зависит от базового типа элемента массива и от количества элементов в массиве и определяется формулой $V_{мас} = V_{ел} \cdot k$; где Vмас — объем памяти для массива, Vэл — объем памяти одного элемента (слот), k — количество элементов.На логическом уровне СД типа массив можно описать следующим образом: Type T_ar = array [T1] of T2; {T1 — тип индекса } Var Ar : T_ar; {T2 — тип элемента}
Массив Ar типа T_ar располагается в статической памяти.
Массив — это статическая структура. В процессе выполнения программы количество элементов массива не изменяется. Доступ к элементу массива прямой, осуществляется через индекс элемента.
Вычисление адреса многомерного массива:

$$p_1 \le i_1 \le q_1$$
$$\ldots\ldots\ldots$$
$$p_n \le i_n \le q_n$$

$$\mathrm{Adr}(B[i_1, i_2, \ldots, i_n]) = \mathrm{Adr}(B[p_1, p_{2,} \ldots, p_n]) - S\sum_{k=1}^{n} p_k D_k + \sum_{k=1}^{n} i_k D_k$$

Dk зависит от способа размещения массива.
При размещении по строкам:
$D_k = (q_{k+1} - p_{k+1} + 1)D_{k+1}$, k = n – 1, n – 2, …., 1
При размещении по столбцам:
$D_k = (q_{k-1} - p_{k-1} + 1)D_{k-1}$, k = 2, 3, …, n

3. СД типа «запись»: прямое декартово произведение. Дескриптор записи

Структура — это множество элементов (полей), которые могут быть различных типов (аналогом в языке Pascal является запись).
На абстрактном уровне структура представляет собой множество — отношения между элементами отсутствуют.
На физическом уровне структура реализована последовательной схемой хранения. Располагаться она может в статической или в динамической памяти. Размер памяти, выделяемый под структуру, зависит от типов полей и от их количества и определяется формулой Vзап = ΣVi | i=1,k , где Vзап — объем памяти для структуры, k — количество полей, Vi —объем памяти для i-го поля.
На логическом уровне СД типа структура можно записать следующим образом:
typedef struct t_struct { S1: T1; S2: T2; ……. Sn: Tn; };t_struct str;
Здесь: S1, …, Sn — идентификаторы полей; T1, …, Tn — типы полей; str — идентификатор записи; t_struct — тип записи.
Если DT1 — множество значений элементов типа T1, DT2 — множество значений элементов типа T2, … , DTn — множество значений элементов типа Tn, то Dt_str — множество значений элементов типа t_str будет определяться с помощью прямого декартова произведения:
$D_{t\_str} = D_{T1} \cdot D_{T2} \cdot \ldots \cdot D_{Tn}$.
Кардинальное число для структуры t_str: Car(t_str) = Π Car(Ti) | i=1,n.
Допустимые операции для СД типа структура аналогичны операциям для СД типа массив. По характеру изменчивости структура — это статическая структура данных. Доступ к элементам структуры прямой, осуществляется по имени поля.

4. СД типа «таблица»: классификация операций

Таблица — это набор элементов одинаковой организации, каждый из которых можно представить в виде двойки <K,V>, где K — ключ, а V — тело (информационная часть) элемента. Ключ уникален для каждого элемента, т.е. в таблице нет двух элементов с одинаковыми ключами. Ключ используется для доступа к элементам при выполнении операций.

Таблица — динамическая структура. Над таблицей определены следующие основные операции:

1. Инициализация. 2. Включение элемента. 3. Исключение элемента с заданным ключем. 4. Чтение элемента с заданным ключем. 5. Изменение элемента с заданным ключем. 6. Проверка пустоты таблицы. 7. Уничтожение таблицы.

На абстрактном уровне таблица представляет собой множество.

На физическом уровне таблица реализуется последовательной или связной схемой хранения. Располагаться таблица может в статической или динамической памяти. В зависимости от способа размещения элементов таблицы классифицируются на неупорядоченные, упорядоченные и хеш-таблицы.

Для хранения элементов неупорядоченной таблицы используется дополнительная структура данных — линейный список (последовательный или односвязный). Элементы списка неупорядочены по значению ключа. Для поиска элемента с заданным ключем применяется алгоритм линейного или быстрого линейного поиска.

Для хранения элементов упорядоченной таблицы можно использовать дополнительные структуры данных — линейный список (последовательный или односвязный) или бинарное дерево. Если элементы таблицы расположены в линейном списке, то они упорядочиваются по возрастанию значений ключа. Для поиска элемента с заданным ключем применяются алгоритмы линейного, бинарного или блочного (в случае реализации таблицы на основе последовательного списка) поиска. После выполнения операции включения элементы списка должны остаться упорядоченными, поэтому перед выполнением операции включения необходимо найти элемент списка, после которого нужно включить элемент.

Классификация операций

Конструкторы – эти операции создают информационный объект рассматриваемой структуры, они выполняются первыми и все поля переменных инициализируются

Деструкторы – операции осуществляют разрушение объекта или освобождение занимаемой памяти.

Модификаторы – эти операции на входе объекты заданной структуры, на выходе те же объекты с другим количеством элементов

Наблюдатели – операции в качестве входных параметров – объекты структуры, а возвращают операции другого типа. Эти операции ничего не меняют, ни кол-во оперпций, ни структуру

Итераторы – операции доступа к элементам объекта в определенном порядке.

5. Временная сложность алгоритмов. Порядок функции временной сложности. Их определение в алгоритмах поиска

Временная сложность (ВС) алгоритма — это зависимость времени выполнения алгоритма от количества обрабатываемых входных данных. Здесь представляет интерес среднее и худшее время выполнения алгоритма. ВС можно установить с различной точностью. Наиболее точной оценкой является аналитическое выражение для функции: t = t(N), где t — время, N — количество входных данных (размерность). Данная функция называется функцией временной сложности (ФВС). Две функции f1(N) и f2(N) одного порядка, если  . Иначе это записывается в виде: f1(N) = O(f2(N)) (Читается «О большое»).

Порядок функции, заданной многочленом, определяется только тем членом, который растет быстрее других с увеличением N, причем коэффициент при нем не учитывается. Для определения порядка функции ВС алгоритма достаточно найти зависимость числа выполнения того оператора от количества исходных данных, который выполняется в алгоритме чаще других.Для оценки алгоритмов можно использовать функцию зависимости числа операций сравнения C = C(N) от количества обрабатываемых данных, т.к. функции t(N) и C(N) — функции одного порядка. Порядок функции временной сложности дает объективную оценку эффективности алгоритма и не зависит ни от мощности компьютера, ни от реализации.

Определение функции временной сложности в алгоритмах поиска: *Линейный поиск:* ПФВС O(N) *Бинарный поиск:* ПФВС O(log2N) т.к. на каждом шаге поиска вдвое уменьшается область поиска *Блочный поиск:* ПФВС зависит от выбранного поиска внутри блока.

6. Основные факторы выбора алгоритмов сортировки. Базовые сортировки. Анализ сложности

- Имеющиеся ресурсы памяти: Должны ли входные и выходные данные располагаться в разных областях памяти или выходные могут быть сформированы на месте входных
- Исходная упорядоченность входного множества: Во входном множестве могут попадаться упорядоченные части или оно вовсе упорядоченно. Одни алгоритмы не учитывают исходную упорядоченность, другие учитывают
- Временные характеристики операций: Числовые ключи сравниваются быстрее строковых, надо учитывать время перестановок, размер записи, которые надо переставлять

Существует 3 базовые сортировки: Сортировка включением Выбором Обменом

Алгоритм сортировки включением

1.Запоминаем элемент, подлежащий вставке. 2.Перебираем справа налево отсортированные элементы и сдвигаем каждый элемент вправо на одну позицию, пока не освободится место для вставляемого элемента. 3.Вставляем элемент на освободившееся место. Пункты 1-3 выполняем для всех элементов массива, кроме первого.

Анализ: Если первоначальный массив отсортирован, то на каждом просмотре делается только одно сравнение, так что эта сортировка имеет порядок O(N). Если массив первоначально отсортирован в обратном порядке, то данная сортировка имеет порядок $O(N^2)$, поскольку общее число сравнений равно $1 + 2 + 3 + ... + (N – 2) + (N – 1) = (N – 1)\cdot N / 2$, что составляет $O(N^2)$. Чем ближе к отсортированному виду, тем более эффективной становится сортировка простыми вставками. Среднее число сравнений в сортировке простыми вставками также составляет $O(N^2)$.

Алгоритм сортировки выбором

1.Находим наименьший элемент в неупорядоченной части массива. 2.Меняем местами найденный элемент с тем, который соседствует с упорядоченной частью. 2.Пункты 1 и 2 выполняем, пока в неупорядоченной части имеется более одного элемента.

Анализ:При сортировке простым выбором число сравнений ключей не зависит от их начального порядка. На первом просмотре выполняется N – 1 сравнение, на втором — N – 2 и т.д. Следовательно, общее число сравнений равно $(N – 1) + (N – 2) + (N – 3) + ... + 1 = N\cdot(N – 1) / 2$, что составляет $O(N^2)$.

Порядок функции ВС не зависит от упорядоченности сортируемого массива, однако время сортировки упорядоченного массива будет минимальным, т.к. от упорядоченности массива зависит число перестановок элементов.

Алгоритм сортировки обменом

1. Перебираем поочередно все пары соседних элементов, начиная с последнего, и меняем местами элементы в парах, нарушающих порядок.
2. Пункт 1 повторяем n – 1 раз.

Анализ: При использовании сортировки обменом число сравнений элементов не зависит от их начального порядка. На первом просмотре выполняется N – 1 сравнение, на втором — N – 2 и т.д. Следовательно, общее число сравнений равно $(N – 1) + (N – 2) + (N – 3) + ... + 1 = N\cdot(N – 1) / 2$, что составляет $O(N^2)$.

Перестановки в упорядоченном массиве не выполняются, а в массиве, упорядоченном в обратном порядке, их число равно числу сравнений элементов. Следовательно, время выполнения алгоритма зависит от упорядоченности массива.

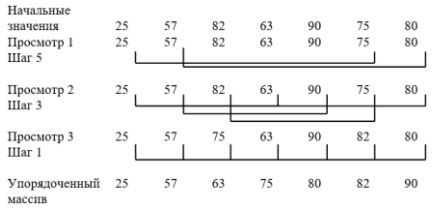
7. Улучшенные сортировки. Сортировка Шелла. Анализ сложности

Улучшенная сортировка обменом 1: После каждого прохода в сортировке обменом может быть сделана проверка, были ли совершены перестановки в течение данного прохода. Если перестановок не было, то это означает, что массив упорядочен и дальнейших проходов не требуется.

Анализ улучшенной сортировки обменом 1: Число сравнений для этого метода зависит от числа проходов, необходимых для сортировки. Общее число сравнений равно N·(N – 1) / 2, что составляет O(N2). В этом случае выполняется максимальное число перестановок, что увеличивает время выполнения алгоритма. В лучшем случае, когда массив уже упорядочен, потребуется всего один проход и N – 1 сравнение, что составляет O(N). Перестановки в этом случае не выполняются.

Улучшенная сортировка обменом 2: В отличие от улучшенной сортировки обменом 1 здесь в течение прохода фиксируется последний элемент, участвующий в обмене. В очередном проходе этот элемент и все предшествующие в сравнении не участвуют, т.к. все элементы до этой позиции уже отсортированы.

Анализ улучшенной сортировки обменом 2: Анализ улучшенной сортировки обменом 2 аналогичен анализу улучшенной сортировки обменом 1. Порядок функций ВС этих алгоритмов в лучшем и худшем случаях одинаковый.



Сортировка Шелла — это улучшенный метод сортировки вставками. Рассмотрим этот метод на примере (см. ниже). При первом проходе группируются и сортируются элементы, отстоящие друг от друга на 5 позиций: (X1,X6), (X2,X7), (X3), (X4), (X5), т.е. выполняется сортировка массива с шагом 5; при втором проходе — элементы, отстоящие друг от друга на три позиции: (X1,X4,X7), (X2,X5), (X3,X6); при третьем — на 1 позицию.

Алгоритм сортировки Шелла: Определить количество проходов t и шаг сортировки на каждом проходе. Результат сохранить в массиве h. На i-ом проходе (i=1,…,t) выполнить сортировку включением с шагом h(i). *Анализ:* Д. Кнут предложил выбирать шаги из следующего ряда: 1, 4, 13, 40, 121, .. а вычислять их по формуле: hi – 1 = 3· hi + 1; ht = 1; t = [log 3 N] – 1 (количество просмотров), где N — размерность исходного массива. Т.е. элементы должны быть взаимно простыми числами. Исходя из этого порядок сортировки может быть аппроксимирован величиной O(N(log 2 N)).

8. Улучшенные сортировки. Быстрая сортировка выбором. Анализ сложности

Улучшить сортировку выбором можно, если после каждого прохода получать больше информации, чем просто идентификация минимального элемента. Пример: 10 – элемент MaxInt. k1*(N-1) – количество просмотров; log 2 N – высота бинарного дерева.

Мы проходим от корня к листу по тому пути, по которому шел минимальный элемент (шаг 2). Затем осуществляем модификацию бинарного дерева. Общее количество модификаций: k2*log 2 N * (N-1).

Алгоритм быстрой сортировки выбором: 1. Сравниваем пары соседних ключей и запоминаем значение меньшего ключа из каждой пары. 2. Выполняем п.1 по отношению к значениям, полученным на предыдущем шаге. Так повторяем, пока не определим наименьший ключ и не построим бинарное дерево. 3. Вносим значение корня, найденное в п.2, в массив упорядоченных ключей. 4. Проходим от корня к листу дерева, следуя по пути, отмеченному минимальным ключом, и заменяем значение в листе на наибольшее целое число. 5. Проходим от листа к корню, по пути обновляя значения в узлах дерева, и определяем новый минимальный элемент. 6. Повторяем пп.3-6, пока минимальным элементом не будет MaxInt.

Анализ быстрой сортировки выбором. k1*(N-1)+ k2*log 2 N*(N-1) = O(N * log 2 N) (ФВС = порядок)

Сформулируем алгоритм перестроения дерева, у которого левое и правое поддерево — пирамиды, в пирамиду.

Алгоритм **MakeHeap.**

1. Запоминаем корневой элем. 2. Перебираем элементы в направлении большего сына и сдвигаем каждый элемент “вверх” на одну позицию, пока не освободится место для корневого элемента. 3. Вставляем корневой элемент на освободившееся место.

Для построения пирамиды из произвольного дерева (массива) необходимо построить пирамиду по алгоритму MakeHeap для каждого элемента, имеющего хотя бы одного сына, причем построение должно идти от последнего такого элемента к первому, т.е. снизу вверх.

Теперь можем сформулировать алгоритм пирамидальной сортировки.

Алгоритм **HeapSort.** 1. Построить пирамиду для исходного массива. 2. Пока в массиве более одного элем, переставить первый и последний элем, уменьшить размер массива на единицу и перестроить дерево в пирамиду по алгоритму MakeHeap.

Анализ пирамидальной сортировки

Пусть дерево массива на нижнем (нулевом) уровне содержит максимальное число вершин. Для построения пирамиды из произвольного дерева (первая часть алгоритма) нужно обработать все вершины, начиная с первого уровня. Для обработки вершины на i-том уровне число сравнений пропорционально i. Число вершин на i-том уровне равно $2^{[\log 2N]-1}$, а всего уровней m=[$\log_2$N], следовательно, общее число сравнений определяется формулой: $1\cdot 2^{[\log 2N]-1} + 2\cdot 2^{[\log 2N]-2} + \ldots + m\cdot 2^{[\log 2N]-m} = O(N)$.

Во второй части алгоритма последовательно обрабатываются деревья с N, N – 1, … ,2 вершинами. Число сравнений для перестройки дерева, состоящего из i вершин, в пирамиду пропорционально [$\log_2$i], следовательно, общее число сравнений

[$\log_2$ N] + [$\log_2$ (N – 1)] + … + [$\log_2$ 2] = O(N·$\log_2$ N).

Таким образом порядок функции ВС алгоритма пирамидальной сортировки O(N·$\log_2$N).

9. Улучшенные сортировки. Быстрая обменная сортировка (Хоара). Анализ сложности

Сортировка Хоара — лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его автор Ч. Хоар назвал эту сортировку быстрой. Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно разбить массив на подмассивы и сортировать подмассивы меньшего размера. **Алгоритм быстрой обменной сортировки:**

1. Выбираем элемент массива в качестве разделителя (например, первый).
2. Располагаем элементы меньшие разделителя в первой части массива, а большие — во второй.
3. Если число элементов в первой части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.
4. Если число элементов во второй части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.

Данный алгоритм основан на том утверждении, что ни один из элементов, расположенных в первой части массива не обменяется с каким либо элементом, расположенным во второй части.

Пункт 2 алгоритма выполняется следующим образом: просматриваем массив от краев к центру и меняем местами элементы, нарушающие порядок.

Пример.

(30 10 40 20 15 17 45 60)
(17 10 15 20) (40 30 45 60)
(30 10 17 20 15) (40 45 60)

Анализ сортировки Хоара: Пусть m = $\log_2$ N, где N — количество элементов. Будем считать, что количество элементов, меньших разделителя, равно количеству элементов, больших разделителя, т.е. массив разбивается пополам на две равные части. Определим количество сравнений в этом случае: N + 2·(N / 2) + … + m·(N / m) = O(N·m) = O(N·$\log_2$N). Если каждый раз одна из частей массива содержит не более одного элемента, то порядок будет O(N^2). Характер разбиения массива, а, следовательно, и порядок функции ВС, зависит от соотношения разделителя и остальных элементов массива. Для определения «хорошего» разделителя имеются различные алгоритмы.

10. СД типа «стек». Базовые операции. Интерфейс СД типа «стек»

Стек — это последовательность элементов, в которой доступ (операции включения, исключения, чтения элемента) осуществляется только с одного конца структуры — с вершины стека. Стек называют структурой типа LIFO (от англ. Last In, First Out). Иногда стек называют магазином (по аналогии с магазином автомата). Стек — это динамическая структура.

Над стеком определены следующие основные операции: 1. Инициализация. 2. Включение элемента. 3. Исключение элемента. 4. Чтение элемента. 5. Проверка пустоты стека. 6. Уничтожение стека.

Кардинальное число стека определяется по формуле: CAR(стек) = CAR(BaseType)0 + CAR(BaseType)1 + … + CAR(BaseType)max,

где CAR(BaseType) — кардинальное число элемента стека типа BaseType, max — максимальное количество элементов в стеке (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).

На абстрактном уровне стек представляет собой линейную структуру — последовательность.

На физическом уровне стек может быть реализован последовательной или связной схемой хранения. Располагаться стек может в статической или динамической памяти. Стек, по сути, является линейным списком с ограниченным доступом к элементам, по этому он может быть реализован на основе СД ЛС. Достаточно ввести ограничения на операции и обеспечить доступ к элементу, являющемуся вершиной стека. В качестве такого элемента может быть либо первый, либо последний элемент в ЛС.

Интерфейс СД типа стек на массиве

```
#define Stack_Size 100          const int StackOk=0; const int StackEmpty=1;          const int StackFull=2;extern int StackError=0;
typedef int BaseType;
typedef struct {        BaseType buf [Stack_Size];        unsigned uk;      } Stack;
void StackInit(Stack *s);        int StackisEmpty(Stack *s);      int StackisFull(Stack *s);        void StackPut(Stack *s, BaseType e);
void StackGet (Stack *s, BaseType *e);
```

11. СД типа «стек». Применение СД типа «стек» в вычислительных системах и алгоритмах см 10 вопрос

Используется в скрытом от программиста виде для обеспечения вложенных вызовов процедур.

Системы программирования используют стек для размещения в нем параметров значения и локальных переменных. При каждой активизации память для этих переменных выделяется в стеке, при завершении освобождается.

Рекурсия использует стек в скрытом виде.

Алгоритм быстрой обменной сортировки с помощью стека.

Применение стека в явном виде для постфиксного калькулятора(польская запись)

12. СД типа «очередь». Базовые операции. Интерфейс СД типа «очередь»

Очередь — это последовательность элементов, в которой включают элементы с одной стороны («хвост» очереди), а исключают с другой («голова» очереди). Очередь — это динамическая структура.

Над очередью определены следующие основные операции: 1. Инициализация. 2. Включение элемента. 3. Исключение элемента. 6. Проверка пустоты очереди. 7. Уничтожение очереди.

На абстрактном уровне очередь представляет собой линейную структуру — последовательность

На физическом уровне очередь может быть реализована последовательной или связной схемой хранения. Располагаться очередь может в статической или динамической памяти. Очередь, по сути, является линейным списком с ограниченным доступом к элементам, по этому он может быть реализован на основе СД ЛС. Достаточно ввести ограничения на операции и обеспечить доступ к элементам, расположенным в начале и в конце очереди.

Для того чтобы использовать пустые слоты используют кольцевую очередь, здесь могут различные состояния указателей.

Интерфейс СД типа очередь const int FifoOk = 0; const int FifoFull = 1;const int FifoEmpty = 2; extern int errorFifo;typedef int BaseType; typedef struct { BaseType buf[SIZE_FIFO]; unsigned ukEnd; unsigned ukBegin; unsigned len;} FIFO;
void initFifo(FIFO *F); void putFifo(FIFO *F, BaseType E); void geFifo(FIFO *F, BaseType *E); int FifoisFull(FIFO *F);
int FifoisEmpty(FIFO *F);

13. СД типа «очередь». Очереди с приоритетами См 12

В реальных задачах возникает необходимость в создании очередей отличных от FIFO (включение/исключение элементов из таких очередей определяется приоритетами элементов)

Приоритет может быть представлен числовым значением, как ключ, либо значением какого либо внешнего фактора.

Стек может быть частным случаем приоритетной очереди

Возможны очереди с приоритетным включением, в которых последовательность элементов очереди поддерживается упорядоченной, что позволяет исключить элемент с O(1), но перед включением необходимо просмотреть все элементы O(N)

Возможны очереди с приоритетным исключением. Новый элемент включается в конец очереди O(1), а при исключении ищется элемент с максимальным приоритетом O(N).

Задать приоритет можно так: typedef struct {int data; int priority;} baseType;

Для очереди с приоритетным исключением, операция исключения:

```
void getQueuePriority(QueuePriority *F, queuePriorityBaseType *E) { if (isEmptyQueuePriority(F)) {return;}
queuePriorityBaseType max = F->buf[0];   // Максимальным примем 1-й элемент
int maxPos = 0;                                  // Позиция элемента в макс. приоритетом
for (unsigned i = 1; i < F->uk; i++) {          // Для всех элементов начиная со второго выполнить
if (F->buf[i].priority > max.priority) {         // поиск элемента с максимальным приоритетом
max = F->buf[i]; maxPos = i; }}*E = max;  // Запись данных в переменную
F->buf[maxPos] = F->buf[--F->uk];           // Установка на позицию исключаемого элемента последнего Уменьшение длины }
```

14. СД типа «очередь». Применение очередей в вычислительных системах см 12

Кольцевая очередь применяется для буфера базовой системы ввода/вывода

Многозадачных операционных систем. Ресурсы вычислительной системы используются задачами которые одновременно выполняются такой ОС. Задачи претендующие на использование того или иного ресурса выстраиваются в очередь к этому ресурсу. В современных ОС одним из средств взаимодействия между параллельно выполнимыми задачами являются очереди сообщений. В этом случае каждая задача имеет очередь и все сообщения отправляются им от других задач. Задача владелец очереди выбирает из нее сообщение.

15. Связное представление данных в памяти компьютера. + и -

Динамические структуры по определению характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.
Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

- информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, записью и т.п.;
- поля связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Достоинства связного представления данных: 1. размер структуры ограничивается только доступным объемом машинной памяти и 2. при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей.
Недостатки: 1.работа с указателями требует, как правило, более высокой квалификации от программиста;
на поля связок расходуется дополнительная память и 2. доступ к элементам связной структуры может быть менее эффективным по времени.

16. Односвязный линейный список. Базовые операции. Функциональный и свободный списки

Линейный список (ЛС) — это конечная последовательность однотипных элементов (узлов).
Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться, поэтому ЛС — динамическая структура.
Над СД ЛС определены следующие основные операции: 1. Инициализация. 2. Включение элемента. 3. Исключение элемента.4. Чтение текущего элемента. 5. Переход в начало списка. 6. Переход в конец списка. 7. Переход к следующему элементу. 8. Переход к i-му элементу. 9. Определение длины списка. 10. Уничтожение списка.
Кардинальное число СД ЛС определяется по формуле:
CAR(ЛС) = CAR(BaseType)0 + CAR(BaseType)1 +… + CAR(BaseType)max, где CAR(BaseType) — кардинальное число элемента ЛС типа BaseType, max — максимальное количество элементов в ЛС (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).
На абстрактном уровне ЛС представляет собой линейную структуру — последовательность.
На физическом уровне ЛС может быть реализован последовательной или связной схемой хранения.
Важнейшие операции над связными списками - включение и исключение элементов. Список, уже сформированный к текущему моменту времени и хранящий в содержательных полях своих элементов полезную информацию, называют функциональным списком. Свободным списком называют такой дополнительный список элементов (или один элемент), который служит источником памяти при формировании функциональных списков. Каждый элемент свободного списка имеет такой же формат полей, как и элемент функционального списка, но содержимое информационной его части не определено.
Свободный список формируется из элементов, исключаемых из функционального списка. В случае если на удаляемом элементе не сохранить указатель, то его слот перестанет быть адресуемым, то есть в куче появится неадресуемое пространство - ’’дыра’’. С целью экономии памяти исключенный элемент следует включить в список, сформированный из других исключенных элементов одинакового формата, то есть в свободный список. В дальнейшем прежде чем добавить в функциональный список новый элемент, следует проверить соответствующий свободный список, и, в случае если в свободном списке есть элементы, то взять элемент из него, а не создавать в памяти новый элемент.

17. Односвязный линейный список. Реализация ОЛС как отображение на массив

Линейный список (ЛС) — это конечная последовательность однотипных элементов (узлов).
Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться, поэтому ЛС — динамическая структура.
Над СД ЛС определены следующие основные операции: 1. Инициализация. 2. Включение элемента. 3. Исключение элемента.4. Чтение текущего элемента. 5. Переход в начало списка. 6. Переход в конец списка. 7. Переход к следующему элементу. 8. Переход к i-му элементу. 9. Определение длины списка. 10. Уничтожение списка.
Кардинальное число СД ЛС определяется по формуле:
CAR(ЛС) = CAR(BaseType)0 + CAR(BaseType)1 +… + CAR(BaseType)max, где CAR(BaseType) — кардинальное число элемента ЛС типа BaseType, max — максимальное количество элементов в ЛС (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).
На абстрактном уровне ЛС представляет собой линейную структуру — последовательность.
На физическом уровне ЛС может быть реализован последовательной или связной схемой хранения.
Односвязный список состоит из двух полей: содержательного поля и указателя.
Элементы ОЛС располагаются в массиве. Элемент массива может содержать элемент ОЛС или быть «свободным». При включении элемента в ОЛС необходимо найти «свободный» элемент, сделать его «занятым» и включить в ОЛС. При исключении элемента нужно исключить элемент из ОЛС путем переопределения указателя предшествующего элемента на последующий и «освободить» исключенный элемент. «Свободные» элементы массива могут иметь специальный признак, отличающий их от элементов ОЛС, или могут быть объединены в список свободных элементов (ССЭ), на первый элемент которого указывает поле-указатель первого элемента массива. При включении элемента в ОЛС берется первый элемент ССЭ, а исключаемый элемент заносится в начало ССЭ.
В статической памяти находится дескриптор ОЛС, состоящий из трех полей:
1 — указатель на фиктивный элемент ОЛС;
2 — указатель на текущий элемент;
3 — длина ОЛС.
Адрес фиктивного элемента определяется при инициализации.
Элемент ОЛС может содержать либо поле данных, либо адрес данных.

18. Односвязный линейный список. Реализация ОЛС с использование указательного типа. Интерфейс ОЛС см 16 - 17

Линейный список (ЛС) — это конечная последовательность однотипных элементов (узлов).Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться, поэтому ЛС — динамическая структура.
При реализации ОЛС в статической памяти располагают дескриптор, состоящий из двух полей:
Указатель на фиктивный элемент
Указатель на текущий элемент

```
#ifndef LIST_H_INCLUDED    #define LIST_H_INCLUDED
static const int ListOk=0,  ListNotMem=1, ListEmpty=2, ListEnd=3;static short ListError;        //переменная ошибок
typedef   /*базовый тип*/  BaseType;        typedef struct element *ptrEl; typedef struct element{    BaseType data;    ptrEl next;};
typedef struct List {    ptrEl start;    ptrEl ptr;};
void InitList(List *L);       void PutList(List *L, BaseType E);   void GetList(List *L, BaseType *E);  //Исключение элемента из списка
```

int EmptyList(List *L); int EndList(List *L); void MovePtr(List *L); void DoneList(List *L); //Удаление списка (деструктор)
#endif // LIST_H_INCLUDED

19. Односвязный линейный список. Применение ОЛС. Стек как отображение на список. Очередь как отображение на список

Линейный список (ЛС) — это конечная последовательность однотипных элементов (узлов).
Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться, поэтому ЛС — динамическая структура.
Односвязный список состоит из двух полей: содержательного поля и указателя.
Применение: Линейные списки используются для представления таблиц, если размер таблицы может существенно меняться
Список можно применить для организации стека, для этого необходимо установить указатель на фиктивный элемент и не перемещать его. Список можно применить для организации очереди, чтобы включить элемент в очередь необходимо рабочий указатель переместить в конец, а если исключить, то рабочий указатель переместить в начало.
Стек:#ifndef STACONOLS_H_INCLUDED #define STACONOLS_H_INCLUDED #include "List.h" #define StackOk ListOk #define StackNotMem ListNotMem #define StackEnd ListEnd short StackError;typedef List Stack; void StackInit(Stack *S); void StackDone(Stack *S); void StackPut(Stack *S, BaseType E); void StackGet(Stack *S, BaseType *E); int StackEmpty(Stack *S); #endif // STACONOLS_H_INCLUDED
Пример: void StackInit(Stack *S) {InitList(S);StackError = ListError;}
Очередь:#ifndef FIFOOLS_H_INCLUDED #define FIFOOLS_H_INCLUDED #include "List.h" #define FifoOlsOk ListOk #define FifoOlsNotMem ListNotMem #define FifoEmpty ListUnder short FifotError; typedef List FifoOls; typedef BaseType BaseTypeFifo; void initFifoOls(FifoOls *FO); void putFifoOls(FifoOls *FO, BaseTypeFifo E); void getFifoOls(FifoOls *FO, BaseTypeFifo *E); void doneFifoOls(FifoOls *FO); int EmptyFifoOls(FifoOls *FO); #endif // FIFOOLS_H_INCLUDED
Пример: void putFifoOls(FifoOls *FO, BaseTypeFifo E); {EndPtr(FO);PutList(FO, E);FifotError =ListError;}

20. СД типа «циклический ОЛС». Интерфейс «циклического ОЛС» См 16-19

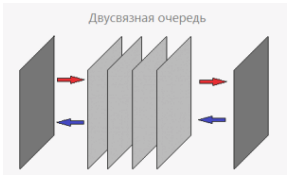
Каждый узел однонаправленного (односвязного) циклического списка (ОЦС) содержит одно поле указателя на следующий узел. Поле указателя последнего узла содержит адрес корневого элемента.

Основные действия, производимые над элементами ОЦС: Инициализация списка Добавление узла в список Удаление узла из списка Вывод элементов списка Взаимообмен двух узлов списка
Поскольку список является циклическим, реализация отдельной функции для удаления корня списка не требуется.
const int listOK = 0;const int listEmpty = 1;const int listNoMem = 2;const int listEnd = 3;
extern int ListError;typedef int BaseType;
typedef struct element *ptrEl;
struct element { BaseType data; ptrEl link; }element;
typedef struct { ptrEl start; ptrEl ptr; unsigned N; } List;
void InitList(List *L);void ListPut(List *L, BaseType E); void ListGet(List *L, BaseType *E);
void ListMove(List *L); void ListBegin(List *L); void ListDone(List *L); int ListEmpty(List *L);

21. Двухсвязный линейный список. Базовые операции. Интерфейс ДЛС

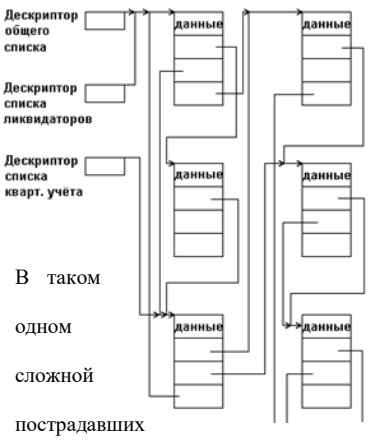
СЛС подразделяются на односвязные линейные списки (ОЛС) и двусвязные линейные списки (ДЛС). В. В ДЛС каждый элемент состоит из информационного поля и двух полей указателей, одно из которых содержит адрес следующего элемента в списке, а другое — предыдущего. Реализуется ДЛС, как правило, — с первым и последним фиктивным элементом. Введение фиктивных элементов позволяет упростить реализацию некоторых операций над ЛС.
Базовые операции: Инициализация - Включить элемент до рабочего указателя - Включить элемент после рабочего указателя-исключить элемент до рабочего указателя - Исключить элемент после рабочего указателя - Передвинуть рабочий указатель на шаг вперед/назад - Список пуст/не пуст - Сделать список пустым - Передвинуть рабочий указатель в начало/конец
Интерфейс:
const int DListOk=0; const int DListNoMem=1; const int DListEmpty=2; const int DListBegin=3; const int DListEnd=4;
extern int DListError; typedef int BaseType;
typedef struct element * ptrEl;
typedef struct element { BaseType data; ptrEl next; ptrEl pred; } element;
typedef struct{ ptrEl Start; ptrEl End; ptrEl ptr; } DList;
void InitDlist (DList *DL); void PredPut(DList *DL, BaseType E); void postPut(DList *DL, BaseType E); void predGet(DList *DL, BaseType* E); void postGet(DList *DL, BaseType* E); void DListMoveRigth (DList *DL); void DListMoveLeft (DList *DL); void ptrBegin (DList *DL); void ptrEnd (DList *DL); int DListisEmpty (DList *DL); void DoneDList(DList *DL);

22. СД типа «дек». Базовые операции. Интерфейс СД типа «дек».



Дек — линейная структура (последовательность), в которой операции включения и исключения элементов могут выполняться как с одного, так и с другого конца последовательности.
Дек — симбиоз стека и очереди, то есть дисциплинами обслуживания являются одновременно LIFO и FIFO.
Операции: Инициализация Включение элемента в начало / конец дека Исключение элемента из начала / конца дека Проверка: дек пуст / не пуст Удаление дека
Обычно для реализации дека используют двусвязный линейный список, то есть на основе готовых функций списка создают новые для дека.
static const int DackOk=0, DackEmpty=1, DackNotMem=2; static int DackError; //переменная ошибок
typedef t_dlist Dack; //дек отображается на ДЛС, базовый тип задается в реализации ДЛС
void InitDack(Dack *D); void DoneDack(Dack *D);
void PutDackB(Dack *D, t_base E); void PutDackE(Dack *D, t_base E); void GetDackB(Dack *D, t_base *E); void GetDackE(Dack *D, t_base *E); void ReadDackB(Dack *D, t_base *E); void ReadDackE(Dack *D, t_base *E); int EmptyDack(Dack *D);

23. Многосвязные списки (мультисписки)



В общем случае число указателей в связном списке может быть произвольным. В результате такого обобщения мы получаем многосвязный список. Рассмотрим несколько примеров таких списков.

Пример. Каждый элемент многосвязного списка может входить одновременно в несколько односвязных списков. Т.е. получается, что многосвязный список “прошит” в нескольких направлениях.

Например, необходимо организовать данные об абитуриентах, но кроме организации всех сведений, необходимо из общего списка выбрать абитуриентов, характеризующихся некоторыми общими данными (абитуриенты-медалисты, абитуриенты, проживающие в одном городе, абитуриенты, сдавшие экзамен на “отлично”).

В таком случае данные можно организовать в виде четырехсвязного списка, каждый элемент которого содержит все необходимые сведения, а также четыре указателя: первый связывает всех студентов; второй – проживающих в одном городе; и т.д. Для каждого из четырех односвязных списков имеется свой дескриптор, в который входит количество элементов, условия их размещения в списке. В программных системах, обрабатывающих объекты сложной структуры, могут решаться разные подзадачи, каждая из которых требует, возможно, обработки не всего пострадавших множества объектов, а лишь какого-то его подмножества. Так, например, в автоматизированной системе учета лиц, вследствие аварии на ЧАЭС, каждая запись об одном пострадавшем содержит более 50 полей в своей информационной части.

Решаемые же автоматизированной системой задачи могут потребовать выборки, например:

- участников ликвидации аварии;

· переселенцев из зараженной зоны;· лиц, состоящих на квартирном учете;· лиц с заболеваниями щитовидной железы;· и т.д., и т.п. (рисунок)

К достоинствам мультисписков помимо экономии памяти (при множестве списков информационная часть существует в единственном экземпляре) следует отнести также целостность данных - в том смысле, что все подзадачи работают с одной и той же версией информационной части и изменения в данных, сделанные одной подзадачей немедленно становятся доступными для другой подзадачи.

Каждая подзадача работает со своим подмножеством как с линейным списком, используя для этого определенное поле связок. Специфика мультисписка проявляется только в операции исключения элемента из списка. Исключение элемента из какого-либо одного списка еще не означает необходимости удаления элемента из памяти, так как элемент может оставаться в составе других списков. Память должна освобождаться только в том случае, когда элемент уже не входит ни в один из частных списков мультисписка. Обычно задача удаления упрощается тем, что один из частных списков является главным - в него обязательно входят все имеющиеся элементы. Тогда исключение элемента из любого неглавного списка состоит только в переопределении указателей, но не в освобождении памяти. Исключение же из главного списка требует не только освобождения памяти, но и переопределения указателей как в главном списке, так и во всех неглавных списках, в которые удаляемый элемент входил.

24. Классификация алгоритмов по временной сложности

В информатике недостаточно высказать утверждение о существовании некоторого объекта (в математике для этого пользуются теоремами о существовании), а также недостаточно найти конструктивное доказательство этого факта, т.е. алгоритм. В информатике необходимо, чтобы решение задачи можно было осуществить, используя конечный объем памяти и время, приемлемое для пользователя.

По временной сложности все задачи классифицируются следующим образом:

1 класс. Задачи класса P – это задачи, для которых известен алгоритм и временная сложность оценивается полиномиальной функцией f(N) = a k∗N + a k-1∗N 2 +…+a 0∗N k+1. Алгоритмы таких задач называют также “хорошими” алгоритмами. Этих алгоритмов мало, они образуют небольшой по мощности класс. Легко реализуются. Функция временной сложности для таких алгоритмов имеет вид O(Nk).

2 класс. Задачи класса E – это задачи, для которых алгоритм известен, но сложность таких алгоритмов O(f N), где f – константа. Задачи такого класса – это задачи построения всех подмножеств некоторого множества, задачи построения всех полных подграфов графа. При небольших N экспоненциальный алгоритм может работать быстрее полиномиального.

3 класс. На практике существуют задачи, которые не могут быть отнесены ни к одному из вышерассмотренных классов. Это прежде всего задачи, связанные с решением систем уравнений с целыми переменными, задачи определения цикла, проходящего через все вершины некоторого графа, задачи диагностики и т.д. Такие задачи независимы от компьютера, от языка программирования, но могут решаться человеком.

25. СД типа «таблица» прямого доступа. Недостатки

Применяется, если колво ключей невелико. Ключи различны. Для их хранения исп массив.

Таблица – это набор элементов одинаковой организации, каждый из которых можно представить в виде двойки <K, V>, где K – ключ, а V – тело (информационная часть) элемента. Ключ уникален для каждого элемента и используется для доступа при выполнении операций. Над таблицей определены следующие операции: Инициализация - Включение элемента - Исключение элемента с заданным ключом - Чтение элемента с заданным ключом - Изменение элемента с заданным ключом - Проверка – пуста ли таблица - Уничтожение таблицы

На абстрактном уровне таблица представляет собой множество. На физическом уровне таблица реализуется последовательной или связной схемой хранения. Располагаться таблица может в статической или динамической памяти. В зависимости от способа размещения элементов таблицы классифицируются на неупорядоченные, упорядоченные и хеш-таблицы.Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу - каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись, если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа эффективны в использовании, но область их применения весьма ограничена. Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех слотов в памяти, которые мы выделяем для хранения таблицы. Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве реальных задач, однако, размер пространства записей много меньше, чем пространства ключей.

26. СД типа «хеш-таблица». Требования к хеш-функциям. Разрешение коллизий с помощью цепочек. Анализ сложности

Хеш-таблица — это таблица, в которой положение адреса элемента определяется с помощью некоторой функции H (хеш-функции), аргументом которой является значение ключа элемента. Функция хеширования определяется как отображение

H:K->A, где K={k1,k2,…,km} – множество значений ключа; A={a1,a2,…,an} – адресное пространство; m<=n.

Для хранения элементов хеш-таблицы может быть использована дополнительная СД типа «массив». Доступ к записи с ключом k осуществляется непосредственно путем вычисления значение H(k). Таблицы, для которых существует и известна такая хеш-функция, называют хеш-таблицами с прямым доступом. Время выполнения операций поиска, включения, исключения, чтения и изменения не зависит от количества элементов в таблице и определяется временем вычисления значения хеш-функции и временем доступа к элементу массива. Для ускорения поиска элементов таблицы, ключи которых отображаются хеш-функцией в одно и то же значение a_i, в качестве элемента массива может быть использована СД типа БД. “Хорошая” хеш-функция – это функция, которая быстро вычисляется и минимизирует количество коллизий. Коллизия – это ситуация, когда для разных ключей адрес один и тот же, т.е. h(ki) = h(kj), где i не равно j. Для разрешения коллизий наиболее часто используют алгоритмы из двух основных групп: открытая адресация (методы двойного хеширования и линейного опробования) и метод цепочек. **Время выполнения операций над хеш-таблицей зависит от числа коллизий и времени вычисления значения хеш-функции.**

В методе цепочек элемент массива T с индексом a_i содержит цепочку элементов таблицы, ключи которых отображаются хеш-функцией в значение a_i. Для хранения таких цепочек элементов может быть использована дополнительная СД – линейный список (ПЛС или ОЛС), т.о. элемент массива T представляет собой линейный список. Алгоритмы включения и исключения элементов в такую таблицу очень просты:

1. Вычислить $a_i = H(k_j)$.
2. Включить/исключить элемент в линейный список $T[a_i]$

27. СД типа «хеш-таблица». Метод открытой адресации. Анализ сложности

Хеш-таблица — это таблица, в которой положение адреса элемента определяется с помощью некоторой функции H (хеш-функции), аргументом которой является значение ключа элемента. Функция хеширования определяется как отображение

H:K->A, где K={k1,k2,…,km} – множество значений ключа; A={a1,a2,…,an} – адресное пространство; m<=n.

Для хранения элементов хеш-таблицы может быть использована дополнительная СД типа «массив». Доступ к записи с ключом k осуществляется непосредственно путем вычисления значение H(k). Таблицы, для которых существует и известна такая хеш-функция, называют хеш-таблицами с прямым доступом. Время выполнения операций поиска, включения, исключения, чтения и изменения не зависит от количества элементов в таблице и определяется временем вычисления значения хеш-функции и временем доступа к элементу массива.

Для ускорения поиска элементов таблицы, ключи которых отображаются хеш-функцией в одно и то же значение a_i, в качестве элемента массива может быть использована СД типа БД.

“Хорошая” хеш-функция – это функция, которая быстро вычисляется и минимизирует количество коллизий. Коллизия – это ситуация, когда для разных ключей адрес один и тот же, т.е. h(ki) = h(kj), где i □ j. Для разрешения коллизий наиболее часто используют алгоритмы из двух основных групп: открытая адресация (методы двойного хеширования и линейного опробования) и метод цепочек.

Время выполнения операций над хеш-таблицей зависит от числа коллизий и времени вычисления значения хеш-функции.

В методе открытой адресации в качестве дополнительной СД используется массив, элементы которого могут содержать элементы таблицы. Каждый элемент массива имеет признак: содержит ли элемент массива элемент таблицы, содержал ли элемент массива элемент таблицы или элемент массива свободен. Для разрешения коллизий используется две хеш-функции: H1(k) и H2(a).

Алгоритм выполнения операций поиска, включения, исключения, чтения и изменения элемента таблицы с ключом k_j:

Вычислить $a_i = H1(k_j)$.

Если при включении в таблицу новой записи элемент массива a_i содержал ранее элемент таблицы, но в настоящее время не содержит, а при исключении содержит элемент таблицы с ключом k_j, то поиск завершен. В противном случае перейти к следующему пункту.

Вычислить $a_i = H2(k_j)$ и перейти к пункту 2.

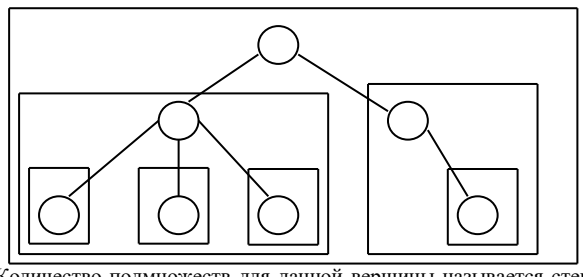
При включении новых элементов таблицы алгоритм остается конечным до тех пор, пока есть хотя бы один свободный элемент массива. При исключении элемента из таблицы алгоритм конечен, если таблица содержит элемент с заданным ключом. При невыполнении этих условий произойдет зацикливание.

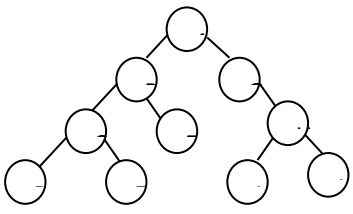
28. Нелинейные СД, деревья. Основные определения. Представления деревьев в связной памяти

Дерево – конечное непустое множество T, состоящее из одного и более узлов таких, что выполняются следующие условия:

1. Имеется один специально обозначенная вершина, называемая корнем данного дерева.
2. Остальные вершины (исключая корень) содержатся в m>=0 попарно не пересекающихся множествах T1, T2, …, Tm, каждое из которых в свою очередь является деревом. Деревья T1, T2, …, Tm называются поддеревьями данного корня.

Если подмножества T1, T2, …, Tm упорядочены, то дерево называют упорядоченным. Если два дерева считаются равными и тогда, когда они отличаются порядком, то такие деревья называются ориентированными деревьями. Конечное множество непересекающихся деревьев называется лесом.





Количество подмножеств для данной вершины называется степенью вершины. Если степень вершины равна нулю, то такая вершина называется листом. Уровень вершины – длина пути от корня до вершины. Максимальная длина пути от корня до вершины определяет высоту дерева.

Бинарное дерево – конечное множество элементов, которое может быть пустым, состоящее из корня и двух непересекающихся бинарных деревьев, причем поддеревья упорядочены: левое поддерево и правое поддерево.

БД – динамическая структура. Над СД БД определены следующие основные операции: инициализация, создание корня, запись данных, чтение данных, проверка – есть ли левый/правый сын, переход к левому/правому сыну, проверка, пустое ли дерево, удаление листа.

Различают три основных способа представления деревьев в связной памяти: стандартный, инверсный, смешанный.

При стандартном способе узлы, находящиеся на одном уровне являются братьями. Если же узел находится на более нижнем уровне, то он считается сыном.

При инверсном способе каждый узел дерева имеет указатель, показывающий на родителя.

Если же говорить о смешанном способе представления дерева в связной памяти, то здесь, как видно из названия, каждый узел включает указатели, указывающие как на сыновей, так и на родителя.

29. Алгоритмы прохождения n-арных деревьев в «ширину» и «глубину» (лекция)

Ширина: Задача обхода состоит в том, чтобы вывести информационную часть каждой вершины дерева. Порядок прохождения вершин определяет алгоритм обхода дерева.

В обходе в ширину сначала выписывается корень, , затем вершины первого уровня, затем второго и т.д. до последнего.

Алгоритм прохождения дерева в ширину:

<взять две очереди O1 и O2>;
<поместить корень в очередь O1>;
while <O1 или O2 не пуста> do
if <O1 не пуста> then {P – узел, находящийся в голове очереди O1}
begin
<исключить узел из очереди O1 и пройти его>;
<поместить все узлы, относящиеся к братьям этого узла P в очередь O2>;
end
else <O1=O2; O2=□> и повторяем цикл.

Глубина: Для прохождения n-арного дерева в прямом порядке (в глубину) необходимо выполнить n+1 операций:

Попасть в корень==Пройти в прямом порядке 1-е поддерево==…==n. Пройти в прямом порядке n-е поддерево.

Алгоритм рекурсивный, так как поддеревья обрабатываются так же, как и само дерево.

Для обхода дерева в прямом порядке можно использовать итеративный алгоритм. В этом алгоритме для возвращения к очередному сыну после прохождения поддерева используется стек для запоминания корня пройденного поддерева:

<пустой стек S>;
<пройти корень и включить его в стек S>;
while <стек S не пуст> do
begin {пусть P – узел, находящийся в вершине стека S}
if <не все сыновья узла P пройдены>
then <пройти старшего сына и включить его в стек S>
else begin
<исключить из вершины стека узел P>;
if <не все братья вершины P пройдены>
then <пройти старшего брата и включить его в стек S> end; end;

30. Алгоритмы представления n-арных деревьев в виде бинарных. Представление бинарных деревьев в памяти. Прошитые бинарные деревья

Дерево и лес любого вида можно преобразовать единственным образом в эквивалентное бинарное дерево.

Правило построения бинарного дерева из любого дерева:

В каждом узле оставить только ветвь к старшему сыну (вертикальное соединение);

Соединить горизонтальными ребрами всех братьев одного отца;

Таким образом перестроить дерево по правилу: левый сын - вершина, расположенная под данной; правый сын - вершина, расположенная справа от данной (т.е. на ярусе с ней).

Развернуть дерево таким образом, чтобы все вертикальные ветви отображали левых сыновей, а горизонтальные - правых.

В результате преобразования любого дерева в бинарное получается дерево в виде левого поддерева, подвешенного к корневой вершине.

В процессе преобразования правый указатель каждого узла бинарного дерева будет указывать на соседа по уровню. Если такового нет, то правый указатель NIL. Левый указатель будет указывать на вершину следующего уровня. Если таковой нет, то указатель устанавливается на NIL.

1. изобразить корень дерева 2. Старший сын по верстикали 3. По горизонтали его брат 4. 1-2-3 для след уровня.

ПРОШИВКА БИНАРНЫХ ДЕРЕВЬЕВ. Под прошивкой дерева понимается замена по определенному правилу пустых указателей на сыновей указателями на последующие узлы, соответствующие обходу.

В бинарном дереве имеется много указателей со значением NIL. В дереве с N вершинами имеется (N+1) указателей типа NIL. Это незанятое пространство можно использовать для изменения представления деревьев. Пустые указатели заменяются указателями - "нитями", которые адресуют вершины дерева, расположенные выше. При этом дерево прошивается с учетом определенного способа обхода. Например, если в поле left некоторой вершины P стоит NIL, то его можно заменить на адрес, указывающий на предшественника P, в соответствии с тем порядком обхода, для которого прошивается дерево. Аналогично, если поле right пусто, то указывается преемник P в соответствии с порядком обхода. Поскольку после прошивания дерева поля left и right могут характеризовать либо структурные связи, либо "нити", возникает необходимость различать их, для этого вводятся в описание структуры дерева характеристики левого и правого указателей (FALSE и TRUE).

Таким образом, прошитые деревья быстрее обходятся и не требуют для этого дополнительной памяти (стек), однако требуют дополнительной памяти для хранения флагов нитей, а также усложнены операции включения и удаления элементов дерева.

В случае представления бинарного дерева в виде узлов, содержащих информационное поле и два поля связи, количество полей связи, имеющих значения nil, всегда больше числа связей, указывающих на реально существующие узлы. Поэтому часто такой способ хранения деревьев оказывается неэффективным с точки зрения использования памяти, особенно, если размер информационного поля сопоставим с размером указателя.

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется нитью. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. Операция, заменяющая пустые указатели на нити, называется прошивка. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, при просмотре в симметричном порядке, то бинарное дерево называется симметрично прошитым справа.Похожим образом может быть определено бинарное дерево, симметрично прошитое слева:дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить. На рис. 6.1 показано дерево, симметрично прошитое справа. На нем пунктирными линиями обозначены прошивочные нити.

31. Формирование идеально сбалансированного дерева. Алгоритм. Недостатки

Поскольку максимальный путь до листьев дерева определяется высотой дерева, то при заданном числе n узлов дерева стремятся построить дерево минимальной высоты. Этого можно достичь, если размещать максимально возможное количество узлов на всех уровнях, кроме последнего. В случае двоичного дерева это достигается таким образом, что все поступающие при построении дерева узлы распределяются поровну слева и справа от каждого узла.

Говорят, что бинарное дерево идеально сбалансировано, если для каждого его узла количество узлов в левом и правом поддеревьях различается не более чем на 1.

Создание идеально сбалансированного дерева не вызывает затруднений. Если число узлов и известно и дана последовательность значений поля данных вершин a[0], a[1],...,a[n - 1], то можно использовать следующий рекурсивный алгоритм построения идеально сбалансированного дерева:

1. Начиная с a[0], берем очередное значение a[i] в качестве значения корня дерева (поддерева).
2. Строим левое поддерево с n1 = n/2 узлами тем же способом.
3. Строим правое поддерево с nr = n - nl - 1 узлами.

Таким образом, значение a[0] окажется в корне дерева, а именно на него будет ссылаться указатель дерева, a[1],...a[nl] значения попадут в левое поддерево, остальные — в правое поддерево. Следовательно, распределение значений по узлам дерева полностью определяется сходной последовательностью данных. Идеально сбалансированное бинарное дерево, построенное из последовательности ключей: 9,17,20,16,12,21,6,3,11,4,19,14,13,1,5,2,8,18,7,10,15, будет иметь следующий вид.

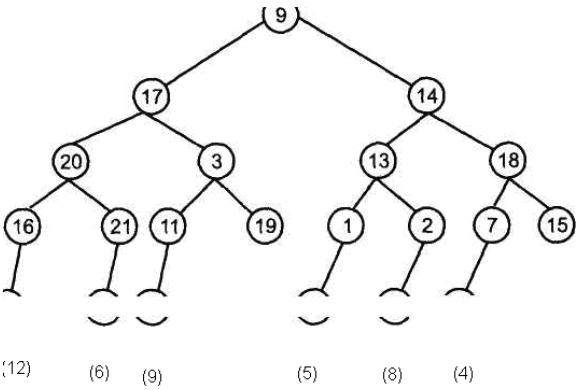
```
typedef struct RECORD    /* Структура элемента дерева */
{ int DATA;         /* данные узла — здесь только ключ */
RECORD *LPTR,*RPTR;   /* левый и правый указатели на поддерево*/
};
RECORD *ROOT;    /* Указатель корня дерева */
static int ndi=0;
/*последовательный номер включаемого элемента ,
удобства работы пользователя и контроля наличия дерева. Перед
каждым обращением к функциям tree, preorder в вызывающей функции
переменную ndi нужно обнулять. */


RECORD* tree(int k);        /* функция создания дерева, к - число узлов*/
int preorder(RECORD *)          ;     /* функция нисходящего обхода дерева*/
void p(RECORD*);   /* Печать данных в узле */
void lnData(RECORD*);   /* Ввод данных узла*/
/*=============================================9
/* ГЛАВНАЯ ФУНКЦИЯ */
main()
{int i,n;
printf("\n\n РАБОТА С ИДЕАЛЬНО СБАЛАНСИРОВАННЫМ БИНАРНЫМ ДЕРЕВОМ");
printf("\n Создание дерева");
printf("\n Введите число узлов дерева =>");
scanf("%d",&n);
/* Создание идеально сбалансированного дерева */
ndi=O;
if( (ROOT=tree(n))==NULL)
     { printf("\n Дерево не создано");
       getch();         /* return -1; */
}
printf("\n Обход дерева с выводом данных в узлах");
printf("\n Нисходящий обход дерева:\n");
ndi=O;    preorder(ROOT); getch();
printf("\n Конец\n"); getch(); return 0;
}
/* ПОСТРОЕНИЕ БИНАРНОГО ИДЕАЛЬНО СБАЛАНСИРОВАННОГО ДЕРЕ-ВА*/
RECORD* tree(int k)
{ RECORD *newnode;
int nl,nr;
if (k≤0)                    /* k<0 - ошибочное задание параметра */
newnode =0;
  else
   { nl = k/2;
     alloc(1,sizeof(RECORD));
InData(newnode);
newnode.LPTR=tree(nl);
newnode.RPTR=tree(nr);

}
return newnode;
}
```

```
/* НИСХОДЯЩИЙ ОБХОД ДЕРЕВА */
int preorder(RECORD *T)
  { if(T==NULL && ndi==0)
     { printf("\n Дерево пустое");
getch(); return -1;
}
if (T!=0)
{ ndi++; p(T);       /* обработка данных в узле */
preorder(T.LPTR);
preorder(T.RPTR);
}
return 0;
}


//* ОБРАБОТКА ДАННЫХ УЗЛА ДЕРЕВА */
void p(RECORD *B)
{ printf(" %d ",B.DATA);
}
/* ВВОД ДАННЫХ УЗЛА ДЕРЕВА */
void lnData(RECORD* d)
   {       printf("Введите %d-й ключ =>",++ndi);
           scanf("%d",&d.DATA);
}
```

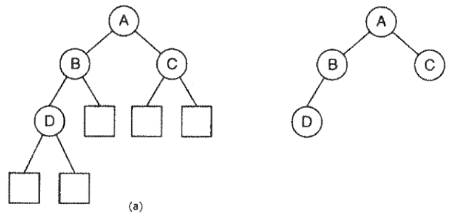


32. Применение бинарных деревьев в алгоритмах поиска. Таблица как отображение на бинарное дерево

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

33. Операция включения в бинарное дерево. Анализ сложности

Двоичные деревья представляют эффективный способ поиска. Двоичное дерево представляет собой структурированную коллекцию узлов. Коллекция может быть пустой и в этом случае мы имеем пустое двоичное дерево. Если коллекция непуста, то она подразделяется на три раздельных семейства узлов: корневой узел n (или просто корень), двоичное дерево, называемое левым поддеревом для n, и двоичное дерево, называемое правым поддеревом для n. На рис. 1а узел, обозначенный буквой A, является корневым, узел B называется левым потомком A и является корнем левого поддерева A, узел C называется правым потомком A и является корнем правого поддерева A.



Для включения нового элемента в двоичное дерево поиска вначале нужно определить его точное положение — а именно внешний узел, который должен быть заменен путем отслеживания пути поиска элемента, начиная с корня. Кроме сохранения указателя n на текущий узел мы будем хранить указатель p на предка узла n. Таким образом, когда n достигнет некоторого внешнего узла, p будет указывать на узел, который должен стать предком нового узла. Для осуществления включения узла мы создадим новый узел, содержащий новый элемент, и затем свяжем предок p с этим новым узлом (рис. 3).

Если дерево пусто, заменить его на дерево с одним корневым узлом ((K,V), null, null) и остановиться.
Иначе сравнить K с ключом корневого узла X.
Если K>X, циклически добавить (K,V) в правое поддерево T.
Если K<X, циклически добавить (K,V) в левое поддерево T.
Если K=X, заменить V текущего узла новым значением (хотя можно и организовать список значений V, но это другая тема).

Анализ

Дерево называется идеально сбалансированным, если число вершин в его ЛПД и ППД отличается не более чем на 1. Приблизительно подсчитаем высоту h идеально сбалансированного дерева с n вершинами:

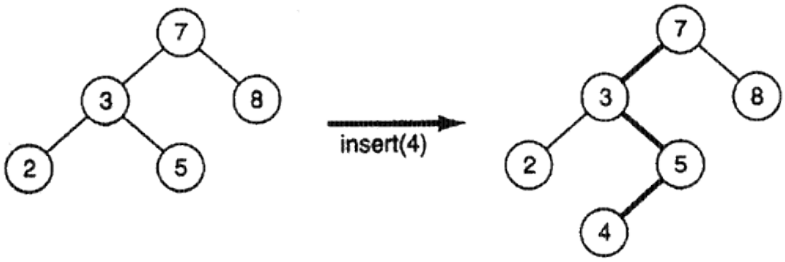
$20+21+22+...+2h = \frac{2^{h+1}-1}{2-1} \approx n$, откуда $2h \approx \frac{n+1}{2}$, т.е. при достаточно больших значениях n:
h = O (log2 (n)).

При построении дерева поиска с помощью алгоритма поиска с включением заранее неизвестно, как будет расти дерево и какую форму оно примет.

В лучшем случае дерево поиска окажется идеально сбалансированным. Высота этого дерева определит максимальную сложность операций поиска, включения и удаления элементов, которая составит, таким образом, Tmax(n) = O (log2 (n)).

Однако в худшем случае, когда все поступающие из входного потока ключи идут в порядке неубывания (или невозрастания), дерево выродится в обычный линейный список и максимальная сложность операций поиска, включения и удаления элементов составит Tmax(n)=O(n), т.е. все преимущества дерева поиска по сравнению с линейным списком будут утрачены.

Доказано, что затраты на перестройку случайного дерева в идеально сбалансированное себя не оправдывают, т.к. они гасят выигрыш в длине пути.

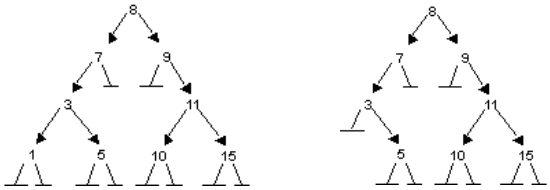


34. Операция исключения из бинарного дерева. Анализ сложности

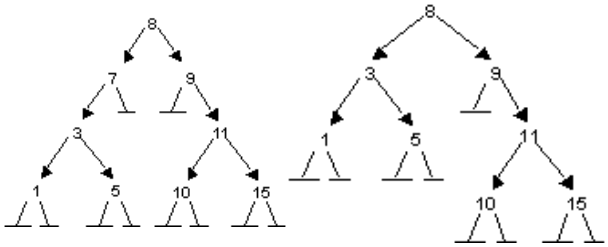
Исключение узлов из дерева поиска следует проводить так, чтобы для каждого из оставшихся узлов сохранялось свойство: значения в левом поддереве были меньше его значения, а значения в правом поддереве - больше (либо равны). То есть свойство отсортированности дерева должно сохраняться.

При исключении узла из дерева поиска возможны три случая. Первые два из них простые и затрагивают только непосредственного предка удаляемого узла, третий случай сложный и затрагивает несколько узлов.

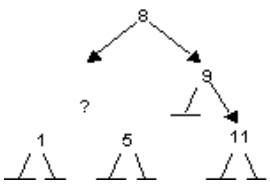
Случай 1 (простой). Исключаемый узел не имеет поддеревьев (llink=nil, rlink=nil). В этом случае узел просто удаляется, а соответствующая связь у непосредственного предка становится равной nil. Например, из следующего дерева удаляется узел 1:



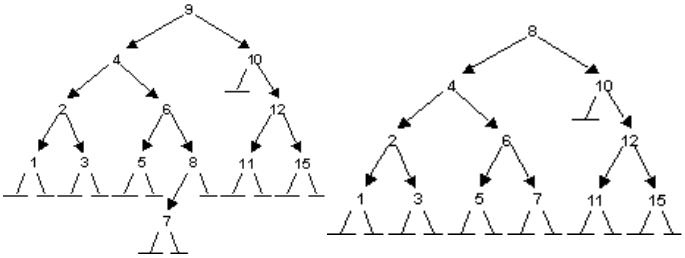
Случай 2 (простой). Исключаемый узел имеет только одно поддерево (llink<>nil и rlink=nil) или (llink=nil и rlink<>nil). В этом случае узел удаляется, а его единственное поддерево перемещается на место удаленного узла (поднимается на уровень выше). Например, из следующего дерева удаляется узел 7:



Случай 3 (сложный). Исключаемый узел имеет оба поддерева (llink<>nil и rlink<>nil). Просто удалить этот узел нельзя - останутся два его поддерева, переместить на уровень вверх мы можем только одно. Например, если из следующего дерева удалит узел 3, то неясно, что делать с его поддеревьями:



Решение заключается в следующем: на место исключаемого узла поместить либо самый правый узел его левого поддерева (предшественник при b-обходе), либо самый левый узел его правого поддерева (преемник при b-обходе). Например, удаляя из следующего дерева узел 9, имеющий оба поддерева, помещаем на его место предшественника - узел 8:



35. Применение бинарных деревьев. Алгоритмы прохождения бинарных деревьев

Применение: Сортировки(пирамидальная) Поиск элемента

Алгоритмы прохождения

Обход бинарного дерева «в глубину» (в прямом порядке)

Чтобы пройти БД в прямом порядке нужно выполнить следующие три операции:

1. Попасть в корень. 2. Пройти в прямом порядке левое поддерево. 3. Пройти в прямом порядке правое поддерево.

Это рекурсивный алгоритм, т.к. поддеревья обрабатываются точно так же, как и БД. Нерекурсивный выход произойдет при достижении пустого дерева. Применив алгоритм к БД на рис.21, получим последовательность: ABCDEFGHIJ.

Для обхода БД в прямом порядке можно использовать итеративный алгоритм. В этом алгоритме, для того, чтобы вернуться к правому сыну после прохождения левого поддерева используется стек для запоминания корня пройденного левого поддерева.

Итеративный алгоритм прохождения БД «в глубину»:

1. Инициализировать стек. 2. Пройти корень Т и включить его адрес в стек. 3. Если стек пуст, то перейти к п.5. Если есть левый сын вершины Т, то пройти его и занести его адрес в стек, иначе извлечь из стека адрес вершины Т и если у нее есть правый сын, то пройти его и занести в стек. 4. Перейти к п.3. 5. Конец алгоритма.

Обход бинарного дерева «в ширину» (по уровням)

В этом обходе сначала выписывается корень, затем вершины первого уровня, второго и т.д. до последнего. Выполнив обход БД на рис.21, получим последовательность: ABGCFHDEIJ.

В алгоритме обхода БД «в ширину» будем использовать очередь, в которую будут помещаться адреса вершин в порядке обхода.

Алгоритм прохождения БД «в ширину»:

1. Инициализировать очередь. 2. Занести адрес корня Т в очередь. 3. Если очередь пуста, то перейти к п.5. Извлечь из очереди адрес вершины Т и пройти ее. Если у нее есть левый сын, то занести его адрес в очередь. Если у нее есть правый сын, то занести его адрес в очередь. 4. Перейти к п.3. 5. Конец алгоритма.

Обход бинарного дерева в симметричном порядке

Для прохождения БД в симметричном порядке нужно выполнить следующие действия:

1. Пройти в симметричном порядке левое поддерево. 2. Попасть в корень. 3. Пройти в симметричном порядке правое поддерево.

Выполнив обход в симметричном порядке БД на рис.21, получим последовательность: DCEBFAGIHJ.

Для обхода БД в симметричном порядке, также как и для обхода в прямом порядке, можно применить итеративный алгоритм с использованием стека.

Итеративный алгоритм прохождения БД в симметричном порядке:

1. Инициализировать стек. 2. Адрес корня Т включить в стек. 3. Если стек пуст, то перейти к п.5. Если есть левый сын вершины Т, то занести его адрес в стек, иначе извлечь из стека адрес вершины Т, пройти ее и если у нее есть правый сын, то занести его в стек. 4. Перейти к п.3. 5. Конец алгоритма.

Обход бинарного дерева в обратном порядке

Для прохождения БД в обратном порядке нужно выполнить следующие действия:

1. Пройти в обратном порядке левое поддерево.2. Пройти в обратном порядке правое поддерево.3. Попасть в корень.

36. Коды Хоффмана. Общие сведения. Префиксные коды

Один из первых алгоритмов эффективного кодирования информации был предложен Д. А. Хаффманом в 1952 году. Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево).[1]

- Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
- Выбираются два свободных узла дерева с наименьшими весами.
- Создается их родитель с весом, равным их суммарному весу.
- Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
- Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.
- Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Определение. Код, обладающий тем свойством, что никакое кодовое слово не является началом (префиксом) другого кодового слова, называется префиксным.

Пре́фиксный код в теории кодирования — код со словом переменной длины, имеющий такое свойство (выполнение условия Фано): если в код входит слово a, то для любой непустой строки b слова ab в коде не существует. Хотя префиксный код состоит из слов разной длины, эти слова можно записывать без разделительного символа.

Например, код, состоящий из слов 0, 10 и 11, является префиксным, и сообщение 01001101110 можно разбить на слова единственным образом:

0 10 0 11 0 11 10

Код, состоящий из слов 0, 10, 11 и 100, префиксным не является, и то же сообщение можно трактовать несколькими способами.

0 10 0 11 0 11 10

0 100 11 0 11 10

37. Коды Хоффмана. Алгоритмы построения кода Хоффмана. Анализ сложности

Один из первых алгоритмов эффективного кодирования информации был предложен Д. А. Хаффманом в 1952 году. Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (H-дерево).[1]

- Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
- Выбираются два свободных узла дерева с наименьшими весами.
- Создается их родитель с весом, равным их суммарному весу.
- Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
- Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.
- Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Допустим, у нас есть следующая таблица частот:

Символ	А	Б	В	Г	Д
Частота	15	7	6	6	5

Этот процесс можно представить как построение дерева, корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом.

Символ	А	Б	В	Г	Д
Код	0	100	101	110	111

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

При этом общая длина сообщения, состоящего из приведённых в таблице символов, составит 87 бит (в среднем 2,2308 бита на символ). При использовании равномерного кодирования общая длина сообщения составила бы 117 бит (ровно 3 бита на символ). Заметим, что энтропия источника, независимым образом порождающего символы с указанными частотами, составляет ~2,1858 бита на символ, то есть избыточность построенного для такого источника кода Хаффмана, понимаемая как отличие среднего числа бит на символ от энтропии, составляет менее 0,05 бит на символ.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Во-первых, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частот, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частот, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частот и H-дерева), другого для собственно кодирования. Во-вторых, избыточность кодирования обращается в ноль лишь в тех случаях, когда вероятности кодируемых символов являются обратными степенями числа 2. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

Рассмотрим кодирование по Хаффману более подробно. Предположим, что вероятности (их заменят частоты) всех символов алфавита уже подсчитаны одним из вышеописанных способов. Тогда:

1. Выписываем в ряд все символы алфавита в порядке убывания вероятностей (частоты) их появления в потоке данных (для удобства построения дерева);
2. Объединяем два символа с наименьшими вероятностями в новый составной символ, вероятность которого определяется как сумма вероятностей составляющих его символов. Последовательно повторяем эту операцию до образования единственного составного символа (корня). В результате получается дерево символов, каждый узел которого имеет суммарную вероятность всех объединённых им узлов.
3. Прослеживаем путь от каждого листа дерева к корню, помечая направление движения к каждому узлу (например, вверх/направо –1, вниз/налево - 0). При этом не важен конкретный вид разметки «ветвей» дерева (т.е. помечать направление вверх/направо –1, вниз/налево – 0, или наоборот), но важно придерживаться выбранного способа разметки ко всем «ветвям» дерева.
4. Получившиеся двоичные комбинации, записанные от конца к началу и формируют коды Хаффмана.

Полученный коэффициент сжатия подсчитывается по следующей формуле

$$k_{сж} = \frac{n}{\sum_i p(X_i) n_H(X_i)}, \quad (2)$$

где

n – количество бит, необходимое для кодирования символов алфавита фиксированным числом разрядов;

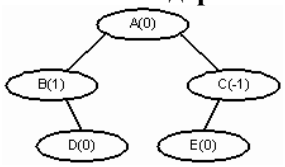
$p(X_i)$ - вероятность (частота) повторения символа X_i во входном потоке;

$n_H(X_i)$ - количество бит в коде Хаффмана для символа X_i .

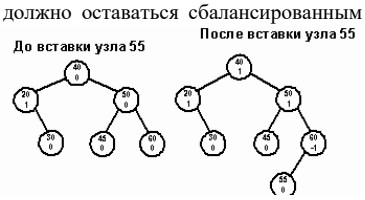
38. Сбалансированные деревья. АВЛ-деревья. Определение АВЛ дерева

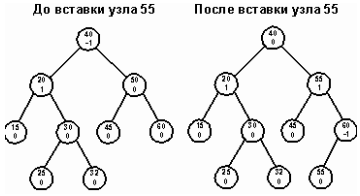
Дерево является СБАЛАНСИРОВАННЫМ тогда и только тогда, когда для каждого узла высота его двух поддеревьев различается не более, чем на 1.
Существуют алгоритмы, выполняющие над сбалансированным деревом операции вставки и удаление элементов таким образом, что сбалансированность дерева поддерживается. Эти алгоритмы достаточно сложны и затратны. Ниже приводится алгоритм балансировки двоичного дерева, упорядоченного для обхода слева направо.
1).Выполнить обход дерева слева направо и выбрать элементы дерева в упорядоченный линейный список.
2).Выбрать элемент, находящийся в середине этого списка, назначить его корнем нового, сбалансированного дерева.
3).Проделать операции пп 2- 4 с левым подсписком и сформировать левое поддерево
4). Проделать операции пп 2- 4 с правым подсписком и сформировать правое поддерево
АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются. Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: *для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу*. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log2(n + 1)$ до $1.44 \log2(n + 2) − 0.328$. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированнуюлогарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

39. АВЛ-деревья. Операции включение/исключения из АВЛ-дерева

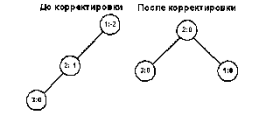


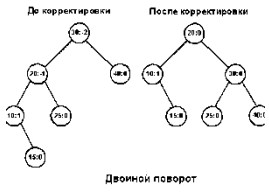
АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются. Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log2(n + 1)$ до $1.44 \log2(n + 2) − 0.328$. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированнуюлогарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю. ***
АВЛ-дерево представляет собой нелинейную списковую структуру, похожую на бинарное дерево поиска, с одним дополнительным условием: дерево должно оставаться сбалансированным по высоте после каждой операции вставки или удаления.





Поэтому вводится показатель сбалансированности для каждого узла. Его можно вычислять каждый раз при включении (исключении) элементов в (из) дерево или хранить вместе с узлом и при необходимости корректировать. Показатель сбалансированности может быть равным 0 (поддерево сбалансировано полностью), -1 (перевешивает левое поддерево) и 1 (перевешивает правое поддерево). Если показатель сбалансированности отличается от этих значений, то возникла ситуация разбалансировки и требуется восстановление сбалансированности.





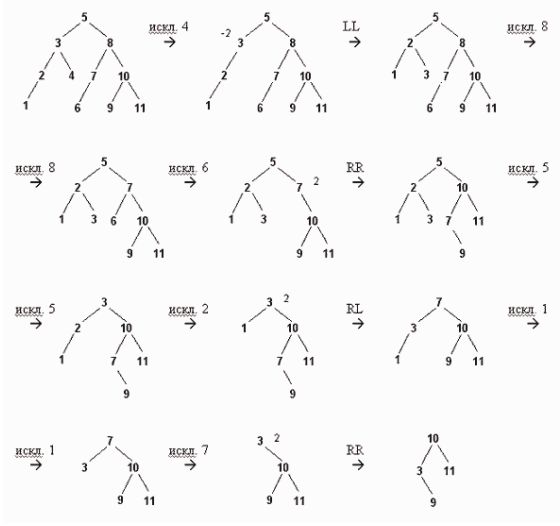
Алгоритм avl-вставки: Процесс вставки почти такой же, что и для бинарного дерева поиска. Осуществляется рекурсивный спуск по левым и правым сыновьям, пока не встретится пустое поддерево, а затем производится пробная вставка нового узла в этом месте. В течение этого процесса мы посещаем каждый узел на пути поиска от корневого к новому элементу.

Включение в АВЛ-дерево

После включения нового узла в АВЛ-дерево оно должно оставаться сбалансированным. Рассмотрим, в каких случаях потребуется балансировка дерева после включения узла. Во всех случаях будем указывать показатель сбалансированности корневого узла

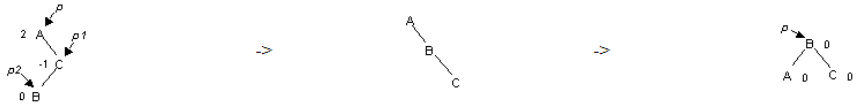
До включения (дерево АВЛ-сбалансировано)	После включения	
	В левое поддерево	В правое поддерево
$h_R = h_L$ $h_R - h_L = 0$	или $h_R < h_L$ $h_R - h_L = -1$ критерий сбалансированности не нарушен	или $h_R > h_L$ $h_R - h_L = 1$ критерий сбалансированности не нарушен
$h_R < h_L$ $h_R - h_L = -1$	$h_R < h_L$ $h_R - h_L = -2$ критерий сбалансированности нарушен, нужна балансировка	$h_R = h_L$ $h_R - h_L = 0$ критерий сбалансированностине нарушен
$h_R > h_L$ $h_R - h_L = 1$	$h_R = h_L$ $h_R - h_L = 0$ критерий сбалансированностине нарушен	$h_R > h_L$ $h_R - h_L = 2$ критерий сбалансированности нарушен, нужна балансировка

Поскольку процесс рекурсивный, обработка узлов ведется в обратном порядке. При этом показатель сбалансированности родительского узла можно скорректировать после изучения эффекта от добавления нового элемента в одно из поддеревьев. Необходимость корректировки определяется для каждого узла, входящего в поисковый маршрут. Есть три возможных ситуации. В двух первых случаях узел сохраняет сбалансированность и реорганизация поддеревьев не требуется. Нужно лишь скорректировать показатель сбалансированности данного узла. В третьем случае разбалансировка дерева требует одинарного или двойного поворотов узлов.
Случай 1. Узел на поисковом маршруте изначально является сбалансированным (balanceFactor = 0). После вставки в поддерево нового элемента узел стал перевешивать влево или вправо в зависимости от того, в какое поддерево была произведена вставка. Если элемент вставлен в левое поддерево, показателю сбалансированности присваивается -1, а если в правое, то 1. Например, на пути 40-50-60 каждый узел сбалансирован. После вставки узла 55 показатели сбалансированности изменяются (рисунок 2).
Случай 2. Одно из поддеревьев узла перевешивает, и новый узел вставляется в более легкое поддерево. Узел становится сбалансированным. Сравните, например, состояния дерева до и после вставки узла 55 (рисунок 3).
Случай 3. Одно из поддеревьев узла перевешивает, и новый узел помещается в более тяжелое поддерево. Тем самым нарушается условие сбалансированности, так как balanceFactor выходит за пределы -1..1. Чтобы восстановить равновесие, нужно выполнить поворот.
Рассмотрим пример. Предположим, дерево разбалансировалось слева и мы восстанавливаем равновесие, вызывая одну из функций поворота вправо. Разбалансировка справа влечет за собой симметричные действия.
Сказанное иллюстрируется рисунком 4.
Удаление узлов
Также как и вставку узла, его удаление удобно задать рекурсивно. Пусть x – удаляемый узел, тогда если x – лист (терминальный узел), то алгоритм удаления сводиться к простому исключению узла x, и подъему к корню с переопределением balance factor’ов узлов. Если же x не является листом, то он либо имеет правое поддерево, либо не имеет его. Во втором случае, из свойства АВЛ-дерева, следует, что левое поддерево имеет высоту 1, и здесь алгоритм удаления сводиться к тем же действиям, что и при терминальном узле. Остается ситуация когда у x есть правое поддерево. В таком случае нужно в правом поддереве отыскать следующий по значению за x узел y, заменить x на y, и рекурсивно вернуться к корню, переопределяя коэффициенты сбалансированности узлов. Из свойства двоичного дерева поиска следует, что узел y имеет наименьшее значение среди всех узлов правого поддерева узла x.



Исключение узла из АВЛ-дерева производится так же, как и из обычного дерева поиска, но после этого может возникнуть необходимость проведения балансировки. При этом если включение в АВЛ-дерева может потребовать не более одного поворота, то исключение может вызвать несколько поворотов на пути от удаляемого узла до корня дерева.
В качестве примера рассмотрим последовательное исключение узлов 4, 8, 6, 5, 2, 1, 7 из следующего дерева:
Процедура исключения из АВЛ-дерева.

Двойной RL-поворот. Выполняется, когда <перевес> идет по пути R-L от узла с нарушенной балансировкой.



```
    p1 := p^.rlink;
    p2 := p1^.llink;
p1^.llink := p2^.rlink;
    p2^.rlink := p1;
 p^.rlink := p2^.llink;
    p2^.llink := p;
       p := p2;
```

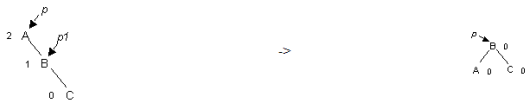
Таким образом, есть 4 варианта нарушения балансировки:

Балансировка выполняется с помощью действий, называемых поворотами узлов. Рассмотрим алгоритмы поворотов, используя указатели p, p1, p2 и считая, что p указывает на узел с нарушенной балансировкой.

Одинарный LL-поворот. Выполняется, когда <перевес> идет по пути L-L от узла с нарушенной балансировкой.

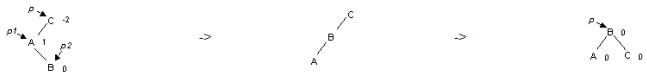
```
    p1 := p^.llink;
p^.llink := p1^.rlink;
   p1^.rlink := p;
      p := p1;
```

Одинарный RR-поворот. Выполняется, когда <перевес> идет по пути R-R от узла с нарушенной балансировкой.



```
    p1 := p^.rlink;
p^.rlink := p1^.llink;
   p1^.llink := p;
      p := p1;
```

Двойной LR-поворот. Выполняется, когда <перевес> идет по пути L-R от узла с нарушенной балансировкой.



```
    p1 := p^.llink;
    p2 := p1^.rlink;
p1^.rlink := p2^.llink;
    p2^.llink := p1;
p^.llink := p2^.rlink;
    p2^.rlink := p;
       p := p2;
```

Процедура исключения из АВЛ-дерева использует две вспомогательные процедуры: balanceL (балансировка после удаления из левого поддерева) и balanceR (балансировка после удаления из правого поддерева).
Видно, что исключение из АВЛ-дерева еще сложнее, чем включение. Это еще раз подтверждает то, что АВЛ-деревья целесообразно использовать в тех случаях, когда поиск узлов в дереве происходит гораздо чаще, чем включение и исключение улов.

40. Оптимальные деревья поиска. ???

Дерево поиска называется оптимальным, если его цена минимальна. То есть оптимальное бинарное дерево поиска – это бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных.
Существует подход построения оптимальных деревьев поиска, при котором элементы вставляются в порядке уменьшения частот, что дает в среднем неплохие деревья поиска. Однако этот подход может дать вырожденное дерево поиска, которое будет далеко от оптимального. Еще один подход состоит в выборе корня k таким образом, чтобы максимальная сумма вероятностей для вершин левого поддерева или правого поддерева была настолько мала, насколько это возможно. Такой подход также может оказаться плохим в случае выбора в качестве корня элемента с малым значением pk.
Существуют алгоритмы, которые позволяют построить оптимальное дерево поиска. Однако такие алгоритмы имеют временную сложность порядка O(n2). Таким образом, создание оптимальных деревьев поиска требует больших накладных затрат, что не всегда оправдывает выигрыш при быстром поиске.

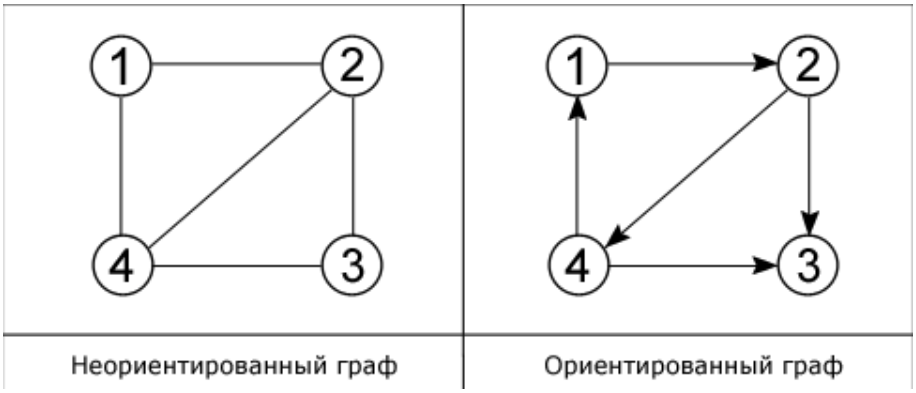
Встречаются ситуации, в которых можно получить информацию о вероятности обращений к отдельным ключам. Обычно в таких случаях дерево поиска строится один раз, имеет неизменяемую структуру, в него не включаются новые ключи, и из него не исключаются существующие ключи. Примером соответствующего приложения является сканер компилятора, одной из задач которого является определение принадлежности очередного идентификатора к набору ключевых слов языка программирования. На основе сбора статистики при многочисленной компиляции программ можно получить достаточно точную информацию о частотах поиска по отдельным ключам.
Пусть дерево поиска содержит n вершин, и обозначим через pi вероятность обращения к i-той вершине, содержащей ключ ki. Сумма всех pi, естественно, равна 1. Постараемся теперь организовать дерево поиска таким образом, чтобы обеспечить минимальность общего числа шагов поиска, подсчитанного для достаточно большого количества обращений. Будем считать, что корень дерева имеет высоту 1 (а не 0, как раньше), и определим взвешенную длину пути дерева как сумму pi*hi (1<=i<=n), где hi - длина пути от корня до i-той вершины. Требуется построить дерево поиска с минимальной взвешенной длиной пути.
В качестве примера рассмотрим возможности построения дерева поиска для трех ключей 1, 2, 3 с вероятностями обращения к ним 1/7, 2/7 и 4/7 соответственно (рисунок 4.15).
Посчитаем взвешенную длину пути для каждого случая. В случае (a) взвешенная длина пути P(a) = 1*4/7 + 2*2/7 + 3*1/7 = 11/7. Аналогичные подсчеты дают результаты P(b)=12/7; P(c)=12/7; P(d)=15/7; P(e)=17/7. Следовательно, оптимальным в интересующем нас смысле оказалось не идеально сбалансированное дерево (c), а вырожденное дерево (a).
На практике приходится решать несколько более общую задачу, а именно, при построении дерева учитывать вероятности неудачного поиска, т.е. поиска ключа, не включенного в дерево. В частности, при реализации сканера желательно уметь эффективно распознавать идентификаторы, которые не являются ключевыми словами. Можно считать, что поиск по ключу, отсутствующему в дереве, приводит к обращению к "специальной" вершине, включенной между реальными вершинами с меньшим и большим значениями ключа соответственно. Если известна вероятность qj обращения к специальной j-той вершине, то к общей средней взвешенной длине пути дерева необходимо добавить сумму qj*ej для всех специальных вершин, где ej - высота j-той специальной вершины.

Взвешенная длина пути – сумма всех путей от корня к каждому узлу умноженное на P: сумма Pi*hi, где h – уровень узла.
Ставится задача найти мин взвешенную длину пути: ТУТ РИСУНКИ))))))))))

41. Алгоритм Гилберта-Мура построения оптимального дерева поиска

42. СД типа «граф». Основные определения

Граф – совокупность точек, соединенных линиями. Точки называются вершинами (узлами), а линии – ребрами (дугами).



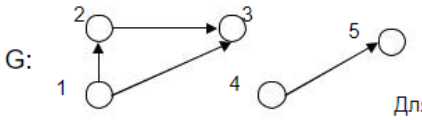
Как показано на рисунке различают два основных вида графов: ориентированные и неориентированные. В первых ребра являются направленными, т. е. существует только одно доступное направление между двумя связными вершинами, например из вершины 1 можно пройти в вершину 2, но не наоборот. В неориентированном связном графе из каждой вершины можно пройти в каждую и обратно. Частный случай двух этих видов – смешанный граф. Он характерен наличием как ориентированных, так и неориентированных ребер.

Степень входа вершины – количество входящих в нее ребер, степень выхода – количество исходящих ребер.

Ребра графа необязательно должны быть прямыми, а вершины обозначаться именно цифрами, так как показано на рисунке. К тому же встречаются такие графы, ребрам которых поставлено в соответствие конкретное значение, они именуются взвешенными графами, а это значение – весом ребра. Когда у ребра оба конца совпадают, т. е. ребро выходит из вершины F и входит в нее, то такое ребро называется петлей.

Компонентами неорграфа являются, соответственно: ребро, цепь, цикл. Цепь – непрерывная последовательность ребер между парой вершин неориентированного графа. Неориентированный граф называют связным, если любые две его вершины можно соединить цепью. Если же граф – несвязный, то его можно разбить на подграфы. Слабая связность – орграф заменяется неориентированным графом, который в свою очередь является связным. Односторонняя связность – это такая связность, когда между двумя вершинами существует путь в одну или в другую сторону. Сильная связность – это связность, когда между любыми двумя вершинами существует путь в одну и в другую сторону.

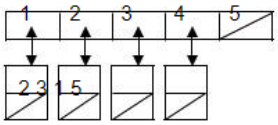
43. СД типа «граф». Представление графов в памяти компьютера. Матрица смежности. Алгоритм Уоршелла см 42



Для неорграфа:
1: 2, 3; 2: 1, 3; 3: 1, 2; 4: 5; 5: 4.

Для орграфа: 1: 2; 2: 3; 3: 1; 4: 5; 5: - .

Структуру смежности можно реализовать массивом из n линейно связанных списков:



Представление графа может оказать влияние на эффективность алгоритма. Часто запись алгоритмов на графах задается в терминах вершин и дуг, независимо от представления графа. Например, алгоритм определения количества последователей

Представление графов в памяти компьютера существенно зависит от типов структур данных, допускаемых используемым алгоритмическим языком и типом компьютера. Представление с помощью матрицы смежности - одно из самых распространенных. Для графов с большим числом дуг это достаточно компактное представление, особенно если есть возможность работать с двоичными битами в машинном слове. К недостаткам следует отнести большой расход памяти при работе с графами, имеющими небольшое число дуг (матрица смежности при этом получается весьма разреженной), а также невозможность уменьшения трудоемкости алгоритмов в том случае, когда она пропорциональна числу дуг, а не числу вершин.

Матрица смежности неориентированного графа симметрична относительно главной диагонали, поэтому достаточно хранить в памяти только половину ее.

Алгоритм 3.12 (рис.3.19) вычисления транзитивного замыкания отношения A основан на рассмотренном выше принципе (алгоритм Уоршалла).

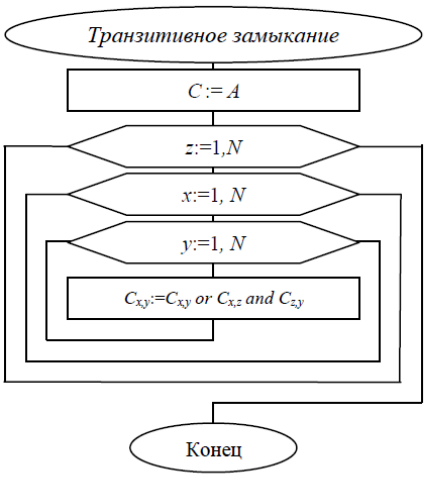


Рис.3.19. Блок-схема алгоритма вычисления транзитивного замыкания

Представление с помощью матрицы инциденций определяет граф однозначно (с точностью до изоморфизма), но применяется крайне редко в силу большой разреженности матрицы и практического отсутствия алгоритмов, работающих на такой структуре данных.

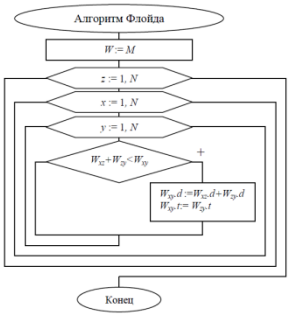
Представление с помощью списков смежности является основной альтернативой представлению с помощью матрицы смежности. Список смежности для вершины v - это список концов дуг, исходящих из вершины v в случае орграфа, или список смежных с v вершин в случае графа. Граф представляется с помощью !x! списков смежности, по одному для каждой вершины. Если число дуг в орграфе существенно мало по сравнению с полным графом, то этот сподоб представления весьма эффективен. Списки смежности занимают объем памяти !x!+!e! и легко реализуются с помощью списочных структур. Менее удобен этот способ представления для задания взвешенных графов, так как тогда возникает необходимость хранения весов дуг и установления соответствующих связей между дугами и их весами.

В связной памяти наиболее часто представление графа осуществляется с помощью структур смежности. Для каждой вершины множества X задается множество M(Xi) соответственно вех его последователей (если это орграф) или соседей (для неорграфа). Таким образом, структура смежности графа G будет представлять собой список таких множеств: < M(X1), M(X2),…, M(Xn)> для всех его вершин.

Рассмотрим пример (узлы обозначаем в виде цифр: 1, 2,…, n):

УОРШЕЛ РОДНОЙ ДИСКРЕТКА:

44. СД типа «граф». Алгоритм Флойда см 42 43



Алгоритм Флойда схож с алгоритмом Уоршелла нахождения транзитивного замыкания. В этом алгоритме последовательно рассматриваются вершины орграфа и, если существует путь в две дуги от некоторой вершины x до вершины y через рассматриваемую вершину z и его длина меньше длины пути между вершинами x и y через рассмотренные ранее вершины, то в качестве кратчайшего пути из x в y нужно принять путь через вершину z. В начальный момент, когда ни одна из вершин графа не рассмотрена, информацию о кратчайших путях и их длинах между каждой парой вершин орграфа представляет модифицированная матрица весов M (см. выше метод Шимбелла). Алгоритм Флойда представлен блок-схемой на рис.4.43. Алгоритм 4.14 нахождения кратчайших путей между каждой парой вершин во взвешенном орграфе. Вход: M — модифицированная матрица весов; N — количество вершин в орграфе. Выход: W — матрица кратчайших путей и их длин между каждой парой вершин орграфа.

$M_{ij}.d$ — длина дуги из вершины i в вершину j;

$$M_{ij}.t = \begin{cases} i, \text{ если есть дуга из вершины } i \text{ в вершину } j; \\ 0, \text{ если } i = j; \\ -1, \text{ если нет дуги из вершины } i \text{ в вершину } j. \end{cases}$$

45. Внешняя сортировка. Особенности внешней сортировки. Алгоритм слияния

Внешние сортировки применяются к данным, которые хранятся во внешней памяти. При выполнении таких сортировок требуется работать с данными, расположенными на внешних устройствах последовательного доступа. Для файлов, расположенных на таких устройствах в каждый момент времени доступен только один компонент последовательности данных, что является существенным ограничением по сравнению с сортировкой массивов, где всегда доступен каждый элемент.

Внешняя сортировка – это сортировка данных, которые расположены на внешних устройствах и не вмещающихся в оперативную память.

Данные, хранящиеся на внешних устройствах, имеют большой объем, что не позволяет их целиком переместить в оперативную память, отсортировать с использованием одного из алгоритмов внутренней сортировки, а затем вернуть их на внешнее устройство. В этом случае осуществлялось бы минимальное количество проходов через файл, то есть было бы однократное чтение и однократная запись данных. Однако на практике приходится осуществлять чтение, обработку и запись данных в файл по блокам, размер которых зависит от операционной системы и имеющегося объема оперативной памяти, что приводит к увеличению числа проходов через файл и заметному снижению скорости сортировки.

К наиболее известным алгоритмам внешних сортировок относятся:

- сортировки слиянием (простое слияние и естественное слияние);
- улучшенные сортировки (многофазная сортировка и каскадная сортировка).

Из представленных внешних сортировок наиболее важным является метод сортировки с помощью слияния. Прежде чем описывать алгоритм сортировки слиянием введем несколько определений.

Основным понятием при использовании внешней сортировки является понятие серии. Серия (упорядоченный отрезок) – это последовательность элементов, которая упорядочена по ключу.

Количество элементов в серии называется длиной серии. Серия, состоящая из одного элемента, упорядочена всегда. Последняя серия может иметь длину меньшую, чем остальные серии файлов. Максимальное количество серий в файле N (все элементы не упорядочены). Минимальное количество серий одна (все элементы упорядочены).

В основе большинства методов внешних сортировок лежит процедура слияния и процедура распределения. Слияние – это процесс объединения двух (или более) упорядоченных серий в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент. Распределение – это процесс разделения упорядоченных серий на два и несколько вспомогательных файла.

Фаза – это действия по однократной обработке всей последовательности элементов. Двухфазная сортировка – это сортировка, в которой отдельно реализуется две фазы: распределение и слияние. Однофазная сортировка – это сортировка, в которой объединены фазы распределения и слияния в одну.

Двухпутевым слиянием называется сортировка, в которой данные распределяются на два вспомогательных файла. Многопутевым слиянием называется сортировка, в которой данные распределяются на N (N > 2) вспомогательных файлов.

Общий алгоритм сортировки слиянием

Сначала серии распределяются на два или более вспомогательных файлов. Данное распределение идет поочередно: первая серия записывается в первый вспомогательный файл, вторая – во второй и так далее до последнего вспомогательного файла. Затем опять запись серии начинается в первый вспомогательный файл. После распределения всех серий, они объединяются в более длинные упорядоченные отрезки, то есть из каждого вспомогательного файла берется по одной серии, которые сливаются. Если в каком-то файле серия заканчивается, то переход к следующей серии не осуществляется. В зависимости от вида сортировки сформированная более длинная упорядоченная серия записывается либо в исходный файл, либо в один из вспомогательных файлов. После того как все серии из всех вспомогательных файлов объединены в новые серии, потом опять начинается их распределение. И так до тех пор, пока все данные не будут отсортированы.

Общий алгоритм 2:

Сначала серии распределяются на два или более вспомогательных файлов. Данное распределение идет поочередно: первая серия записывается в первый вспомогательный файл, вторая – во второй и так далее до последнего вспомогательного файла. Затем опять запись серии начинается в первый вспомогательный файл. После распределения всех серий, они объединяются в более длинные упорядоченные отрезки, то есть из каждого вспомогательного файла берется по одной серии, которые сливаются. Если в каком-то файле серия заканчивается, то переход к следующей серии не осуществляется. В зависимости от вида сортировки сформированная более длинная упорядоченная серия записывается либо в исходный файл, либо в один из вспомогательных файлов. После того как все серии из всех вспомогательных файлов объединены в новые серии, потом опять начинается их распределение. И так до тех пор, пока все данные не будут отсортированы.

Ключевые термины

Внешняя сортировка – это сортировка данных, которые расположены на внешних устройствах и не вмещающихся в оперативную память.

Двухпутевое слияние – это сортировка, в которой данные распределяются на два вспомогательных файла.

Двухфазная сортировка – это сортировка, в которой отдельно реализуется две фазы: распределение и слияние.

Длина серии – это количество элементов в серии.

Естественное слияние – это сортировка, при которой всегда сливаются две самые длинные из возможных серий.

Многопутевое слияние – это сортировка, в которой данные распределяются на N (N > 2) вспомогательных файлов.

Несбалансированное слияние – это естественное слияние, у которого после фазы распределения количество серий во вспомогательных файлах отличается друг от друга более чем на единицу.

Однофазная сортировка – это сортировка, в которой объединены фазы распределения и слияния в одну.

Простое слияние – это одна из сортировок на основе слияния, в которой длина серий фиксируется на каждом шаге.

Распределение – это процесс разделения упорядоченных серий на два и несколько вспомогательных файла.

Сбалансированное слияние – это естественное слияние, у которого после фазы распределения количество серий во вспомогательных файлах отличается друг от друга не более чем на единицу.

Серия (упорядоченный отрезок) – это последовательность элементов, которая упорядочена по ключу.

Слияние – это процесс объединения двух (или более) упорядоченных серий в одну упорядоченную последовательность при помощи циклического выбора элементов доступных в данный момент.

Фаза – это действия по однократной обработке всей последовательности элементов.

46. Внешняя сортировка. Алгоритм прямого слияния и его модификация. Анализ смежности см 45

Исходный файл f: 5 7 3 2 8 4 1

	Распределение	*Слияние*
1 проход	*f1*: 5 3 8 1 *f2*: 7 2 4	*f*: 5 7 2 3 4 8 1
2 проход	*f1*: 5 7 4 8 *f2*: 2 3 1	*f*: 2 3 5 7 1 4 8
3 проход	*f1*: 2 3 5 7 *f2*: 1 4 8	*f*: 1 2 3 4 5 7 8

Прямое слияние.
Алгоритм сортировки прямым слиянием является простейшим алгоритмом внешней сортировки, основанный на процедуре слияния серией. В данном алгоритме длина серий фиксируется на каждом шаге. В исходном файле все серии имеют длину 1, после первого шага она равна 2, после второго – 4, после третьего – 8, после k -го шага – 2k.
Алгоритм сортировки простым слиянием:
1. Исходный файл f разбивается на два вспомогательных файла f1 и f2.
2. Вспомогательные файлы f1 и f2 сливаются в файл f, при этом одиночные элементы образуют упорядоченные пары.
3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.
4. Повторяя шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл (Рис.1).
После выполнения i **проходов** получаем два файла, состоящих из серий длины 2i. Окончание процесса происходит при выполнении условия 2i>=n. Следовательно, процесс сортировки простым слиянием требует порядка O (log n) проходов по данным.
Признаками конца сортировки простым слиянием являются следующие условия:
1. длина серии не меньше количества элементов в файле (определяется после фазы слияния);
2. количество серий равно 1 (определяется на фазе слияния).
3. количество серий равно 1 (определяется на фазе слияния).

47. Внешняя сортировка. Алгоритм многофазной сортировки. Определение исходного распределения серий

При использовании рассмотренного выше метода сбалансированной многопутевой внешней сортировки на каждом шаге примерно половина вспомогательных файлов используется для ввода данных и примерно столько же для вывода сливаемых серий. Идея многофазной сортировки состоит в том, что из имеющихся m вспомогательных файлов (m-1) файл служит для ввода сливаемых последовательностей, а один - для вывода образуемых серий. Как только один из файлов ввода становится пустым, его начинают использовать для вывода серий, получаемых при слиянии серий нового набора (m-1) файлов. Таким образом, имеется первый шаг, при котором серии исходного файла распределяются по m-1 вспомогательному файлу, а затем выполняется многопутевое слияние серий из (m-1) файла, пока в одном из них не образуется одна серия.
Очевидно, что при произвольном начальном распределении серий по вспомогательным файлам алгоритм может не сойтись, поскольку в единственном непустом файле будет существовать более, чем одна серия. Предположим, например, что используется три файла B1, B2 и B3, и при начальном распределении в файл B1 помещены 10 серий, а в файл B2 - 6. При слиянии B1 и B2 к моменту, когда мы дойдем до конца B2, в B1 останутся 4 серии, а в B3 попадут 6 серий. Продолжится слияние B1 и B3, и при завершении просмотра B1 в B2 будут содержаться 4 серии, а в B3 останутся 2 серии. После слияния B2 и B3 в каждом из файлов B1 и B2 будет содержаться по 2 серии, которые будут слиты и образуют 2 серии в B3 при том, что B1 и B2 - пусты. Тем самым, алгоритм не сошелся (таблица 3).
На каждом шаге мы берем наименьший из начальных элементов входных серий и перемещаем в конец выходной серии. Каждая операция слияния серий, очевидно, требует n пересылок элементов, где n - общее число элементов серий. В процессе сортировки мы будем оперировать лентами – структурами данных, где в каждый момент нам доступен либо первый элемент, либо следующий элемент после уже прочитанного. В реальной ситуации в качестве лент выступают односвязные списки или файлы. Пусть у нас имеется N лент: N - 1 входная и одна пустая. Мы будем слиять элементы со входных лент на выходную, пока какая-либо из них не опустеет. Затем она станет входной. Пример сортировки с шестью лентами, содержащими всего 65 серий. Серии обозначены буквами fi. Цифры - количество серий, находящихся на ленте в начале очередного шага.

Стрелки показывают процесс слияния. Например, на втором шаге мы слияем с f1, f2, f3, f4 и f6 на f5, пока одна из лент не опустеет.

В каждый момент времени слияние происходит на пустую ленту с остальных, поэтому число требующихся проходов приблизительно равно log n.N
Далее нам понадобятся числа Фибоначчи порядка p. Они определяются следующим образом: f_(i+1) = f_i + f_(i-1) + ... + f_(i-p) для i >=p, f_p = 1, f_i = 0 для 0 <= i < p.
Очевидно, обычные числа Фибоначчи имеют порядок 1.
В данном примере распределение начальных серий побрано искусственно. Исходные числа серий для такой идеальной многофазной сортировки должны быть суммами n-1 , n-2 , ... , 1 последовательных чисел Фибоначчи порядка n-2.
Из этого следует, что наш алгоритм многофазного слияния применим только к таким входным данным, в которых число серий есть сумма n-1 таких сумм Фибоначчи.
Что делать, если число начальных серий не является такой идеальной суммой?
Ответ прост - предположим существование гипотетических пустых серий, таких что сумма реальных и гипотетических серий дает идеальную сумму. Пустые серии называются фиктивными или холостыми сериями. Холостые серии по возможности равномерно распределяются по лентам, чтобы реальное слияние (в котором холостые серии участвуют лишь 'в уме') проходило с как можно большего количества лент.

Сначала все данные располагаются на одной ленте. Лента читается и отрезки распределяются по другим лентам, имеющимся в системе. после того, как созданы начальные отрезки, они сливаются, как описано выше. Один из методов, который можно использовать для создания начальных отрезков, состоит в чтении порции записей в память, их сортировке и записи результата на ленту. Выбор с замещением(см. другой вопрос) позволяет нам получать более длинные отрезки. Этот алгоритм работает с буфером, располагающимся в оперативной памяти. Сначала мы заполняем буфер. Затем повторяем следующие шаги до тех пор, пока не будут исчерпаны входные данные:

* Выбрать запись с наименьшим ключом, т.е. с ключом, значение которого >= значения ключа последней прочитанной записи.
* Если все "старые" ключи меньше последнего ключа, то мы достигли конца отрезка. Выбираем запись с наименьшим ключом в качестве первого элемента следующего отрезка.
* Записываем выбранную запись.
* Заменяем выбранную и записанную запись на новую из входного файла

48. Внешняя сортировка. Алгоритм каскадной сортировки. Определение исходного распределения серий

49. B-деревья. Основные определения. Алгоритм поиска

50. B-деревья. Алгоритм включения в B-дерево. Пример

51. B-деревья. Алгоритм исключения из B-дерева. Пример

52. СД типа «граф». Представление графа с помощью списков смежности. Алгоритм прохождения графа в «глубину»

Сортировка простым слиянием

Одна из сортировок на основе слияния называется простым слиянием.

Алгоритм сортировки простым слияния является простейшим алгоритмом внешней сортировки, основанный на процедуре слияния серией.

В данном алгоритме длина серий фиксируется на каждом шаге. В исходном файле все серии имеют длину 1, после первого шага она равна 2, после второго – 4, после третьего – 8, после k -го шага – 2k.

Алгоритм сортировки простым слиянием

Шаг 1. Исходный файл f разбивается на два вспомогательных файла f1 и f2.

Шаг 2. Вспомогательные файлы f1 и f2 сливаются в файл f, при этом одиночные элементы образуют упорядоченные пары.

Шаг 3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.

Шаг 4. Повторяя шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл (рис. 43.1).

После выполнения i проходов получаем два файла, состоящих из серий длины 2i. Окончание процесса происходит при выполнении условия 2i>=n. Следовательно, процесс сортировки простым слиянием требует порядка O(log n) проходов по данным.

Признаками конца сортировки простым слиянием являются следующие условия:

длина серии не меньше количества элементов в файле (определяется после фазы слияния);

количество серий равно 1 (определяется на фазе слияния).

при однофазной сортировке второй по счету вспомогательный файл после распределения серий остался пустым.