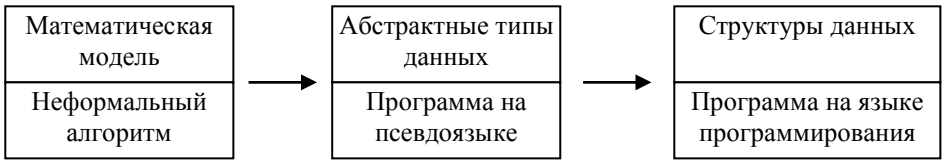
1. Введение

1.1. От задачи к программе (применения структур данных в программировании)

Тема рассматривается на примере решения задачи составления режимов работы светофора на перекрестке Принстонского университета /1/, стр. 16-22



Математическая модель – формальное описание задачи с указанием математических принципов (теоремы, ряды, и т.д.), на которых может строиться её решение (взаимозависимостей внутри задачи).

Алгоритм – конечная последовательность инструкций, направленная на решение задачи. Каждая из инструкций имеет четкий смысл и может быть выполнена с конечными вычислительными результатами за конечное время.

Тип данных – множество значений, которое может принимать переменная.

Пример: переменная логического типа может принимать значения true (истина) и false (ложь).

Структуры данных – набор переменных, возможно различных типов данных, объединенных определенным образом. Базовый строительный блок структуры данных – ячейка. Ячейка предназначена для хранения значения определенного базового или составного типа данных.

Два основных способа создания структур данных:

- агрегирование - создание совокупности ячеек;
- указатели (или курсоры) - интерпретации значения некоторых ячеек как представителей других ячеек.

Среди механизмов агрегирования обычно выделяют:

- массивы (одномерные и многомерные).
- записи (в некоторых языках их называют структурами).
- файлы (многие языки их напрямую не поддерживают).

Кроме того, некоторые языки вводят дополнительные способы агрегирования, например объединения.

Указатель (pointer) – ячейка, чье значение является адресом другой ячейки.

Курсор (cursor) – ячейка со значением-индексом, используемым для указания на элемент массива. Курсоры применяются чаще всего в языках, не поддерживающих прямо указатели.

Абстрактный тип данных – математическая модель с совокупностью операторов, определенных в пределах данной модели.

АТД - некоторая сущность, над которой возможно выполнение некоторых операций. АТД - черный ящик, с определенными, но не описанными операциями. АТД соответствует описанное математическое понятие: граф, стек, очередь, и т.д.

Ничего о реализации или внутреннем представлении АТД не говорится.

Пример: Множество. В дискретной математике множество определяется как набор элементов, который представляется как нечто целое. Над множеством определяются операции объединения, пересечения, дополнения. Кроме того, существуют понятия мощности (размера) множества, принадлежности элемента множеству и эквивалентности множеств.

Исходя из этого, можно определить АТД Множество (Set), как набор операторов:

- Union (A, B) – объединение двух множеств. Результат – новое множество.
- Intersection (A, B) – пересечение множеств. Результат – множество.
- Complement (A, B) – дополнение множеств. Результат – множество.
- Size (A) – размер. Результат – целое число.

И т.д.

Реализацией АТД называют представление на языке программирования операторов над АТД и объявление необходимых для их поддержки переменных.

Например, возможной реализацией множества может быть массив, содержащий элементы множества.

1.2. Замечание о времени выполнения алгоритма (программы).

Говоря о характеристиках алгоритма, чаще всего имеют в виду размер используемой памяти и время выполнения. Задача оптимизации объема используемой памяти встречается реже, нежели задача оптимизации времени.

Время выполнения принято выражать как функцию от числа элементов, над которыми производится обработка. Время работы алгоритмов на ЭВМ разной мощности различно, поэтому принято измерять время выполнения в количестве элементарных (неделимых шагов). Анализируя алгоритм, можно попытаться найти точное количество выполняемых шагов, но в большинстве случаев в этом нет никакого смысла, или вовсе невозможно.

Зачастую достаточно оценить асимптотику (т.е. поведение функции) времени работы алгоритма при стремлении размера входа к бесконечности.

При этом можно указать вид функции, к которой стремится, или которой ограничивается время работы алгоритма.

Пример: Алгоритм последовательного поиска в массиве в среднем равен $\frac{1}{2}n$, а двоичный поиск на упорядоченном массиве дает время работы не превосходящее $c \cdot \log_2 n$ (где c – некоторая константа).

Для записи асимптотических соотношений используют тета- (Θ-), о- (O-), омега- (Ω-) обозначения.

Если g(n) – некоторая функция, то

1) $f(n) = \Theta(g(n))$, означает, что найдется такие $c_1, c_2 > 0$ и такое n_0, что $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ для всех $n \ge n_0$.

2) $f(n) = \mathrm{O}(g(n))$, означает, что найдется такая $c > 0$ и такое n_0, что $0 \le f(n) \le c \cdot g(n)$ для всех $n \ge n_0$

3) $f(n) = \Omega(g(n))$, означает, что найдется такая $c > 0$ и такое n_0, что $0 \le c \cdot g(n) \le f(n)$ для всех $n \ge n_0$

Т.е. O и Ω оценки являются верхней и нижней границей. А Θ - двусторонней.

Теорема. Для любых двух функций $f(n)$ и $g(n)$ свойство $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = \mathrm{O}(g(n))$ и $f(n) = \Omega(g(n))$. Т.е. если употребляется обозначение $f(n) = \Theta(g(n))$, то верхняя и нижняя границы функций совпадают.

Пример: для алгоритма последовательного поиска время работы $T(n) = \Theta(n)$, а для двоичного поиска $T(n) = \mathrm{O}(\log_2 n)$

2. Списки.

2.1. Понятие

Список – последовательность элементов определенного типа.

Важнейшее свойство списка – каждый элемент списка имеет свою позицию. Т.е. элементы в списке *линейно упорядочены*.

Так в списке $a_1, a_2, \ldots, a_n$ элемент a_i предшествует a_{i+1}. Кроме того, элемент a_1 называется первым, а a_n – последним элементом и каждый элемент списка имеет свою позицию i.

Существует два подхода к заданию набора операторов над списками. Они различаются для списков с текущей позицией и без нее.

В первом случае все операции над списками (вставка, удаление, получение, …) выполняются только для *текущей позиции,* кроме того, для таких списков выделяют операции навигации по списку (перейти на начало, на конец, на одну позицию вперед или назад, …).

Во втором случае, позиция, над которой будут осуществляться действия указывается при вызове оператора.

Операторы для списка без текущей позиции:

1. Insert (x, p) – вставить элемент x в позицию p. Над списком будет произведена следующая манипуляция: $a_1, a_2, \ldots, a_n \Rightarrow a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n$.
2. Locate (x) – возвращает позицию объекта x. Если элементов несколько, то только первый от начала.
3. Retrieve (p) – возвращает элемент стоящий в позиции p.
4. Delete (p) – удаляет элемент в позиции p. $a_1, a_2, \ldots, a_n \Rightarrow a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n$.
5. Makenull – очищает список.

Операторы для списка с текущей позицией:

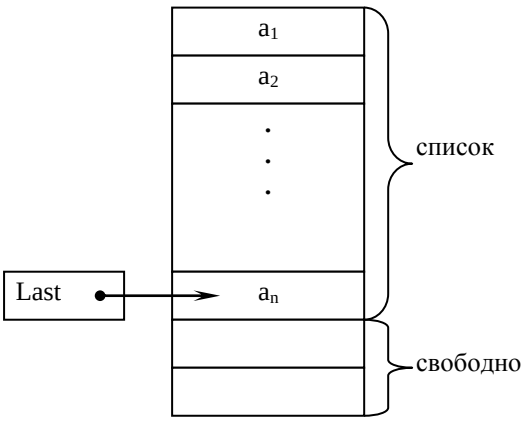
1. Add (x) – добавить x в текущую позицию.
2. Get – получить текущий элемент.
3. Put (x) – изменить текущий элемент на x.
4. Del – удалить текущий элемент.
5. GoFirst, GoLast, GoNext, GoPrevious – перейти на начало, в конец, вперед, назад.

2.2. Реализация

2.2.1. … посредством массивов

При реализации списков на массиве, элементы располагаются в смежных ячейках массива.

Очень легко осуществляется просмотр и вставка в конец списка, зато при удалении или вставке приходится сдвигать все элементы, находящиеся за вставляемым (удаляемым) элементом.



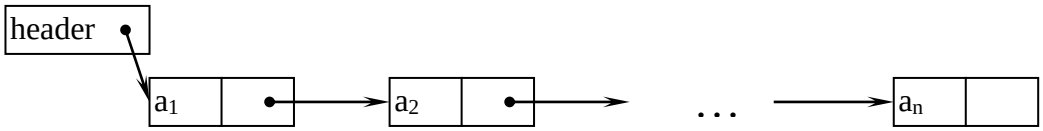
```
const
  Maxlength = 100
type
  TList  = object
     Elements : array [1..maxlength] of TType
      Last:integer;
      Current:integer; {только для списков с текущим элементом}

      {Методы}
  end;
```

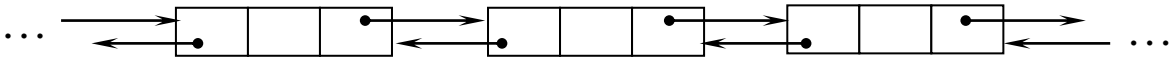
2.2.2. … с помощью указателей

В этом случае каждый элемент списка кроме собственно значения имеет еще указатель на следующий элемент.

Если список состоит из элементов a_1, a_2, …, a_n, то для i=1, 2, …, n-1 ячейка, содержащая a_i, будет иметь указатель на ячейку содержащую a_{i+1}. Ячейка с a_n имеет пустой указатель (указатель, имеющий специальное зарезервированное значение **nil**). Кроме того, имеется ячейка, указывающая на самый первый элемент списка.



Помимо приведенной схемы существуют также дважды связные (двусвязанные) списки. В них каждый элемент кроме указателя на следующий, имеет также указатель на предыдущий элемент.



Реализация:

```
type
  TCellType = record
       Data : TType;
       Next : ^TCellType;
    end;

Tlist = obyect
   Header : ^TCellType;
{Дополнительно}
   Last : ^TCellType;
   Current : ^TCellType;
{Методы}
end;
```

Для двусвязных списков все почти аналогично:

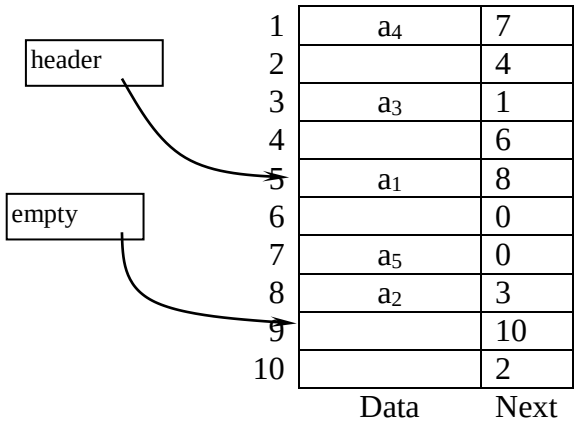
```
type
  TCellType = record
       Data : TType;
       Next, Previous : ^TCellType;
    end;

Tlist = obyect
   Header : ^TCellType;
{Дополнительно}
   Last : ^TCellType;
   Current : ^TCellType;
{Методы}
end;
```

2.2.3. … на основе курсоров

Данная реализация очень напоминает реализацию на основе указателей, за исключением того, что курсоры являются индексами в массиве, и необходимо самим следить за тем, какие ячейки массива пусты, а какие заняты.

Для этого предлагается использовать цепочку пустых ячеек. Т.е. в массиве будет одновременно два списка – пустых ячеек и собственно сам список.



Аналогично можно поступить и с двунаправленными списками.

2.2.4. Сравнение реализаций

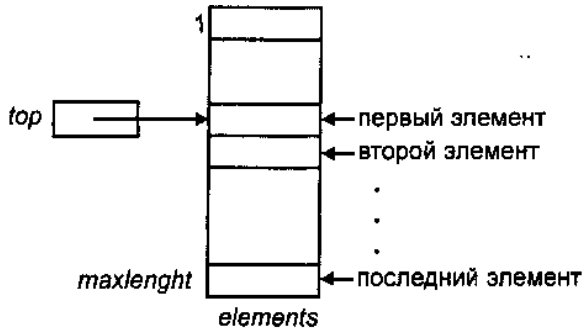
	На массиве	Однонаправленный с указателями	Двунаправленный с указателями
Insert, Delete	O(n)	O(n), но c=1/2	O(n)
Locate / Retrive	O(n) / O(1)	O(n)	O(n)
Add, Delete	O(n)	O(1)	O(1)
Put, Get	O(1)	O(1)	O(1)
GoNext / GoPrevious	O(1)	O(1) / O(n)	O(1)

3. Стеки

Стек (stack) – это специальный вид списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (top). Стеки также иногда называют магазинами. Стратегия работы со стеком носит название LIFO (last-in-first-out – последний вошел – первый вышел).

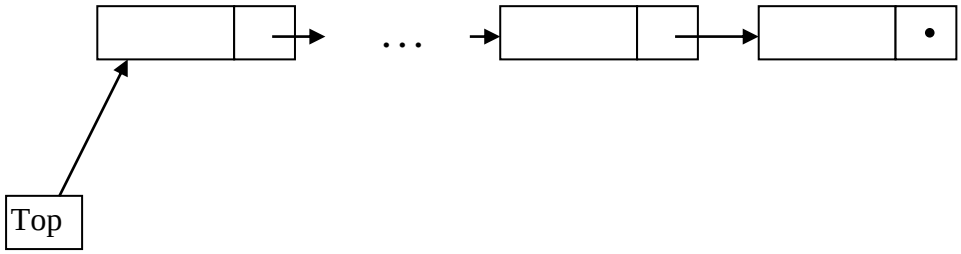
Основные операции над стеком.

1) Push (x) – поместить элемент x на вершину стека.
2) Pop – вытолкнуть элемент из вершины стека.
3) Get – получить вершину стека без выталкивания.
4) Empty – возвращает True, если стек пуст.



Стек по реализации очень похож на однонаправленный список, с той лишь разницей, что все операции выполняются только над вершиной, поэтому их время работы O(1).

Очень часто при реализации на массиве стек “растет” в обратную сторону.



Благодаря специализации стека эффективней считаются реализации на массиве.

4. Очереди

Очередь (queue)– специальный вид списка, где элементы вставляются с одного конца (заднего – rear или хвоста), а удаляются с другого (переднего - front). Очереди также называют "списками типа FIFO" (first-in-first-out: первым вошел – первым вышел)

Основные операции:

1) Enqueue (x) – вставить x в конец очереди
2) Dequeue – извлечь элемент из начала очереди
3) Front – прочесть первый элемент очереди не извлекая его

4.1. Реализация очередей с помощью указателей

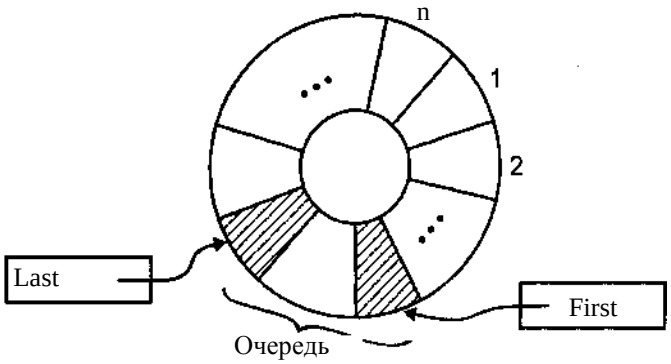
```
type
   celltype = record
      Data: TType;
      Next: ^celltype;
   end;
```

```
type
    TQueue = record
        First, Last:^celltype;
   end;
```

4.2. Реализация очередей с помощью циклических массивов

При обычной реализации очереди как списка на массиве можно добиться, чтобы время вставки было O(1), но операция извлечения из начала будет работать за O(n). Существует более эффективная реализация.

Представим массив в виде циклической структуры, где первая ячейка массива следует за последней. Элементы очереди располагаются в "круге" ячеек в последовательных позициях, конец очереди находится по часовой стрелке на определенном расстоянии от начала.



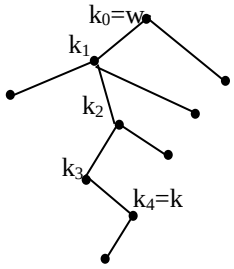
Для вставки элемента достаточно переместить указатель Last по часовой стрелке и записать новый элемент, для удаления – переместить First также по часовой стрелке. Единственная сложность с определением момента, когда очередь пуста – решается, как правило, введением флага или специального состояния для указателей в никуда.

5. Деревья.

5.1. Основные понятия.

Дерево — это совокупность элементов, называемых *узлами,* и отношений между ними, образующих иерархическую структуру узлов.

Свойства:



1) имеется ровно один узел w, называемый корнем, который не имеет предшественников;

2) каждый узел, кроме корня, имеет ровно одного непосредственного предка;

3) для каждого узла k, кроме корня, имеется последовательность узлов $w = k_0, k_1, \dots, k_n = k$, $n \geq 1$, в которой k_i – непосредственный потомок k_{i-1}, $1 \leq i \leq n$.

При рекурсивно определении деревом называют множество вершин, которое:

- либо пусто;
- либо может быть разбито на корень и некоторое число l (эль) непересекающихся подмножеств (поддеревьев), которые также являются деревьями.

Узлы, не имеющие потомков, называются листьями. Степень узла k – число его непосредственных потомков (сыновей). Наибольшая степень узла среди всех узлов дерева называется степенью дерева.

Сыновья могут иметь заданный порядок (правый-левый и т.п.) и не иметь его. В первом случае дерево называется упорядоченным, а во втором – неупорядоченным.

Путем из узла n_i в узел n_k называется последовательность узлов $n_1, n_2, \dots, n_k$, где для всех i, $1 < i < k$, узел n_i является родителем узла n_{i+1}. Длиной пути называется число, на единицу меньшее числа узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе.

Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа (прохождение только прямое). ***Высота дерева*** совпадает с высотой корня.

Глубина узла – длина пути от корня до узла.

Порядок обхода дерева.

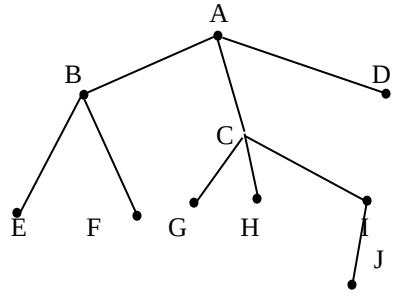
Выделяют

- прямой;
- обратный;
- внутренний (симметричный).

порядки обхода. Определяются они следующим образом:

Пусть *T* — дерево с корнем *n* и поддеревьями $T_1, T_2, \dots, T_k$

1. *Прямой порядок* сначала посещается корень n, затем поддеревья поддерева T_1 далее все поддеревья поддерева T_2, и т.д. Последними посещаются поддеревья поддерева T_k.
2. *Симметричный обход* сначала посещаются в симметричном порядке все поддеревья поддерева T_1 далее корень n, затем последовательно в симметричном порядке все поддеревья поддеревьев $T_2, \dots, T_k$.
3. *Обратный порядок* сначала посещаются в обратном порядке все поддеревья поддерева T_1, затем последовательно посещаются все поддеревья поддеревьев $T_2, \dots, T_{kt}$ также в обратном порядке, последним посещается корень *n.*



Например для данного рисунка:

- прямой A, B, E, F, C, G, H, I, J, D;
- обратный E, F, B, G, H, J, I, C, D, A;
- симметричный E, B, F, A, G, C, H, J, I, D.

Обычно отдельно выделяют так называемые двоичные деревья.

Т. е. деревья, каждый узел которого имеет не более двух потомков, которые называются соответственно левым и правым сыновьями.

5.2. Абстрактный тип данных «Дерево» (Tree)

Операции:
1) Parent (n) – возвращает предка узла n;
2) LeftmostChild (n) – возвращает самого левого сына узла n.
3) RightSibling (n) – возвращает правого брата узла n.
4) DelTree (n) – удаляет поддерево с корнем n
5) AddChild (n) – добавляет сына узлу n. Для упорядоченного дерева этот оператор может иметь вид AddChild (n, i) – где i номер нового узла в списке сыновей узла n.

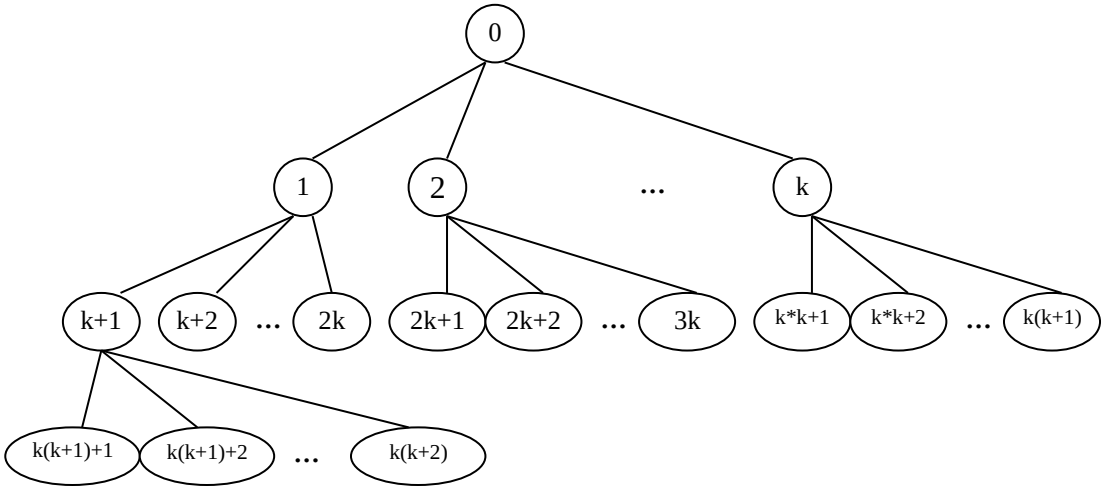
Для двоичного дерева набор операторов может быть следующим:
1) Parent (n)
2) Left(n), Right(n) – возвращает самого левого (правого) сына узла n.
3) Del (n) – удаляет поддерево с корнем n
4) AddLeft(n), AddRight(n) – добавляет правого (левого) сына узлу n.

Плюс операции для записи и чтения данных:
1) Put (n, x) – записать значение x в узел n.
2) Get (n) – получить значение из узла n.

5.3. Реализация деревьев

5.3.1. … на массиве

Пусть k - степень дерева.



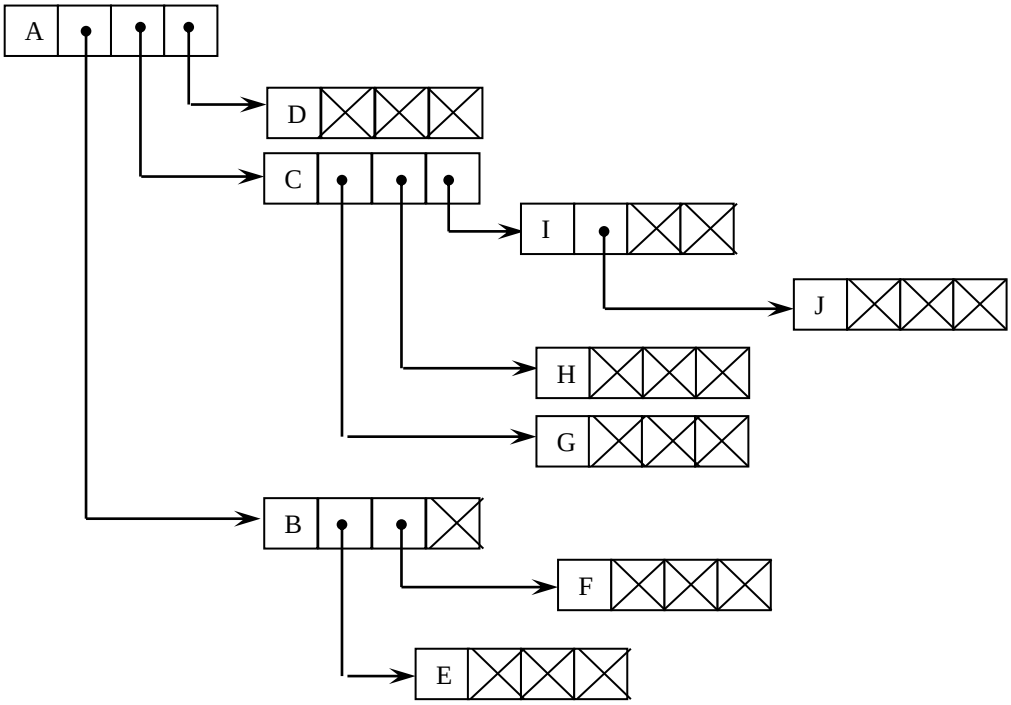
Тогда i-ый сын l-го узла для дерева степени n вычисляется по формуле

l*n+i;

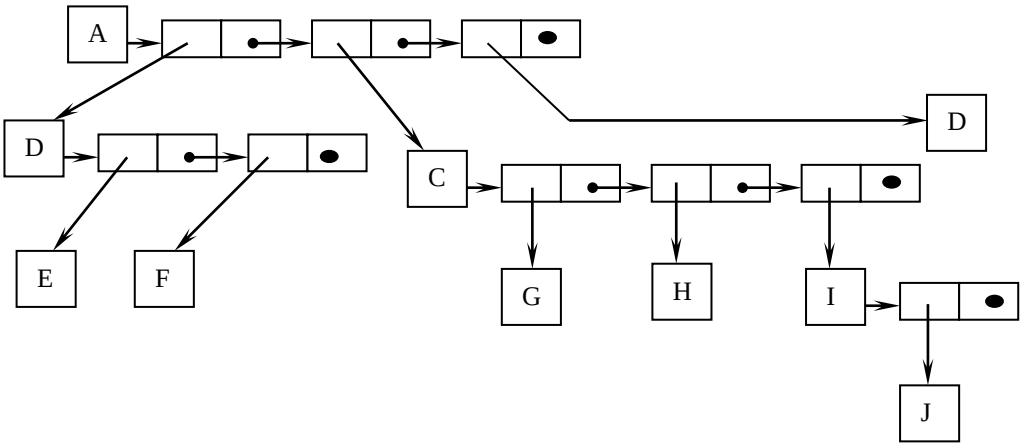
Теперь все элементы дерева можно расположить в последовательных ячейках массива и индексы сыновей вычислять по указанной формуле. Необходимо только завести некоторый признак, что данная ячейка пуста.

Данная реализация наиболее проста, но требует больших объемов памяти. Целесообразно применять только для сбалансированных деревьев с достаточно полным заполнением.

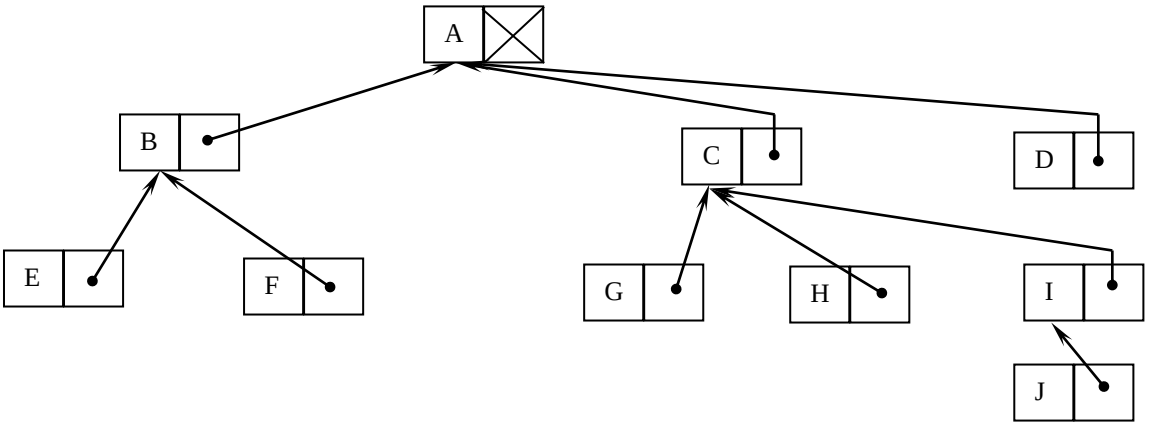
5.3.2. … с использованием фиксированного списка сыновей



5.3.3. … с использованием динамического списка сыновей



5.3.4. … с использованием указателя на предка



Данная схема хранения часто применяется в реляционных СУБД для хранения дерева. Требуется хранить список листьев или обладать способом их нахождения. В других требуется хранить только вершины

6. Множества

6.1. АТД «Множество»

Множество – базовое понятие (как точка) и не определяется. Однако можно дать следующее описание: «множество – собрание определенных и различных объектов, мыслимых как единого целого» (Г.Кантор).

Операции:

1. Union (A, B), Intersection (A, B), Complement/Difference (A, B) – объединение, пересечение и дополнение/разность двух множеств.
2. Member (x) – возвращает true, если $x \in A$.
3. Size – размер (мощность) множества.
4. Insert (x), Delete (x) – вставить/удалить x из множества.
5. Min, Max – возвращает минимальный/максимальный (имеет смысл только для линейно упорядоченных множеств).
6. Equal (A, B) – возвращает true, если множества равны.

6.2. Реализация множеств

6.2.1. … с помощью списков

При данной организации элементы списка являются элементами множества. При вставке элемента в множество, элемент просто добавляется в список, при выполнении оператора пересечения, создается новый список, в который добавляются элементы входящие как в первое, так и во второе множество.

Очевидно, существует два подхода: сортировать и не сортировать списки.

При использовании несортированного списка операция вставки элемента будет выполняться за конечное время, удаления за O(n), а пересечение, объединение и разность за O(nm).

Если же список сортированный, то все операции можно выполнить за O(n) или O(n+m).

6.2.2. … с помощью двоичных векторов.

Это самая эффективная реализация множества. Единственное ограничение – она может применяться лишь для счетных множеств.

В этой реализации множество представляется двоичным вектором, в котором i-ый бит равен 1 (или true), если элемент i является элементом множества и 0 – если не является.

0	1	2	3	4	5	6	7		N-3	N-2	N-1	N
1	0	0	0	1	0	0	1		1	1	0	0

Все операторы выполняются при помощи логических операций:

1. Объединение – A or B (логическое или).
2. Пересечение – A and B (логическое и).
3. Дополнение до универсума - not A
4. Разность множеств - A and (not B)

Пример. Рассмотрим представление и операции над множествами {2, 4, 7, 3, 15, 8, 11} и {2, 4, 3, 9, 8, 11, 1, 6, 10, 5}.

Для их представления достаточно 2 байт

{2, 4, 7, 3, 15, 8, 11} = 0011100110010001;

{2, 4, 3, 9, 8, 11, 1, 6, 10, 5} = 0111111011110000;

Результаты операций

1. Объединение {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15} = 0111111111110001;
2. Пересечение {2, 4, 3, 8, 11} = 0011100010010000;
3. Разности {7, 15} = 0000000100000001;

Т.к. большинство современных моделей компьютеров не могут адресовать биты. То адрес конкретного бита будет состоять из адреса ячейки и номера бита в ячейке.

Например, функция Member можно представить так (считаем, что вектор для множества состоит из ячеек размером в байт (8 бит), т.е. для представления номера достаточно иметь 3 бита).

```
function Member(x: integer) : Boolean;
begin
  Member := (BVector[x shr 3] And (1 shl (x And 7))) <> 0;
end.
```

6.2.3. Ещё одна реализация множества

Существуют и другие подходы к реализации множеств, например интервалами. Задаются интервалы элементов, входящих в множество, например: 1…3, 5…10.

Замечание! Если для реализации множества не нужны операции объединения, пересечения и разности, а нужны только вставка, удаление и проверка вхождения (поиска) элемента, то такие множества принято называть словарями.

6.3. Хеш-таблицы (hash table).

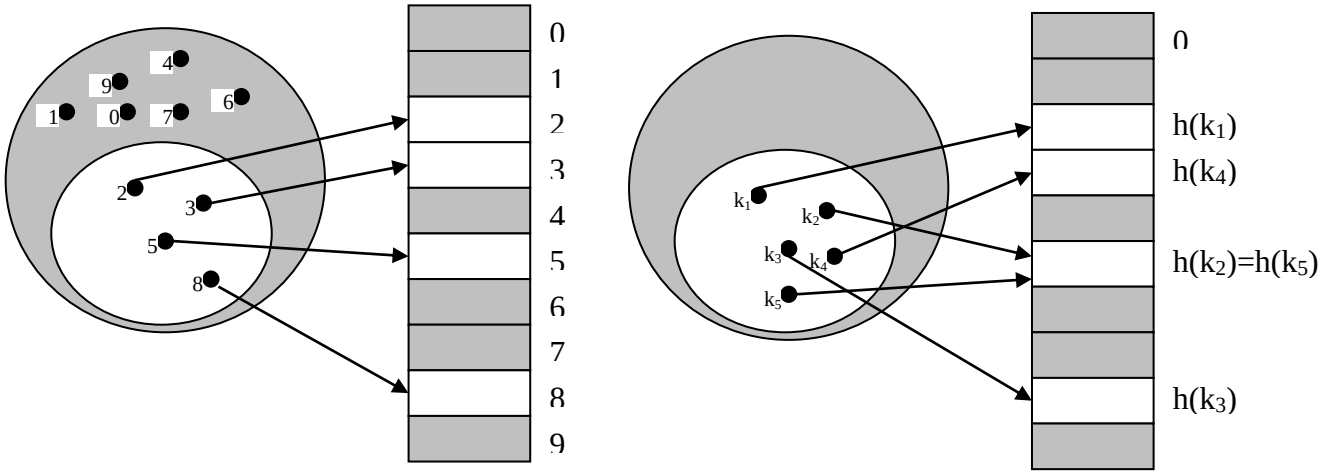
Понятие хеширования и хеш-таблицы.

Хеш-таблицу можно рассматривать как обобщенный вариант реализации массива с ключами. Если число элементов массива равно всех возможных ключей, то для каждого возможного ключа можно отвести ячейку и иметь время поиска O(1). Такой подход называют **прямой адресацией**. Т.к. в общем случае элементу с ключом k отводится позиция номер k.

Однако на практике это почти всегда невозможно, т.е. количество записей в таблице существенно меньше, чем количество всевозможных ключей. Поэтому при хешировании элементу с ключом k ставится в соответствие позиция с номер h(k) в хеш-таблице T[0..m-1], где

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

некоторая функция, называемая хеш-функцией (hash function). Число h(k) называют хеш-значением (hash value) ключа k.



Проблема состоит в том, что хэш-значения двух разных ключей могут совпадать, т.е. случаться коллизии (collision) или столкновения. Наилучшим является вариант, когда хеш-функция подобрана так, что коллизии невозможны. Но при |U|>m неизбежно

существуют разные ключи с одинаковым хеш-значением. Поэтому возникает два вопроса: выбор хеш-функции и выбор метода разрешения коллизий.

Выбор хеш-функции.

Считается, что хорошая хеш-функция должна удовлетворять предположениям равномерного хеширования: для очередного ключа все m хеш-значений должны быть равновероятны. Т.е. если ключи выбираются независимо друг от друга и каждый распределен в соответствии с P, то

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \quad для\ j = 0,1,...,m-1$$

Т.е. ключи *k* группируются в *m* групп, дающие одинаковые хэш-значения *j*, и вероятность появления ключа из каждой группе должна быть *1/m*.

Например, ключи действительные числа равномерно распределенные на [0;1), то хеш-функция $h(k) = \lfloor k \cdot m \rfloor$ (операция взятия целого, не превышающего аргумент), удовлетворяет этому условию.

Но чаще всего P неизвестно, а ключи не независимы, поэтому приходится использовать эвристики, основанные на специфике задачи.

Из «универсальных» функций, довольно популярными являются следующие методы:

a) деление с остатком

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

m рекомендуется выбирать простым и отстоящим от степени двойки.

b) умножение.

Выберем 0<A<1.

$$h(k) = \lfloor m[kA] \rfloor,$$

где [kA] – дробная часть kA.

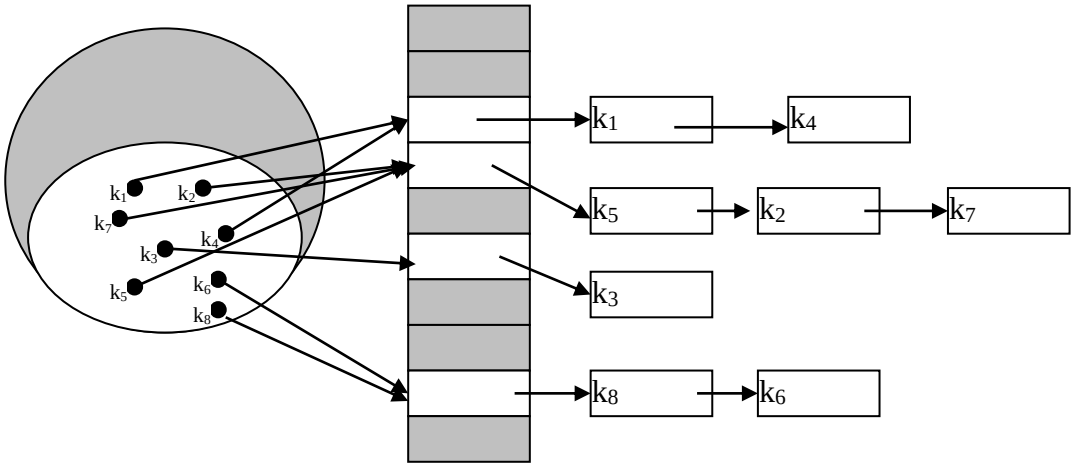
Метод умножения работает при любом A, но константа $A \approx (\sqrt{5}-1)/2 = 0{,}61803...$ является, по мнению Д. Кнута, довольно удачной.

Методы разрешения коллизий:

Для обработки (“разрешения ”) коллизий используют следующие способы:

1) Открытое хеширование (Ахо, Хопкрофт, Ульман) или хеширование с цепочками (Кормен, Лейзерсон, Ривест).
2) Закрытое хеширование (Ахо, Хопкрофт, Ульман) или метод открытой адресации (Кормен, Лейзерсон, Ривест).

При открытом хешировании ячейка хеш-таблицы указывает на список элементов с одинаковым хеш-значением.



Пусть m – число элементов хеш-таблицы, n – число занесенных элементов.

При равномерном хешировании (т.е. когда каждый элемент может попасть в любую из m позиций таблицы с равной вероятностью),время поиска как отсутствующего, так и присутствующего элемента в таблице будет пропорционально $\Theta(1+\alpha)$, где $\alpha=n/m$.

При закрытом хешировании применяется методика повторного хеширования (рехеширования). Если мы пытаемся поместить элемент x в сегмент с номером h(x), которая уже занята, то проверятся последовательность ячеек $h_1(k)$, $h_2(k)$, … до тех пор, пока не будет найдена свободная ячейка, или не выяснится, что таблица заполнена. Т.е. функция рехеширования зависит от номера попытки

Методы рехеширования (i – номер попытки):

1) Линейная последовательность проб.

Пусть h` - обычная хеш-функция, тогда линейная последовательность проб задается как

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

Иными словами, перебираются подряд ячейки, начиная с h`(k).

Недостаток метода: приводит к образованию кластеров – длинных последовательностей занятых ячеек, что сильно удлиняет поиск.

2) Квадратичная последовательность проб. Задается формулой

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

где c_1 и $c_2 \neq 0$ – некоторые константы.

Недостатки: ведет к образованию вторичных кластеров (что лучше чем первичные кластеры).

3) Двойное хеширование

Считается одним из лучших методов закрытого хеширования.

Задается формулой

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

где h_1 и h_2 – обычные хеш-функции. Рекомендуется $h_2(k)$ выбирать взаимно простым с m. Это можно сделать например так:

a) m – степень двойки, h_2 – принимает только нечетные значения.

b) m – простое, h_2 – целые положительные числа меньше m.

Например $h_1(k) = k \bmod m, \quad h_2(k) = 1 + (k \bmod m')$, где m` - чуть меньше m (m-1, m-2).

Для закрытого хеширования число проб в среднем при неуспешном поиске не превосходит,

$$1/(1-\alpha).$$

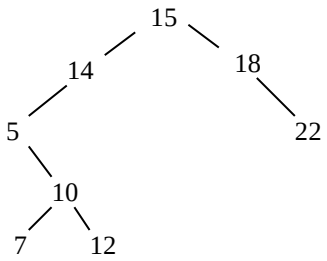
При успешном – не превосходит:

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

7. Специальные виды деревьев

7.1. Деревья двоичного поиска

Двоичное дерево поиска – двоичное дерево, узлы которого обладают следующим свойством: для любого узла i со значением x элементы, хранящиеся в правом поддереве, больше x, а элементы, хранящиеся в левом поддереве, – меньше.



Поиск по такому дереву выполняется аналогично двоичному поиску на массиве.

Можно показать, что в среднем операция поиска для такого дерева выполняется за $O(\log_2 n)$.

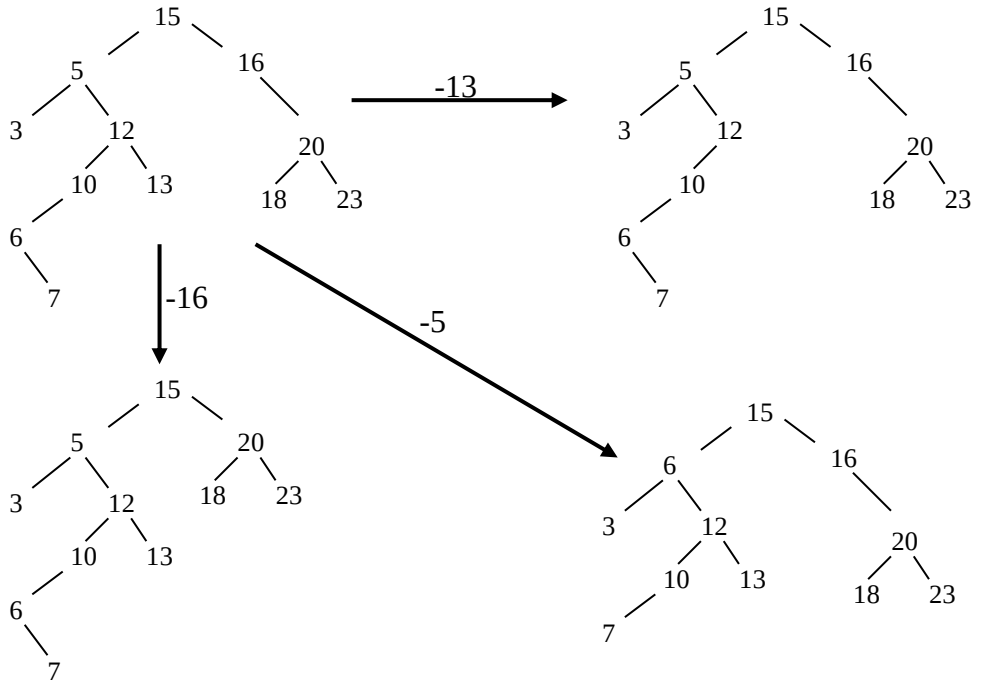
Наиболее употребительные операции для деревьев поиска как для словарей (Insert, Delete, Member, Min, Max).

Вставка всегда производится в листья.

С удалением сложнее. При удалении возможны три ситуации:

- удаляемая вершина является листом;
- удаляемая вершина имеет одного ребенка
- удаляемая вершина имеет двух детей.

Пример. Удаляется 13, 16, 5



При удалении элемента с двумя потомками, его место должен занять либо максимальный элемент из левого (минимального) поддерева, или минимальный из правого (максимального) поддерева.

7.2. Сбалансированные деревья

В общем случае оценка поиска (вставки и удаления) в двоичное дерево поиска не равняется $O(\log_2 n)$, а должна быть $O(h)$, где h – высота дерева. Чтобы сделать это соотношение близким или равным $O(\log_2 n)$, применяют балансировку дерева.

Существует множество подходов к определению сбалансированного дерева:

1. Дерево называется сбалансированным, если глубина любых двух его листьев отличается не более чем в 2 раза (применяется для красно-черных деревьев).
2. α-сбалансированные деревья.

Пусть x-вершина двоичного дерева, size(x) – число ключей в поддереве с вершиной x. ½ ≤ α <1. Тогда вершина x (не являющаяся листом) α-сбалансирована, если

$$Size(left(x)) \leq \alpha * size(x)$$

И

$$Size(right(x)) \leq \alpha * size(x)$$

Дерево называется α-сбалансированным, если все его вершины (кроме листьев) α-сбалансированы.

Известные схемы сбалансированных деревьев

1) АВЛ – деревья (Г.М. Адельсон-Вельский и Е.М. Ландис)
2) 2-3 – деревья
3) Красно-черные деревья
4) Б-деревья

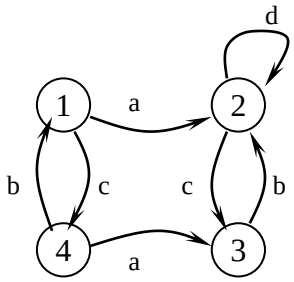
8. Графы

Граф – множество вершин и дуг, соединяющих эти вершины. Дуги могут быть и направленными, и нет. Ориентированные дуги называются ребрами.

Представление ориентированных графов.

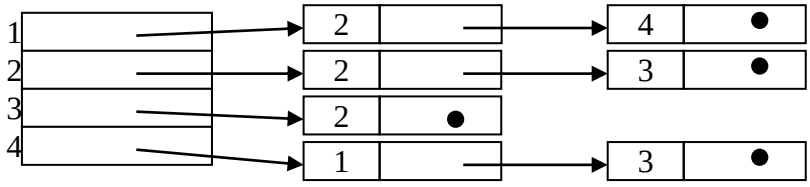
1. Матрица смежности

	1	2	3	4
1		a		c
2		d	c	
3		b		
4	b		a	



Основной недостаток – большой расход памяти и время обхода матрицы (при числе дуг и вершин пропорциональном n обход графа будет выполняться за n^2).

2. Списки смежности.



Представление неориентированных графов.

Методы аналогичны методам для ориентированных графов. За исключением того, что матрица смежности будет симметричной и можно хранить только верхнюю половину таблицы.

Основные задачи на ориентированных графах:

1. Нахождение кратчайшего пути.
2. Обход графа в глубину
   а) Проверка ацикличности графа
   б) Нахождение компонент сильной связности

Основные задачи на неориентированных графах:

1. Обход графа в глубину
2. Обход графа в ширину
3. Построение остовного дерева минимальной стоимости.

Задачи на графах:

1) Нахождение кратчайшего пути

При нахождении кратчайшего пути обычно используют один из двух классических алгоритмов:

- алгоритм Дейкстры;
- алгоритм Флойда;

Алгоритм Дейкстры.

V – множество всех вершин;

S – множество обработанных вершин;

D – массив текущих минимальных расстояний (D[i] – расстояние от первой до i-ой вершины)

procedure Dijkstra;

```
S := {1};
for i:= 2 to n do
  D[i] := C[1, i];
for i:= 1 to n-1 do
  begin
  выбрать вершину w из множества V-S, такую, что D[w] минимально;
  S := S + {w};
  for v∈(V-S) do
       D[v] := min(D[v], D[w]+C[w, v]);
  end;
end;
```

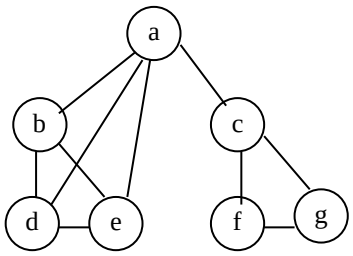
Время работы алгоритма $O(n^2)$.

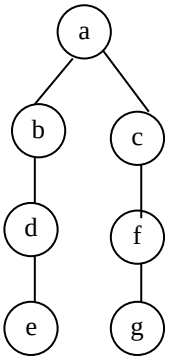
Часто для решения определенного круга задач необходимо систематически обходить вершины и дуги.

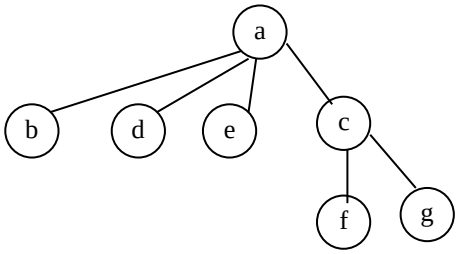
2) Обход графа в глубину и ширину

При обходе графа в глубину мы пытаемся «опуститься» максимально глубоко, пока это возможно, двигаясь по еще не посещенным вершинам. Пройдя таким образом до конца, мы возвращаемся и пытаемся исследовать другие ответвления.

При обходе графа в ширину сначала рассматриваются все вершины смежные с данной, после чего, для каждой вновь рассмотренной вершины операция повторяется.







В ходе выполнения мы построили остовное дерево.

Остовное дерево – связанный подграф исходного графа, содержащий все его вершины.

Обход в глубину (depth-first search).

```
procedure dfs ( v : вершина )
var
   w : вершина;
begin
   mark[v] := visited;
   for w ∈ L[v] do
        if mark[w] = unvisited then dfs ( w );
end;
```

Обход в глубину (breadth-first search).

```
procedure bfs ( v : вершина )
var
   Q : QUEUE;
   x, y : вершина;
begin
   mark[v] := visited;
   Q.Enqueue ( v );
   while not Q.empty() do
        begin
        x := Q.Dequeue();
        for для каждой y, смежной с x do
             if mark[y] := unvisited then
                  begin
                  mark[y] := visited;
                  Q.Enqueue ( y );
                  end;
        end;
end;
```

Обе процедуры должны вызываться следующим образом.

```
for v := 1 to n do
   mark[v] : = unvisited;
for v :=1 to n do
   if mark[v] = unvisited then  dfs ( v ){ bfs ( v ) } ;
```

Обе процедуры строят остовной лес – множество остовных деревьев.

3) Построения остовного дерева минимальной стоимости.

Стоимость ребра – числовая метка, которой помечено ребро.

Стоимость остовного дерева – сумма стоимостей всех ребер, входящих в дерево.

Основное дерево минимальной стоимости (ОДМС) – основное дерево (ОД), стоимость которого минимальна.

Для построения ОДМС используются алгоритмы:

- Прима;
- Крускала;

Алгоритм Прима основывается на свойстве ОДМС:

Пусть G=(V, E) – связанный граф с заданной функцией сложности. V – множество вершин, U – некоторое подмножество множества вершин и (u, v) – ребро минимальной стоимости, такое, что u∈U и v∈(V-U). Тогда для графа G существует остовное дерево минимальной стоимости, содержащее ребро (u, v).

Алгоритм Прима.

```
procedure Prim (G : граф; var T : множество ребер )
var
     U : множество вершин;
     u, v : вершина;
begin
     T := ∅;
     U := {1};
     while U ≠ V do
          begin
          нахождение ребра (u, v) наименьшей стоимости, такого, что u∈U
               и v∈(V-U);
          T := T + { (u, v) };
          U := U + { v };
          end;
end;
```

8. Структуры данных и алгоритмы для внешней памяти.

8.1. Особенности работы с внешней памятью

Внешняя память – запоминающие устройства, не являющееся устройством основной памяти (по модели фон Неймана). Для операций с внешней памятью характерны следующие особенности:

1. Обращение к внешней памяти не возможно как к основной – процессор не способен адресовать данные на устройствах внешней памяти. Поэтому для них предусматривается следующая последовательность действий:
   a. Загрузить данные с внешнего носителя в память.
   b. Обработать.
   c. Сохранить результат на устройство ВП.

   (Если п. b выпустить, то получим копирование).
2. Для большинства внешних устройств самой медленной является операция позиционирования (поиска). С этой особенностью борются следующими методами:
   - последовательный доступ;
   - блочное чтение и запись;
3. Операция ввода- вывода с ВУ очень медленны по сравнению с оперативной памятью. Контрмеры:
   - использовать контроллеры ввода-вывода и прямой доступ к памяти;
   - сокращать число операций ввода-вывода;

8.2 Файлы внешних устройств

Для хранения информации на ВУ очень продуктивной оказалась концепция файла – связанного списка блоков.

Современное понятие файла – упорядоченная последовательность символов (байтов).

Над файлами, как правило, определены следующие операции:

read(f, buf, size);

write(f, buf, size);

seek(f, position);

Плюс добавочные:

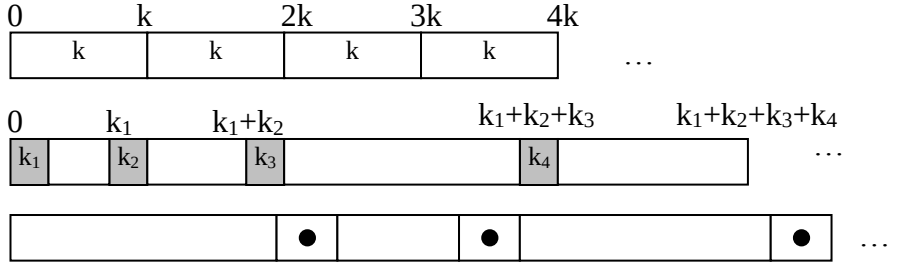
open(f, name); close(f);

Для каждого файла существует, так называемая, текущая позиция, которую можно быть явно или неявно изменять.

8.3. Методы организации файлов.

Как правило, информация в файлах хранится в виде набора **записей**. Обычно различают записи постоянной, переменной и произвольной длины.

1. Записи постоянной длины. Имеют фиксированный размер. Положение i-ой записи может быть вычислено как i·size (если отсчитывается с 0).
2. Записи переменной длины. Имеют размер, изменяющийся в некоторых пределах (от 0 до n). В начале такой записи имеется поле, указывающее размер записи. Для поиска i-ой записи достаточно читать только заголовки с размерами.
3. Записи произвольной длины. Не имеют определенного размера. Признаком конца записи является некоторое специальное значение. Для перехода к следующей необходимо прочесть всю запись.



Кроме того, некоторые записи могут быть **закрепленными** – неперемещаемыми (например, если на записи идет ссылка откуда-то извне).

8.3.1. Простая организация данных.

Записи хранятся в произвольном порядке. Для поиска нужной перебираем все записи. Добавление происходит в конец файла. При удалении происходит перезапись конца. Если записи являются закрепленными, то удаляемая запись помечается специальным значением.

Иногда бывает выгодно хранить файлы отсортированными.

Внешняя сортировка.

Для внешних данных применяют, так называемую, сортировку слиянием.

Идея сортировки слиянием состоит в следующем.

Пусть файл состоит из сортированных последовательностей записей длины k (называемых сериями). Из любых двух серий длины k можно получить серию длинной 2k, осуществив **слияние**. Характерной особенностью слияния является то, что для него достаточно последовательного чтения и записи.

Обычно процесс сортировки начинают с двух файлов, состоящих из серий длины 1. Получаем два файла с сериями 2, потом два файла с сериями 4, 8 … Последние серии могут оставаться неполными.

Пример.

```
28    3    93    10    54    65    30    90    10    69    8    22
31    5    96    40    85    9     39    13    8     77    10

38    31   | 93    96   | 54    85   | 30    39   | 8     10   | 8     10
3     5    | 10    40   | 9     65   | 13    90   | 69    77   | 22

3     5    28    31    | 9     54    65    85    | 8     10    69    77
10    40   93    96    | 13    30    39    90    | 8     10    22

3    5    10   28   31   40   93   96   | 8    8    10   10   22   69   77
9    13   30   39   54   65   85   90   |

3  5  8  8  9  10  10  10  13  22  28  30  31  39  40  54  65  69  77  85  90  93  96
```

Существует несколько модификаций сортировки слиянием.

- с предварительной сортировкой. Слияние начинается с серий длины k, которые получаются считыванием блока из k записей в память и сортировки их каким-нибудь алгоритмом (например, быстрой сортировкой).
- многоканальное слияние. Используют, если имеется несколько (m) устройств ВВ. Объединение происходит в серии длиной km.
- многофазная сортировка. Более сложный аналог многоканальной сортировки (требует меньшего количества устройств ВВ).

8.3.2. Индексированные файлы.

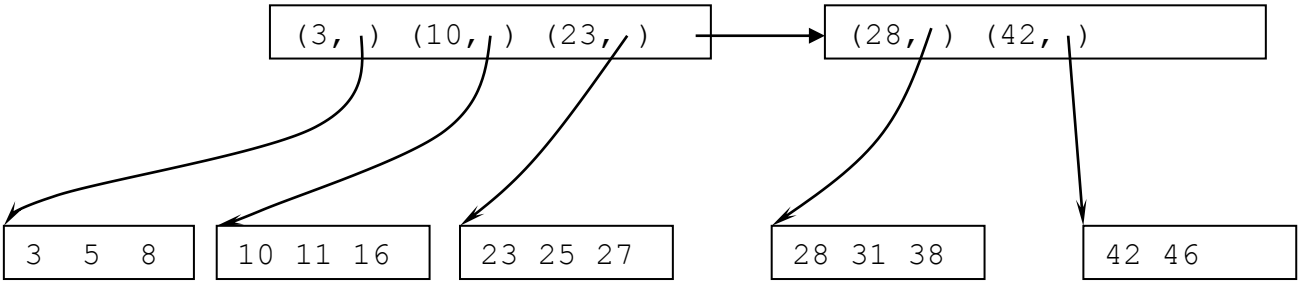
8.3.2.1. Плотный индекс.

При организации файлов с плотным индексом имеется 2 файла:

- основной (содержащий информацию).
- файл индекса, состоящий из пар (x, p), где p – указатель на запись с ключом x.

Файл индекса сортируют по значению ключа. По этому файлу можно организовывать поиск

8.3.2.2. Разряженный индекс.
При этой организации файл индекса состоит из пар (x, b), где b – указатель на блок записей в отсортированном файле, ключ первой записи которого равен x.



Блоки не обязательно должны заполняться целиком.

8.3.3. B-деревья
Под B-деревом обычно понимают дерево, устроенное следующим образом:

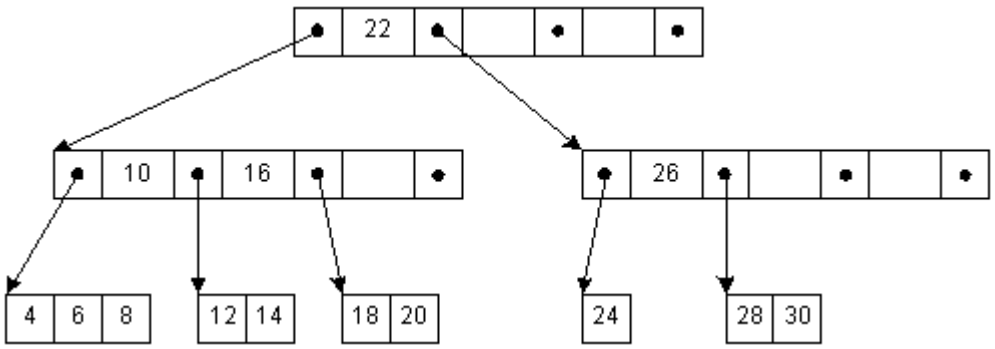
1. Каждая вершина x содержит поля, в которых хранятся:
   a. количество n[x] ключей, хранящихся в ней;
   b. сами ключи в неубывающем порядке $key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$
   c. булевское значение leaf[x], истинное, если x – лист.
2. Если x-внутренняя вершина, то она также содержит n[x]+1 указателей $c_1[x]$, $c_2[x],\ldots, c_{n[x]+1}[x]$ на ее детей. У листьев детей нет и эти поля не определены.
3. Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях:

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$$

где k_i – любой из ключей, хранящихся в поддереве с корнем $c_i[x]$

4. Все листья находятся на одной и той же глубине.

Пример



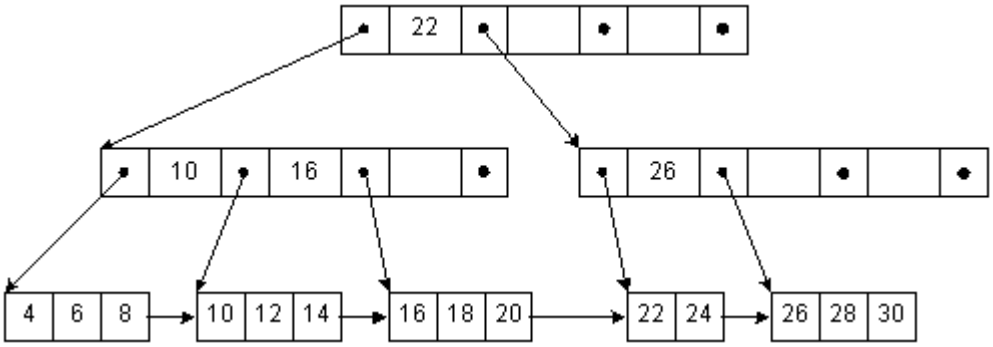
Б-дерево с 3 ключами на узел.

Ключи в внутреннем узле окружены указателями или смещениями записей, отсылающими к ключам, которые либо все больше, либо все меньше окруженного ключа. Например, все ключи, меньшие 22, адресуются левой ссылкой, все большие - правой.
В этом двухуровневом дереве мы можем добраться до любого ключа за три доступа к диску. Если бы мы сгруппировали по 100 ключей на узел, то за три доступа к диску мы могли бы найти любой ключ из 1000000.
Существует и другой подход (B^+-деревья). Их отличительные особенности:

1. Все ключи хранятся в листьях, там же хранится и информационная часть узла.
2. Во внутренних узлах хранятся копии ключей - они помогают искать нужный лист.

3. У указателей смысл немножко не такой, как при работе с обычными Б-деревьями. Левый указатель ведет к ключам, которые *меньше* заданного значения, правый - ключам, которые *больше или равны* (GE).

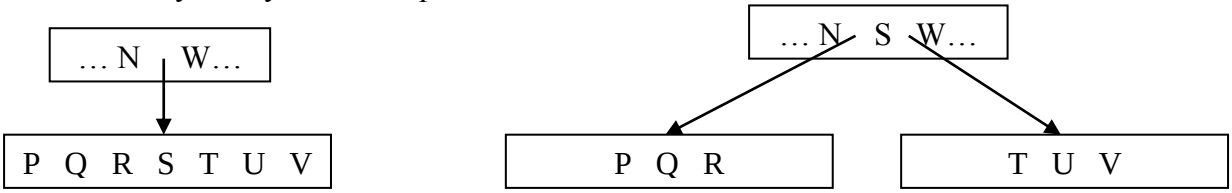
Пример.



$Б^{+}$-дерево

Например, к ключам, меньшим 22, ведет левый указатель, а к ключам от 23 и выше ведет правый. Обратите внимание на то, что ключ 22 повторяется в листе, где хранятся соответствующие ему данные.

Чтобы сохранить свойство сбалансированности для B-деревьев при добавлении и удалении применяют расщепление (рассыпание). Если вершина y, в которую происходит добавление, заполнена, то она разбивается на две, заполненные частично, а ключ-медиана (по которому происходило деление) переходит к предку x вершины y и становится разделителем двух полученных вершин.



Некоторые разновидности B-деревьев

	B-дерево	**B*-дерево**	**B^{+}-дерево**	**B^{++}-дерево**
данные хранятся в	любом узле	любом узле	только в листьях	только в листьях
при вставке - расщепление	1 x 1 –>2 x 1/2	2 x 1 –>3 x 2/3	1 x 1 –>2 x 1/2	3 x 1 –>4 x 3/4
при удалении - слияние	2 x 1/2 –>1 x 1	3 x 2/3 –>2 x 1	2 x 1/2 –>1 x 1	3 x 1/2 –>2 x 3/4

9. Классические алгоритмы.

9.1. Алгоритмы сортировки

Для классификации алгоритмов сортировки можно предложить, например, такую схему (по методическому пособию по ред. Кукушкина Б.А.).

1. Сортировка вставками.
   a. Алгоритм простых вставок
   b. Вставки с убывающим шагом (метод Шелла)
2. Обменная сортировка
   a. Метод пузырька
   b. Модификация метода пузырька
   c. Быстрая сортировка.
   d. Обменная поразрядная сортировка
3. Сортировка выбором
4. Сортировка посредством слияния
5. Карманная сортировка
6. Пирамидная сортировка

9.1.1. Простые вставки

```
procedure insertion_sort
begin
for i:= 2 to n do
     переместить A[I] на позицию j ≤ I такую, что A[I] < A[k] для j ≤ k < I;
end;
```

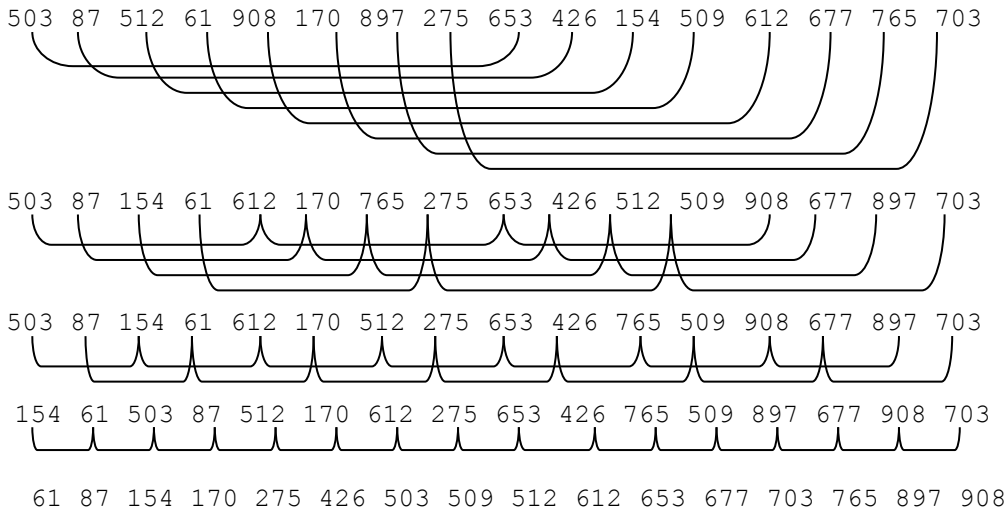
9.1.2. Метод Шелла

Идея метода в следующем.

Сортировке вставками подвергается не весь массив, а пары элементов отстоящих друг от друга на расстоянии n/2 (т.е. A[i] и A[n/2+i]); затем 4-ки отстоящие на n/4 (A[i], A[n/4+i], A[2n/4+i], A[3n/4+i] для 1≤i≤n/4); потом 8-ки… и т.д.

```
procedure shell_sort
begin
     step:= n div 2;
     while step>0 do
          for i:=1 to n div 2 do
               отсортировать последовательность A[i], A[step+i], A[2·step+i],…
                  методом простых вставок;
end;
```

Пример.



9.1.3. Метод пузырька.

```
procedure bubble_sort
begin
     last := n;
     while last ≠ 0 do
          t:=0;
          for j:=1 to last-1 do
               if A[j]>a[j+1] then
                    A[j]↔A[j+1]; t:=j;
          last:=t;
end;
```

9.1.4. Шейкерная сортировка

Метод пузырька, но просмотр идет как с начала в конец, так и в обратную сторону.

9.1.5. Быстрая сортировка.

```
procedure quick_sort ( i, j : integer )
begin
  if A[i]...A[j] имеют хотябы два различных ключа then
        v:=median (A[i], A[i+1], ..., A[j])
        переставить элементы A[i]...A[j], так чтобы для некоторого k, (i≤k≤j),
             A[i]...A[k-1] меньше v, а A[k]...A[j] – больше v.
        quick_sort(i, k-1);
        quick_sort(k, j);
end.
```