Алгоритмы и структуры данных

Лекция 9.1

Часть 1

Быстрый Поиск. Деревья поиска

Новая тема
Быстрый поиск

СД для организации данных с эффективной реализацией набора операций, в т.ч. таких, как

- Поиск заданного элемента
- Добавление (вставка) заданного элемента
- Удаление заданного элемента
- Упорядочение

Реализация в массиве (в упорядоченном массиве).

++ и -- !

Быстрый поиск

Деревья поиска

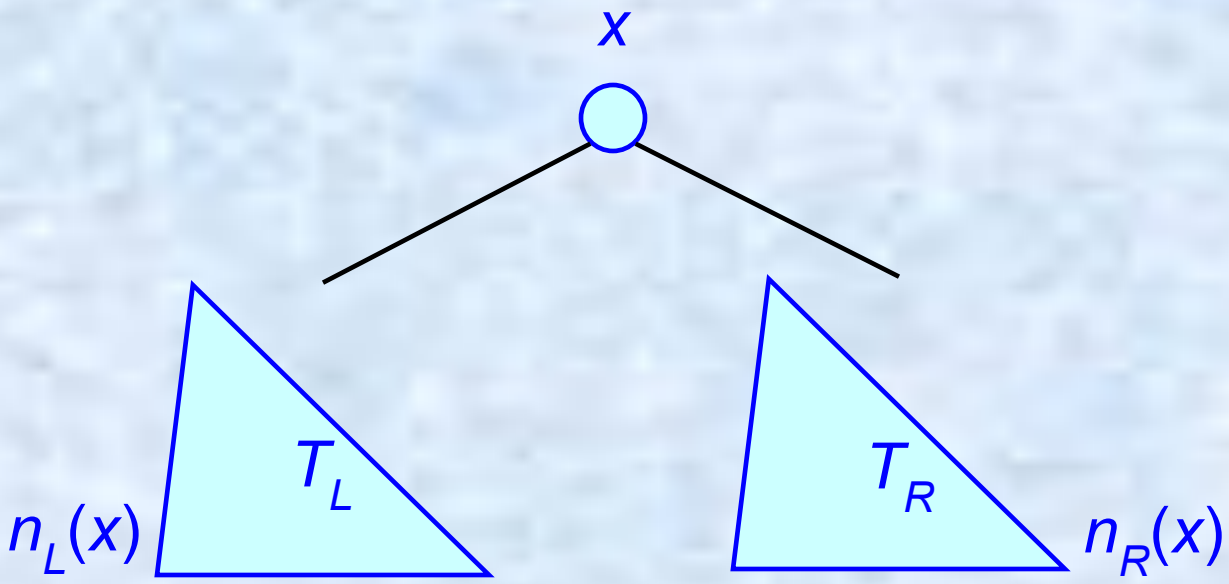
Идеально сбалансированные бинарные деревья

Идеально сбалансированным назовем такое бинарное дерево T, что *для каждого его узла* x справедливо соотношение

$$|n_L(x) - n_R(x)| \le 1,$$

где $n_L(x)$ – количество узлов в левом поддереве узла x, а $n_R(x)$ – количество узлов в правом поддереве узла x.

Идеально сбалансированные бинарные деревья

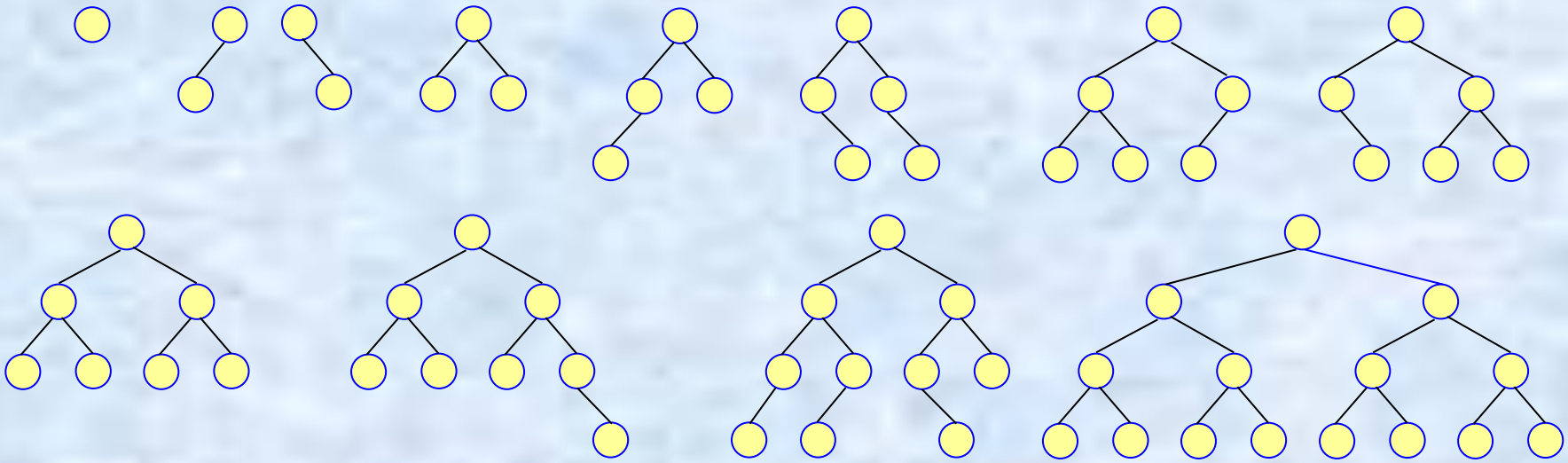


Инвариант такого БД: $\forall\ x \in T$

$$|n_L(x) - n_R(x)| \le 1,$$

где $n_L(x)$ ($n_R(x)$) – количество узлов в левом (правом) поддереве узла x

Примеры идеально сбалансированных деревьев



В идеально сбалансированном дереве число узлов n и высота дерева h связаны соотношением

$$2^{h-1} < n+1 \le 2^h$$

или

$$h = \lceil \log_2(n+1) \rceil$$ *

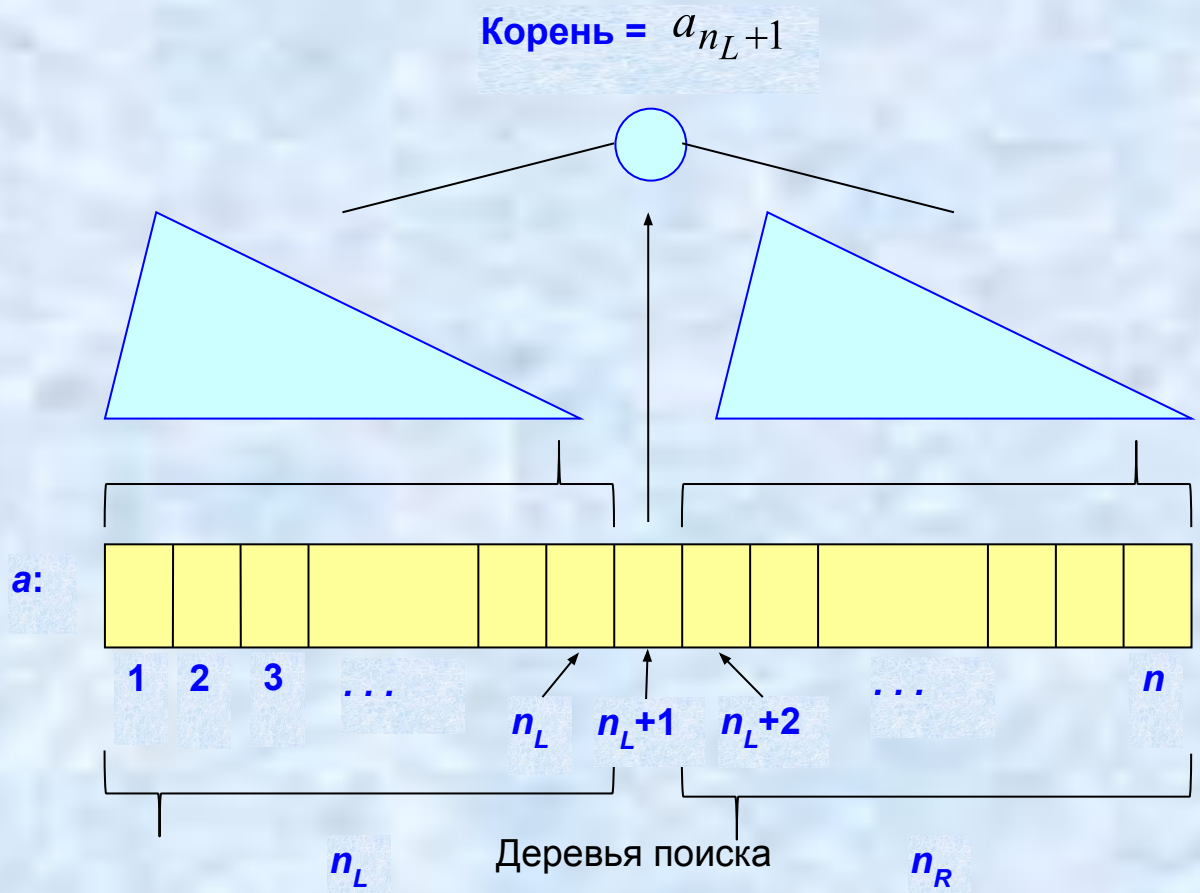
Высота дерева определена так, что при $n = 0$ имеем $h = 0$, а при $n = 1$ имеем $h = 1$

Алгоритм построения идеально сбалансированного дерева по последовательности данных $a_1, a_2, \ldots, a_n$

Пусть, например, данные берутся из потока *infile*.

Идея (рекурсивного) алгоритма

(здесь $n_L = n$ **div** 2 и $n_R = n - n_L - 1$, т. е. $n = n_L + n_R + 1$)



Считаем, что тип данных *binTree*, представляющий бинарное дерево, определен как ранее

```
typedef char base;
struct node {
    base info;
    node *lt;
    node *rt;
};
typedef node *binTree; // "представитель" бинарного дерева
```

(в том числе определены базовые функции этого типа, например, *isNull, Create, RootBT, Left, Right, ConsBT*)

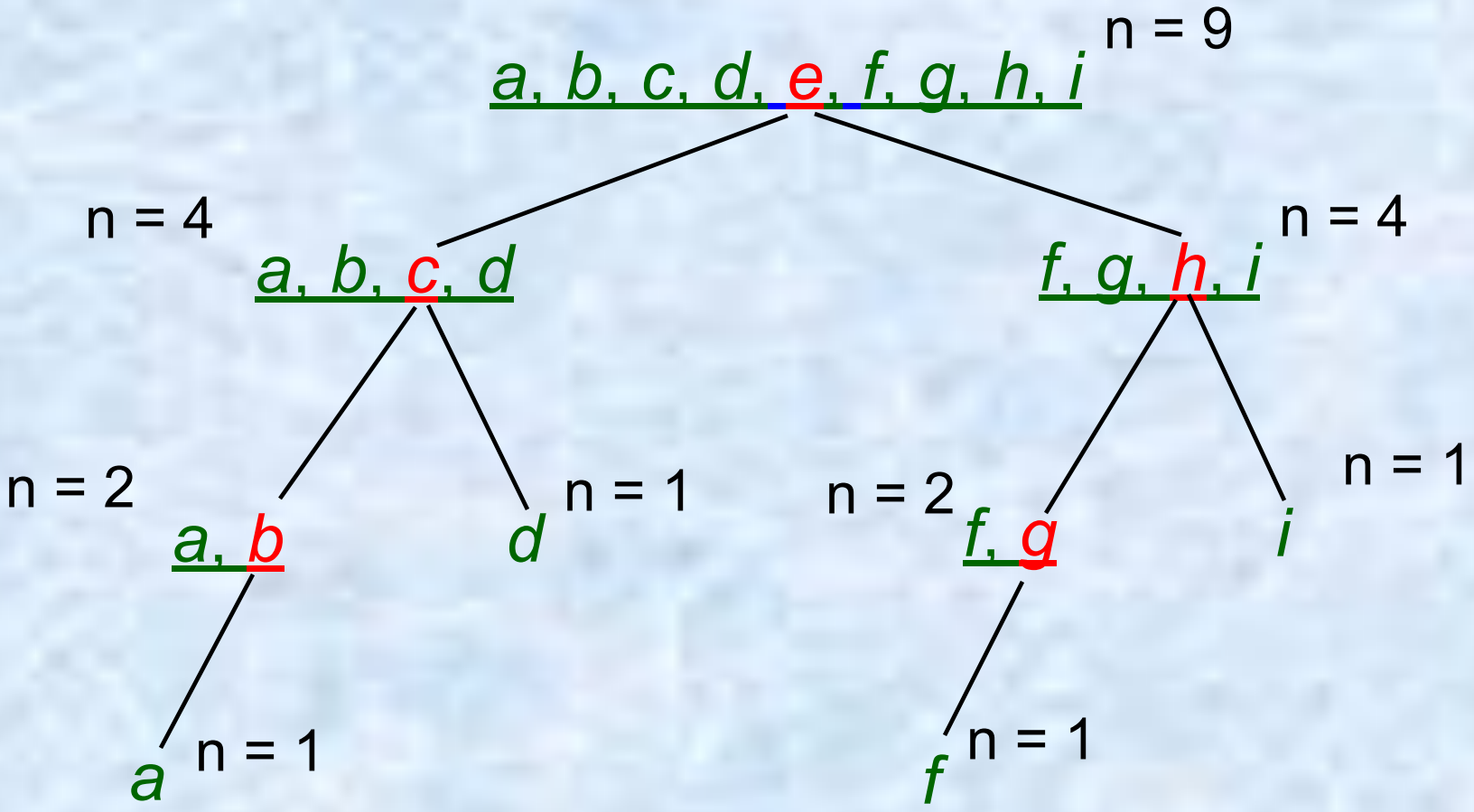
Алгоритм построения идеально сбалансированного дерева по последовательности данных $a_1, a_2, \ldots, a_n$

```
binTree makeTree (unInt n)
// построение идеально сбалансированного дерева с n узлами
{
    unInt  nL, nR;
    binTree   b1, b2;
    base x;

    if (n==0) return Create();
    nL = n/2;  nR = n-nL-1;
    b1 = makeTree (nL);
    infile2 >> x;
    b2 = makeTree (nR);
    return ConsBT(x, b1, b2);
}
```

См. idealBInsTr.cpp

a, b, c, d, e, f, g, h, i

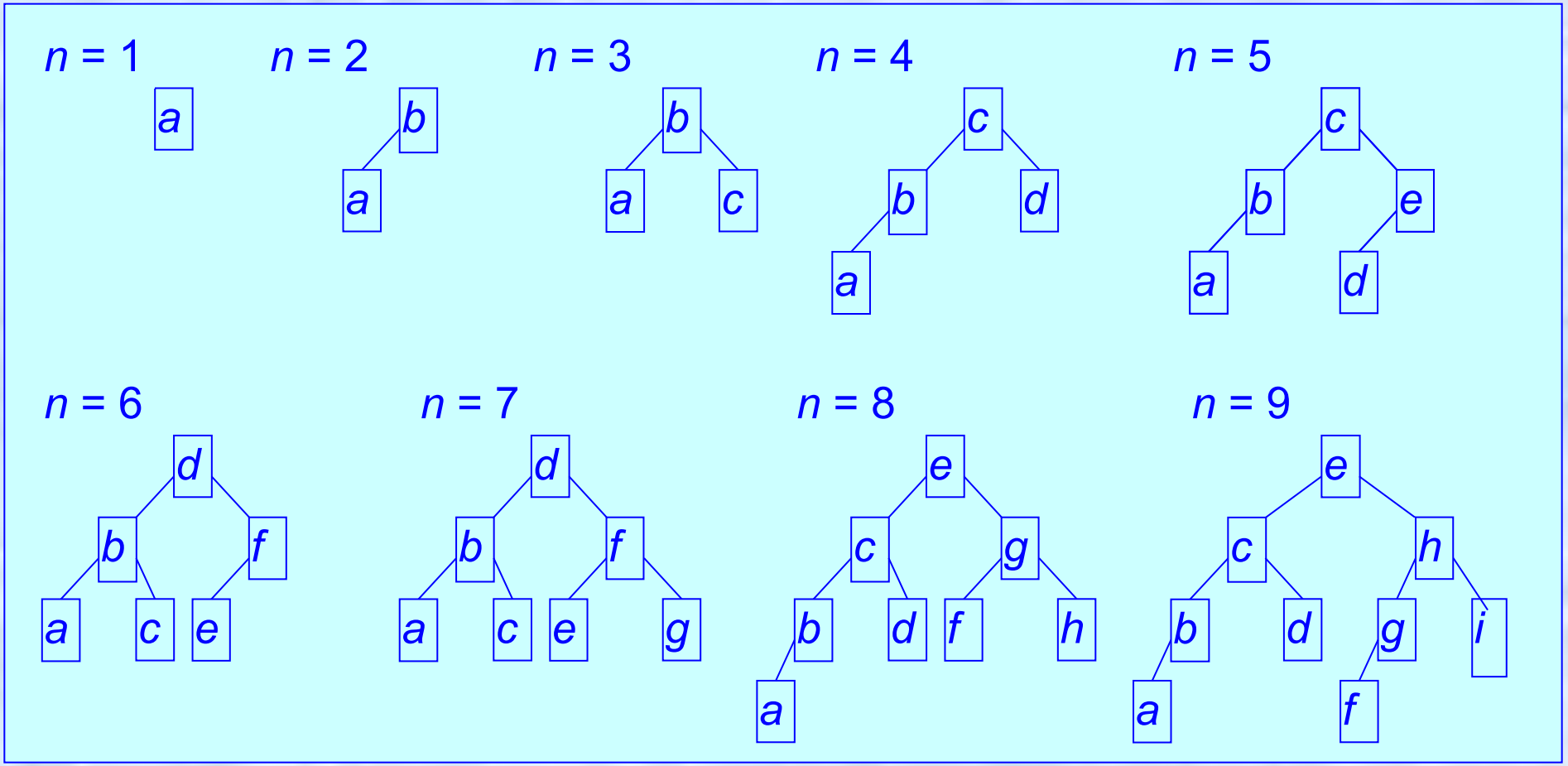


Самостоятельно проанализировать последовательность ввода символов и построения БД

Примеры работы алгоритма. Пусть во входном файле находится последовательность

a, b, c, d, e, f, g, h, i.

Стартовый вызов функции *b = makeTree (n);*



Замечание 1. Алгоритм строит такие идеально сбалансированные деревья, что $n_L(x) - n_R(x) = 0$ или 1, т. е. при $n_L(x) \neq n_R(x)$ именно *левое поддерево содержит на один узел больше.*

Замечание 2. *Структура* дерева определяется только значением параметра n, а *содержимое узлов* зависит от расположения элементов во входной последовательности.

В примере из-за того, что входная последовательность *упорядочена*, все построенные деревья обладают свойством: при обходе этих деревьев «слева направо», т. е. при симметричном или ЛКП-обходе, порождается исходная *упорядоченная* последовательность

(см.демонстрацию выполнения программы)

Упражнение

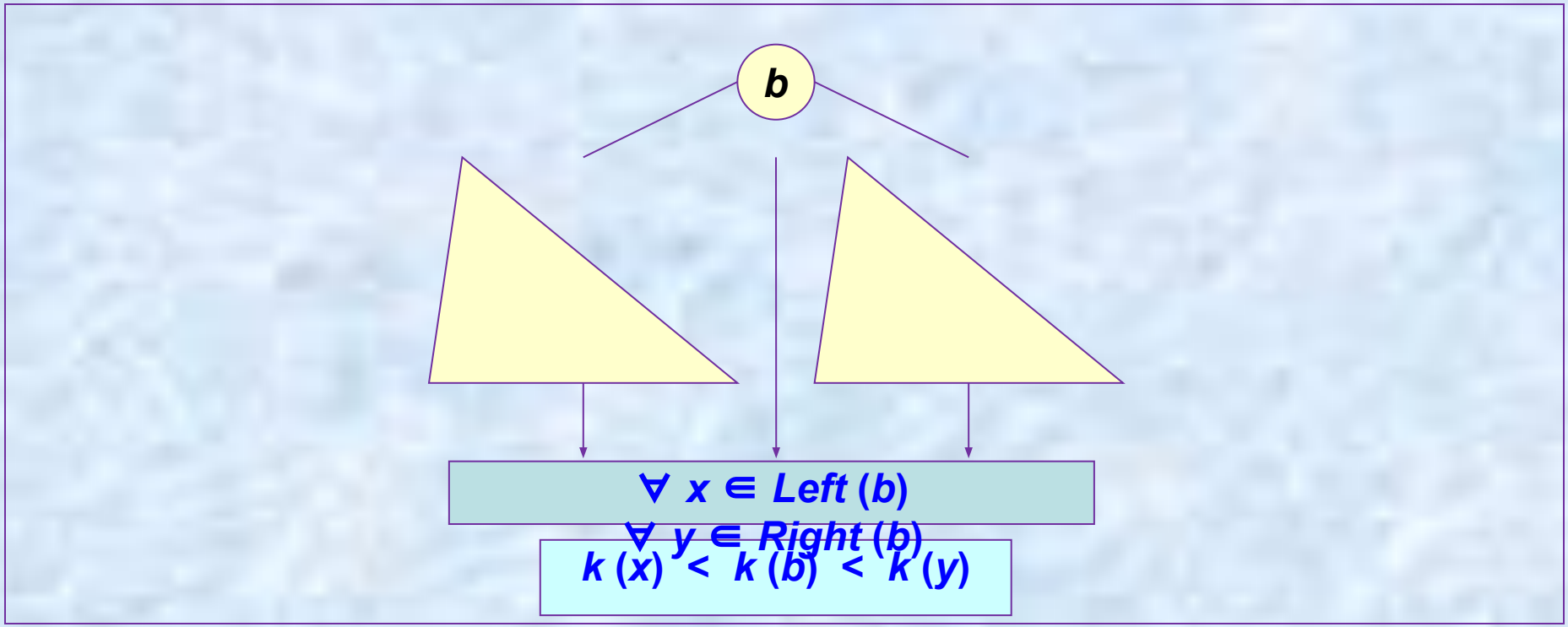
Вариант с КЛП-схемой	Вариант с ЛПК-схемой

if (n==0) return Create();
nL = n/2; nR = n-nL-1;
 	
infile2 >> x;
b1 = makeTree (nL);
b2 = makeTree (nR);
return ConsBT(x, b1, b2);
	if (n==0) return Create();
	nL = n/2; nR = n-nL-1;
	
	b1 = makeTree (nL);
	b2 = makeTree (nR);
	infile2 >> x;
	return ConsBT(x, b1, b2);

Бинарные деревья поиска (БДП)

Пусть $k(b)$– значение *ключа* в узле b дерева T.

Инвариант БДП: условие для каждого узла $b \in T$



Инвариант БДП (*T*: *BSTree*):

Пусть k – значение ключа в узле, тогда в левом поддереве этого узла нет узлов с ключами, большими или равными k, а в правом поддереве – нет узлов с ключами, меньшими или равными k.

И это верно *для каждого узла* дерева.

Формальная запись этого условия:

$$(\forall\ b \in T:$$
$$(\textbf{not}\ Null\ (Left\ (b))) \rightarrow (\forall\ x \in Left\ (b):\ k(x) < k(b)))$$
$$\&$$
$$(\textbf{not}\ Null\ (Right\ (b))) \rightarrow (\forall\ y \in Right\ (b):\ k(b) < k(y))))$$

Бинарные деревья поиска

Операция *поиска* заданного элемента *x*: *base* в непустом БДП *b*: *BSTree* производится *рекурсивно* :

Если *k*(*x*) = *k*(*b*), то элемент *x* находится в корне дерева *b*. Поиск закончен «успешно» – элемент найден.

Если *k*(*x*) < *k*(*b*), то продолжить ***поиск*** в левом поддереве *Left* (*b*).

Если *k*(*x*) > *k*(*b*), то продолжить ***поиск*** в правом поддереве *Right* (*b*).

Если выбранное поддерево оказывается *пустым*, то поиск завершается «неудачно» – элемента *x* в дереве *b* нет.

Операция *поиска* заданного элемента *x*: *base* в непустом БДП *b*: *binTree*

```
binTree Locate (base x, binTree b)
// b – должно быть БДП
{  base r;
   if (isNull(b)) return NULL;
   else {
    r = RootBT(b);
    if (x==r) return b;
    else if (x<r) return Locate(x,Left(b));
    else return Locate(x,Right(b));
   }
}
```

Поскольку в этой рекурсивной функции каждый экземпляр порождает (альтернативно) не более одного следующего рекурсивного вызова, то рекурсия легко преобразуется в итерацию:

```
base r;
bool found = false;
while(!isNull(b) && !found)
{   r = RootBT(b);
    if (x==r) found = true; // b - искомый узел
    else if (x<r) b = Left(b);
    else b = Right(b);
}
if (found) return b;
else return NULL;
```

bstLoc.cpp

Выход из цикла при found = false говорит об отсутствии в дереве искомого элемента, при этом возвращается NULL

bstLoc.cpp

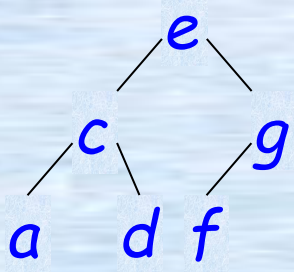
Более короткий вариант того же цикла (без переменных *r* и *found*, но предполагающий режим неполного вычисления булевских выражений) есть

```
while (!isNull(b) && !(x==RootBT(b)))
{
    if (x < RootBT(b)) b = Left(b); else b = Right(b);
}
return b;
```

- Очевидно, что ***время поиска*** (количество шагов по дереву) зависит от положения искомого узла в дереве и *в худшем случае* пропорционально высоте дерева.
- *С* этой точки зрения наиболее предпочтительными являются ***идеально сбалансированные деревья***.
- Однако, как показывает следующий пример, при добавлении или исключении узлов дерева ***поддержание*** структуры идеально сбалансированного дерева требует больших затрат.

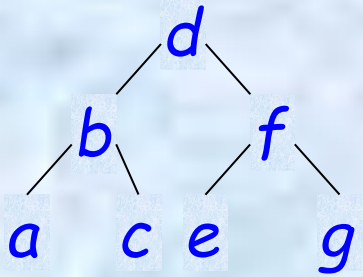
Действительно, пусть имеется идеально сбалансированное дерево из элементов *a, c, d, e, f, g*:

Это БДП



При добавлении узла *b* это дерево должно превратиться в следующее

Это БДП



При этом **ни одна из связей** «отец-сын» не сохраняется, т. е. потребуется перестройка всего дерева!

Далее

будут рассмотрены несколько видов БДП, коррекция которых

(добавление или исключение узлов) производится более экономным способом.

Случайные бинарные деревья поиска

Добавление элемента в БДП

Введем дополнительное поле записи *count*, в котором отмечаются *повторные попытки вставки* элемента в дерево:

```
struct node {
base info;
unsigned int count;
node *lt;
node *rt; }
```

Процедура *SearchAndInsert* – поиска и вставки элемента *x* в дерево *b* :

- *в случае успешного поиска* увеличивает счетчик *count* в найденном узле,
- *в случае неудачного поиска* добавляет лист в *подходящее* (т. е. с сохранением инварианта БДП) место дерева поиска.

```
void SearchAndInsert (base x, binSTree &b)
{if (b==NULL) {
      b = new node;
      b ->info = x;
      b ->count = 1;
 }
 else if(x < b->info) SearchAndInsert (x, b->lt);
 else if(x > b->info) SearchAndInsert (x, b->rt);
 else b->count++;
}
```

bst3/
bst_implementation.cpp

bst3.cpp

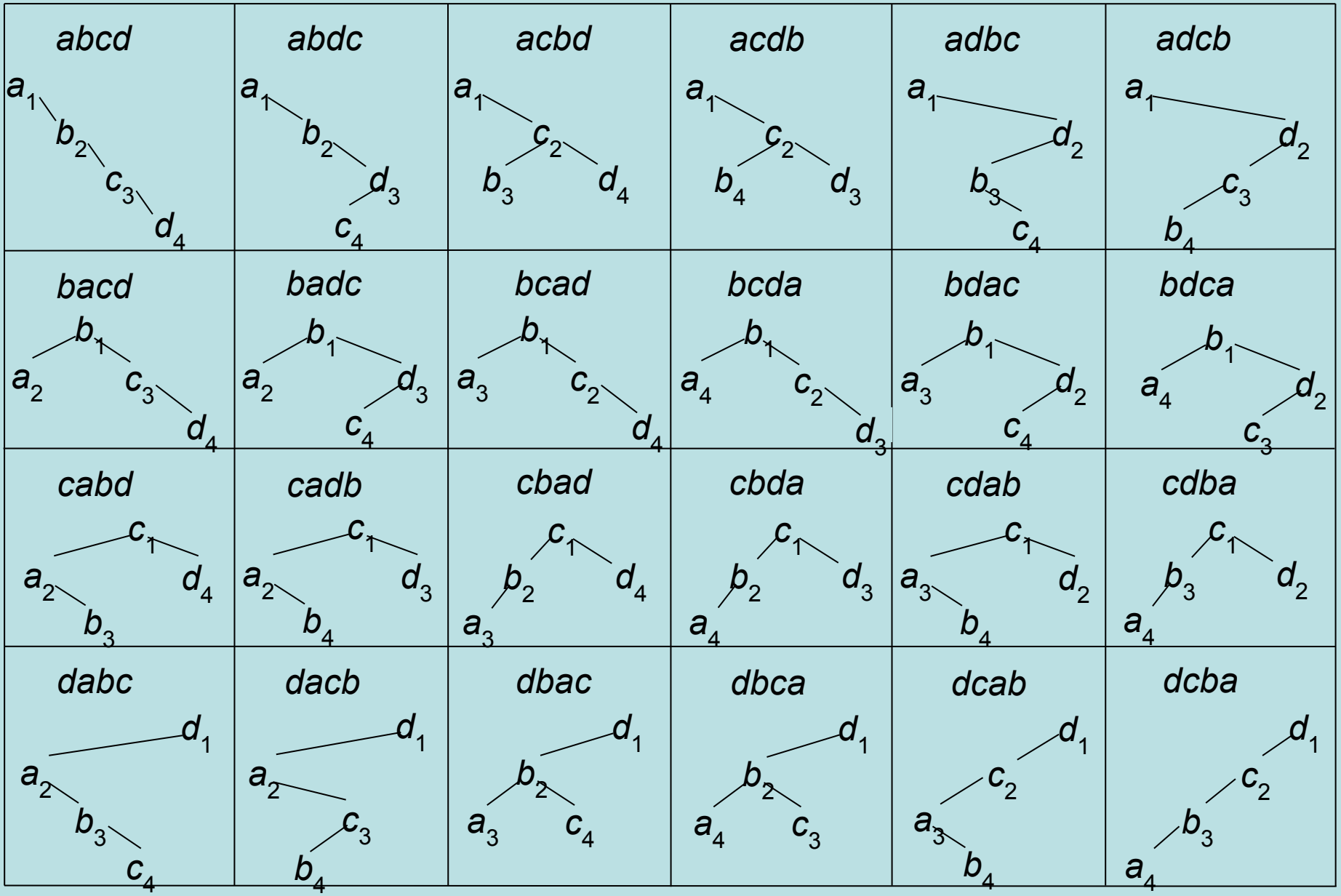
Пусть во входном потоке находится последовательность элементов, по которой функция *SearchAndInsert* строит БДП:

```
b = Create();
while (infile2 >> c)
{   SearchAndInsert (c, b);
}
```

БДП, построенное таким способом, называется *случайным БДП*

Структура *случайного* БДП полностью зависит от того («*случайного* ») порядка, в котором элементы расположены во входной последовательности (во входном файле).
В качестве простейшего примера рассмотрим последовательность из четырех элементов *a, b, c, d.*
Имеется 4! = 24 перестановки из четырех элементов и, следовательно, 24 варианта входной последовательности.

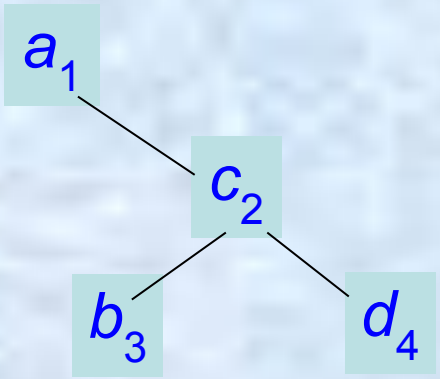
abcd	abdc	acbd	acdb	adbc	adcb
bacd	badc	bcad	bcda	bdac	bdca
cabd	cadb	cbad	cbda	cdab	cdba
dabc	dacb	dbac	dbca	dcab	dcba



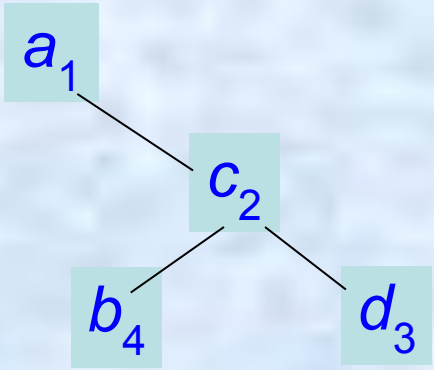
Все случайные БДП для четырёх элементов *a*, *b*, *c*, *d*.

Анализ структуры БДП

acbd

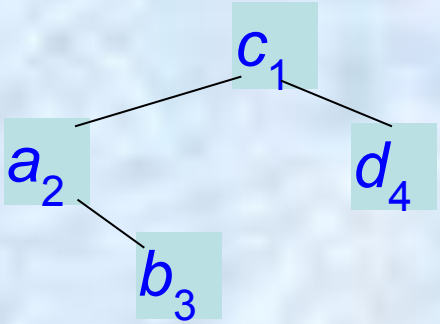


acdb

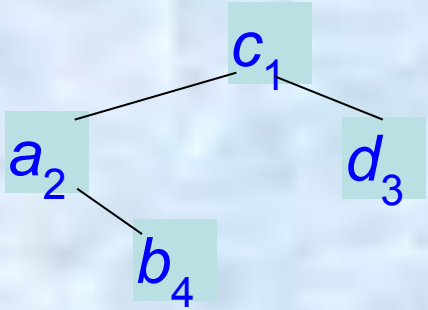


1) *bacd*, *bcad*, *bcda*; 2) *badc*, *bdac*, *bdca*;
3) *cabd*, *cadb*, *cdab*; 4) *cbad*, *cbda*, *cdba*.

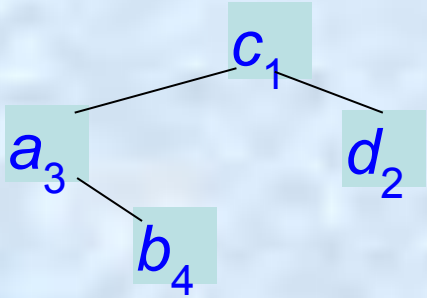
cabd



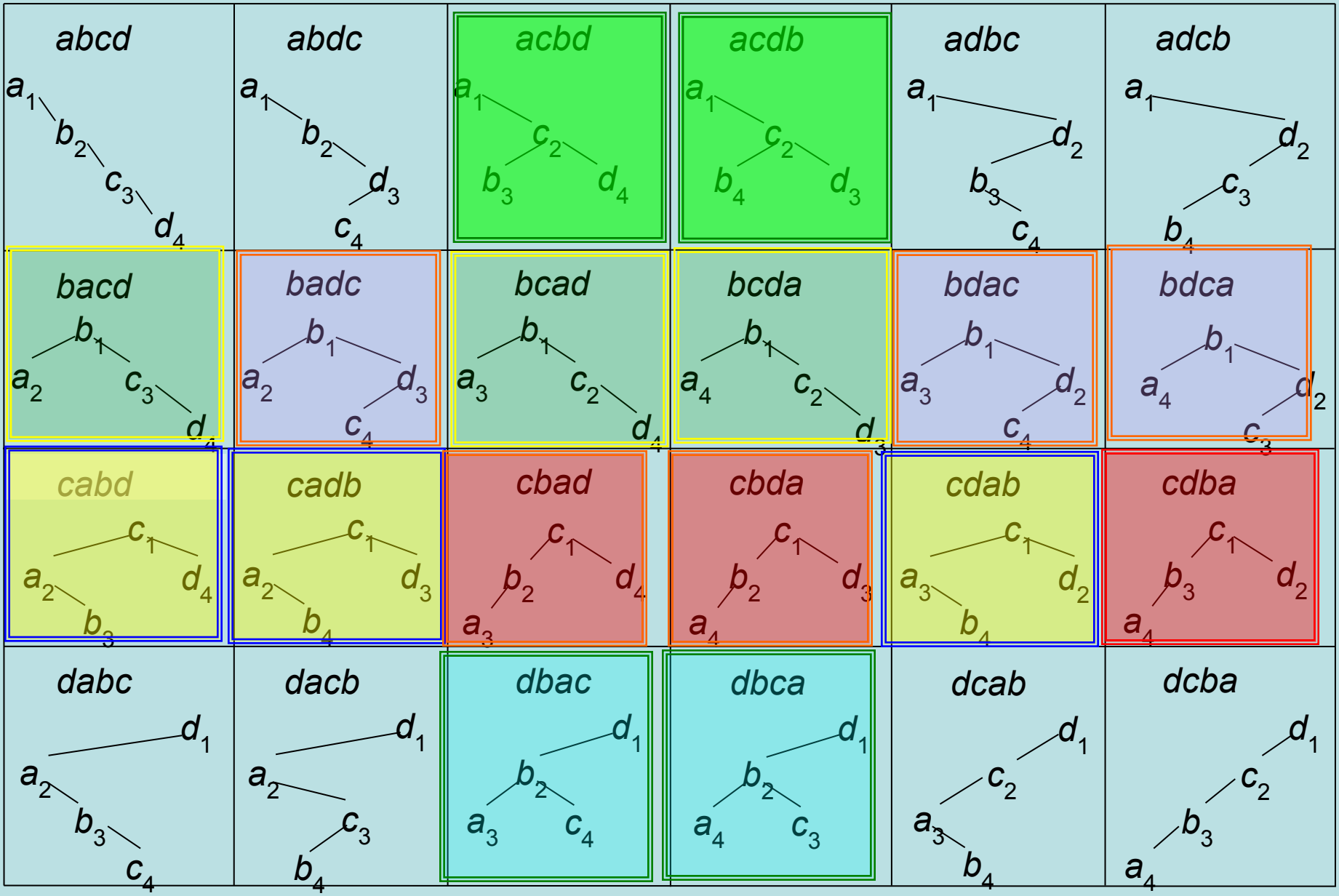
cadb



cdab



См. далее

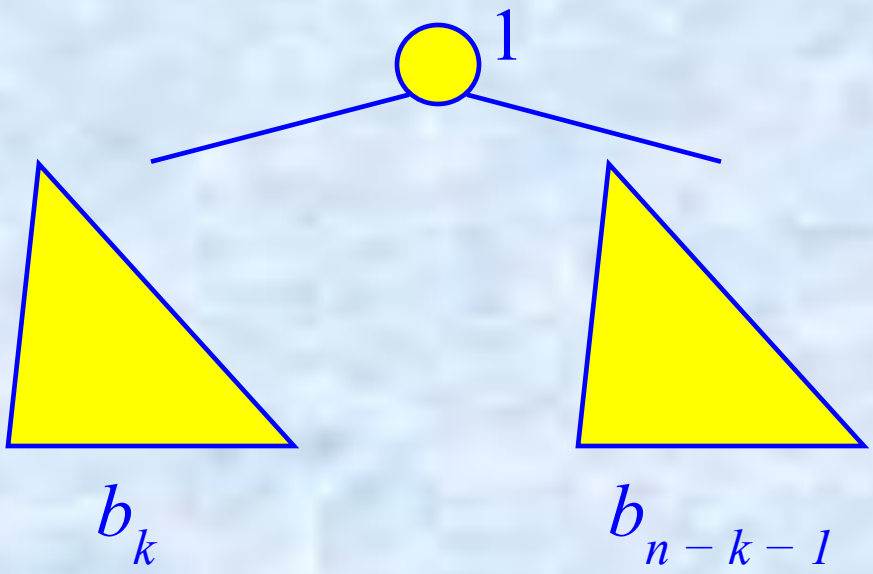


Все случайные БДП для четырёх элементов *a*, *b*, *c*, *d*.

Из 24 бинарных деревьев в этом примере имеется 12 *идеально сбалансированных деревьев* и 14 *структурно различных бинарных деревьев* (например, соответствующих перестановкам *abcd*, *abdc*, *acbd*, *adbc*, *adcb*, *bacd*, *badc*, *cabd*, *cbad*, *dabc*, *dacb*, *dbac*, *dcab*, *dcba*).

Число b_n структурно различных бинарных деревьев с n узлами

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-k-1} = b_0 b_{n-1} + b_1 b_{n-2} + \ldots + b_{n-2} b_1 + b_{n-1} b_0, \quad b_0 = 1, b_1 = 1.$$



$k \in 0..(n-1)$

Начальное условие $b_0 = 1$. Далее

$$b_1 = b_0\, b_0 = 1,$$

$$b_2 = b_0\, b_1 + b_1\, b_0 = 2,$$

$$b_3 = b_0\, b_2 + b_1\, b_1 + b_2\, b_0 = 5.$$

$$b_4 = b_0\, b_3 + b_1\, b_2 + b_2\, b_1 + b_3\, b_0 = 14.$$

Оказывается [7 - Грэхем и др. Конкретная математика: Основание информатики, с. 393], что решением этого рекуррентного уравнения являются так называемые

числа Каталана $b_k = C_k$,

где $C_k = (2\ k \mid k) / (k+1)$,

а запись $(n \mid m)$ обозначает биномиальный коэффициент

$$(n \mid m) = n!/(m!\,(n-m)!).$$

См. также 1.6.10 и 1.7.4 в книге

При больших значениях k удобно использовать формулу Стирлинга

$$k! \approx (2\pi)^{1/2} k^{k+\frac{1}{2}} e^{-k}$$

Тогда для чисел Каталана при больших значениях n справедливо

$$C_n \approx 4^n / n\sqrt{\pi n}$$

т. е. число *структурно различных* бинарных деревьев есть экспоненциальная функция от n.

Несколько первых чисел Каталана

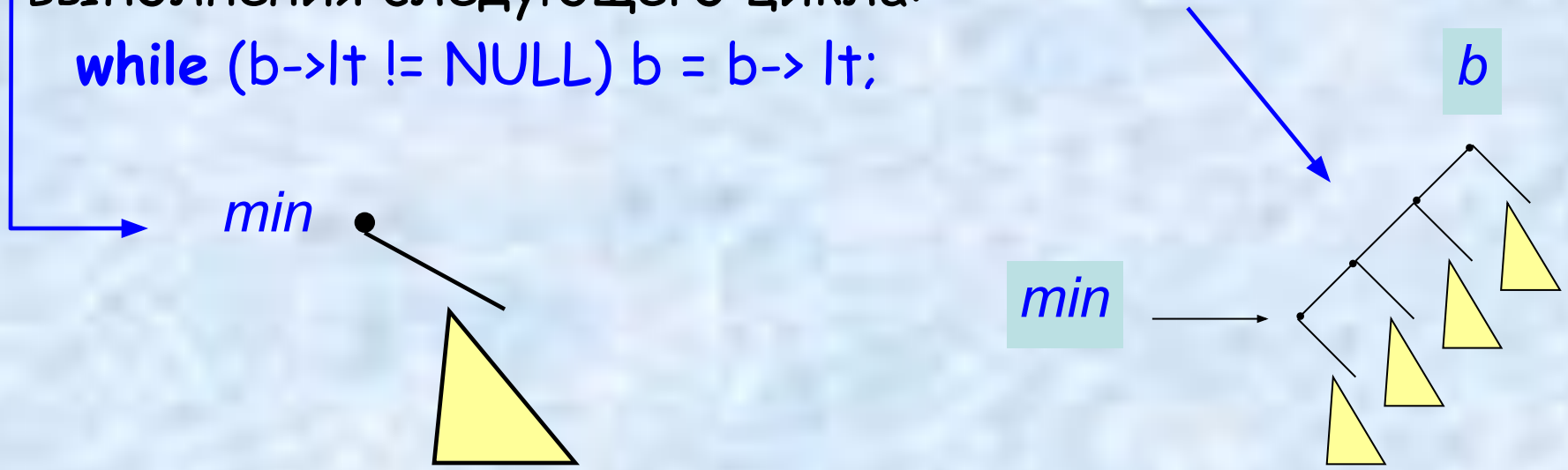
n	0	1	2	3	4	5	6	7	8	9	10
C_n	1	1	2	5	14	42	132	429	1430	4862	16 796

Конец отступления про числа Каталана

Операция поиска минимального элемента в БДП

- Если в дереве *b* *левое* поддерево пусто, то минимальное значение находится в корне.
- Если же *левое* поддерево *не пусто*, то минимальное значение находится в самом левом элементе левого поддерева, который может быть найден после выполнения следующего цикла:

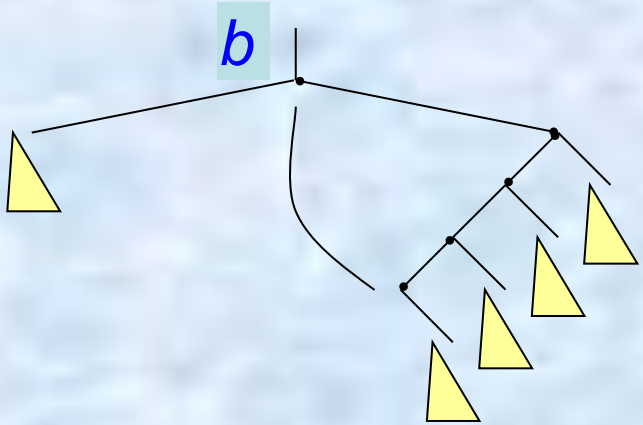
```
while (b->lt != NULL) b = b-> lt;
```



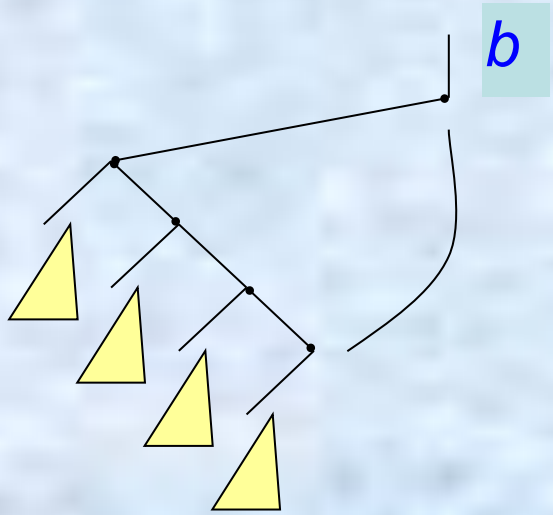
Удаление элемента из случайного БДП

Удалить элемент из случайного БДП проще всего, если этот элемент находится *в листе дерева*. Тогда данный лист непосредственно удаляется.

Если же удаляемый элемент находится *во внутреннем узле* *b*, то в ситуации b -> rt != **NULL** следует найти *минимальный элемент правого поддерева*, рекурсивно удалить его и заменить им содержимое узла *b*. Этот процесс схематично показан на рисунке.



В ситуации b -> rt == NULL (поскольку узел *b* – не лист, то, следовательно, b -> lt != NULL) находим максимальный элемент левого поддерева, рекурсивно удаляем его и заменяем им содержимое узла *b*.



Количество шагов в рассмотренных операциях *вставки*, *удаления* и *нахождения минимального элемента* в случайном БДП в *худшем* случае равно высоте дерева. Зависимость высоты случайного БДП от количества узлов дерева рассмотрена далее.

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ

КОНЕЦ ЛЕКЦИИ