Лекция №6

ПОРАЗРЯДНАЯ СОРТИРОВКА

Сортировка, построенная на обработке данных (значения ключа данных) по частям (порции, разряду) за один шаг, называется поразрядной.

Алгоритмы построены на сортировке объектов с ключами, где ключи рассматриваются как числа представленные в некоторой системе счисления и обработка чисел осуществляется порциями (цифрами) за один шаг.

Группа этих алгоритмов является не устойчивой.

Существует два базовых подхода к поразрядной сортировке:

1. По старшей цифре (поразрядная сортировка most significant digit, MSD)
   (для строковых ключей – по алфавиту)
2. По младшей цифре (поразрядная сортировка least significant digit, LSD)
   (в основном для числовых ключей)

В зависимости от контекста ключом в поразрядной сортировке может быть строка или слово.

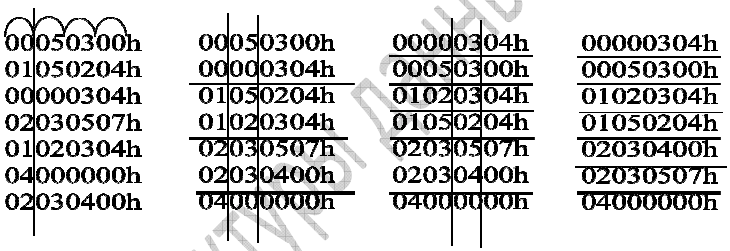
Строка – это последовательность разрядов переменной длины.

Слово – это последовательность разрядов фиксированной длины.

Алгоритмы поразрядной сортировки являются *не устойчивыми*.

Предполагается, что нумерация разрядов начинается слева со значения 0.

Например,
- ключ представляет собой 4-байтное значение;
- цифрой является однобайтное значение.



Числа представлены в шестнадцатеричной системе счисления для удобства демонстрации.

Разбиение ключа на порции:

```
int digit(int value, int nomer) {
   return ((value >> 8*(3-nomer)) & 255);
}
```

```
   00000000 00000101 00000011 00000000
>> 8
   00000000 00000000 00000101 00000011
&  00000000 00000000 00000000 11111111
   00000000 00000000 00000000 00000011
```

$$(int)(value/R^{(N-1-nomer)}) \% R$$

Алгоритм двоичной быстрой поразрядной сортировки:

(Поскольку разряд – это двоичная цифра, то происходит разделение на две части набора данных на каждом разряде, отсюда и применяется быстрая сортировка)

```
public static void sort(byte Array[]) {
        dopSort(Array, 0, Array.length-1, 0);
}
public static void dopSort(byte value[], int left, int right, int position)  {
        int i = left,
            j = right;
        if  ((right <=left) || (position > 8))
                return;
```

```
        while (j != i)   {
                while ((digit(value[i], position) == 0) && (i < j))
                        i++;
                while ((digit(value[j], position) == 1) && (i < j))
                        j--;
                byte temp = value[i];
                value[i] = value[j];
                value[j] = temp;
        }
        if (digit(value[right], position) == 0)
                j++;
        dopSort(value, left, j-1, position+1);
        dopSort(value, j, right, position+1);
    }
```

Алгоритм поразрядной сортировки MSD

Суть – использование алгоритма распределяющего подсчета для сортировки не двоичных цифр ключей. Для более эффективной работы возможно добавление сортировки вставками, если количество оставшихся цифр (разрядов) ключей меньше некоторого установленного значения.

```
                // выделение цифрового символа как разряда
        static String dop[] = new String[count];
        public static int digit(String value, int nomer) {
                return (value.charAt(nomer) - '0');
        }
```

// длина строки 5 символов (ключ – слово);
// диапазон цифровых символов [0; 4]

(Поскольку количество возможных значений разряда равно 5, то происходит разделение на пять групп на каждом разряде, отсюда применяется алгоритм распределяющего подсчета).

```
public static void sort(String m[], int le, int ri, int pos)  {
        int i, j,
            key[] = new int [6];          // размерность массива частичных сумм на 1 больше от
        if (pos > 4)                      // возможного количества ключей
                return;
        if ((ri-le) <= 0)
                return;
        for (i=le; i<ri; i++)                     // подсчет элементов для каждого ключа и их
                key[digit(m[i], pos)+1] ++;       // запись со смещением
        for(j=1; j<key.length; j++)               // подсчет частичных сумм
                key[j] += key[j-1];
        for(i=le; i<ri; i++)                      // перезапись исходного набора в дополнительный
                dop[le+key[digit(m[i], pos)]++] = m[i];
        for(i=le; i<ri; i++)                      // копирование дополнительного в исходный
                m[i] = dop[i];
                // сортировка по следующему разряду для элементов с нулевым значением текущего
                // разряда
        sort(m, le, key[0], pos+1);
                // сортировка всех остальных полученных групп по следующему разрядку
        for(j=0; j<key.length-1; j++)
                sort(m, le+key[j], le+key[j+1], pos+1);
}
```

ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Пирамидальная сортировка базируется на обработке такой структуры данных как *очередь по приоритету*.

Очередь по приоритету – это структура данных, в которой:

- элементы имеют ключ (приоритет);
- операция вставки добавляет элемент либо в конец набора данных либо в соответствующем порядке;
- операция выборки (удаления) удаляет элемент с наибольшим значением ключа (приоритета).

Применение:

1. Системы планирования заданий в компьютерных системах.
2. Системы моделирования

Реализация операций вставки и удаления очень сильно зависит от представления набора данных: упорядоченный или неупорядоченный, векторное или связанное представление (в виде массива или списка).

Неупорядоченное представление определяет «ленивый» подход к решению задачи удаления элемента (когда выполнение работы откладывается до необходимого момента).

Упорядоченное представление – «энергичный» подход (когда заранее выполняется необходимый объем работ, чтобы обеспечить максимальную эффективность реализации операции удаления).

Пирамидальное представление структуры данных

Сортирующее дерево – это совокупность узлов (элементов) с ключами (приоритетами), образующими полное пирамидально упорядоченное бинарное дерево, представленное в виде массива.

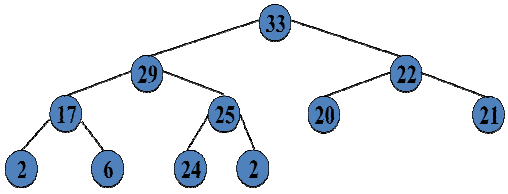
Полное пирамидально упорядоченное бинарное дерево – это структура данных, в которой каждый узел содержит ключ со значением большим или равным значениям ключей узлов-потомков (ключ в каждом узле дерева меньше или равен ключу узла, который является родителем данного узла).

Формирование сортирующего дерева:

1. Создается корневой узел
2. Спускаясь вниз по уровням, перемещаются слева направо, добавляя к каждому узлу предыдущего уровня по два узла текущего уровня.

Например, последовательность значений

33 29 22 17 25 20 21 2 6 24 2



Представление такого дерева в виде массива формируется по следующему правилу: значение элемента с индексом **i** больше или равно значениям элементов с индексами **2i** и **2i+1**, т.е. **i-й** элемент является родителем для элементов, расположенных по индексам **2i** и **2i+1** (левый и правый потомок соответственно).

0	1	2	3	4	5	6	7	8	9	10	11
⨯	33	29	22	17	25	20	21	2	6	24	2

Все алгоритмы для *очередей по приоритету*, представленных в виде сортирующего дерева, работают по правилу:

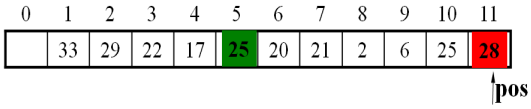
1. Вносится простое изменение.

2. Выполняется проход по пирамиде, внося изменения, связанные с поддержкой сортируемой структуры.

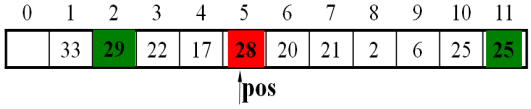
Проход по сортирующему дереву снизу вверх для установки нового порядка структуры данных называется **восходящим**.

Восходящая установка:

Up (*Array*, *pos*)
while *pos* > 1 **and** *Array*[pos/2] < *Array*[pos] **do**
Array[pos] ↔ *Array*[pos/2]
pos ← *pos*/2



а)

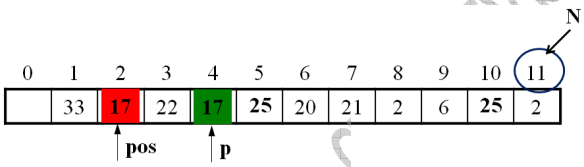


б)

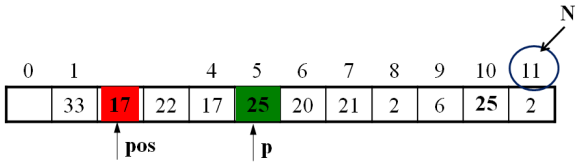
Проход по сортирующему дереву сверху вниз для установки нового порядка структуры данных называется **нисходящим**.

Нисходящая установка:

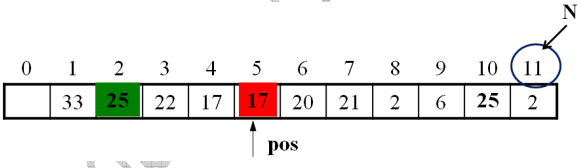
Down(*Array*, *pos*, *N*)
while 2**pos* ≤ *N* **do**
p ← 2**pos*
if *p* < N **and** *Array*[p] < *Array*[p+1] **then**
p ← *p* + 1
if *Array*[pos] < *Array*[p] **then**
Array[p]↔ *Array*[pos]
pos ← *p*
else
BREAK



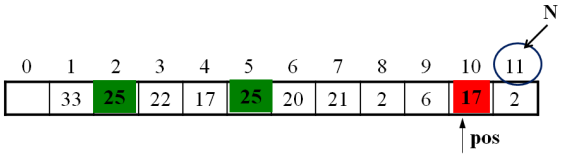
а)



б)

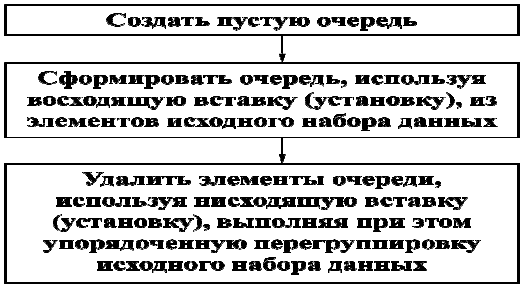


в)



г)

Пример реализации базовой пирамидальной сортировки по возрастанию с использованием очереди по приоритету.



1. Описание класса "Очередь"

```
class QuePri {
    int Array[];
    int size;
    QuePri(int x) {
        Array = new int [x+1];
    }
     // описание метода  Up (Array, pos)
     // описание метода  Down(Array, pos, size)
```

► метод добавления элемента

Insert(*Array*, *data*)

size ← size + 1
Array[size] ← *data*
call Up(*Array*, *size*)

► метод удаления элемента

Delete(*Array*, *size*)

Array[size] ↔ *Array*[1]
call Down(*Array*, 1, *size*-1)
temp ← *Array*[size]
Array[size] ← 0
size ← *size* - 1
return *temp*

```
}
```

2. Сортировка

Sort_Heap_Que(*Array*)

► создание пустой "Очереди по приоритету" **QuePri**

for *k* ← 0 **to** *length*(Array) **do** ►формирование "Очереди "
call Insert(*Array*(Queue), *Array*[k])

for *k* ← *length*(Array)-1 **to** -1 **by** -1 **do** ►удаление из "Очереди"
Array[k] ← **call** Delete(*Array*(Queue), k)

Пример выполнения по шагам формирования очереди по приоритету:

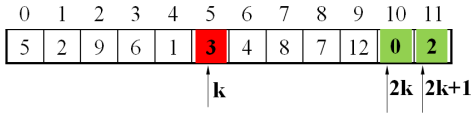
```
C:\Program Files\Java\jdk1.5.0\bin>java Sort_11
Enter number element -> 10
Enter value element 1 -> -3
Enter value element 2 -> 9
Enter value element 3 -> -1
Enter value element 4 -> 5
Enter value element 5 -> 12
Enter value element 6 -> 8
Enter value element 7 -> 2
Enter value element 8 -> 9
Enter value element 9 -> 6
Enter value element 10 -> -7

Massiv do sort :
-3      9       -1      5       12      8       2       9       6       -7

-3
9       -3
9       -3      -1
9       5       -1      -3
12      9       -1      -3      5
12      9       8       -3      5       -1
12      9       8       -3      5       -1      2
12      9       8       9       5       -1      2       -3
12      9       8       9       5       -1      2       -3      6
12      9       8       9       5       -1      2       -3      6       -7

Massiv after sort :
-7      -3      -1      2       5       6       8       9       9       12

C:\Program Files\Java\jdk1.5.0\bin>_
```

Алгоритм пирамидальной сортировки

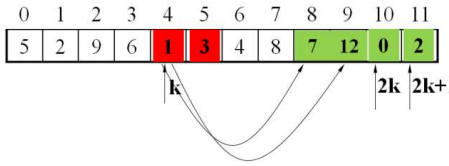
Суть – реализация операций вставки и удаления основана только на алгоритме нисходящей установки:

- сортировка выполняется без создания дополнительной структуры – «Очереди по приоритету»;
- построение сортирующего дерева выполняется прохождением набора данных в обратном порядке.

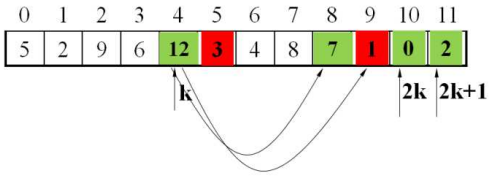
Изначально просмотр сортируемого набора данных начинается с половины и далее просматривается, двигаясь у началу. На каждой итерации текущий элемент считается корнем сортирующего дерева, которое расположено справа от него.



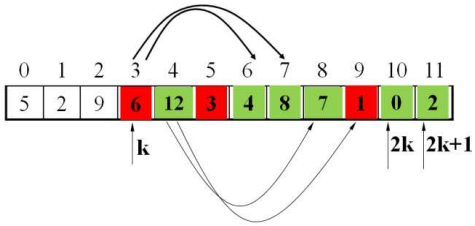
а)



б)



в)



г)

Sort_Heap(*Array*)

► построение сортирующего дерева

size ← *length*(Array)-1

for *k* ← *size*/2 **to** -1 **by** -1 **do**

Down(*Array*, *k*, *size*)

► сортировка

while *size* > 0 **do**

Array[0] ↔ *Array*[size]

size ← *size* – 1

Down(*Array*, 0, *size*)

ДЕРЕВО БИНАРНОГО ПОИСКА (BINARY SEARCH TREE, BST)

BST-дерево – это разновидность бинарного дерева, для которого верно:

- ключ каждого узла дерева **X** (если **X** не равен *null*) имеет левые узлы со значениями ключей меньшими значения ключа узла **X**;
- все правые узлы в правом поддереве узла **X** содержат значения ключей, которые больше или равны значению ключа узла **X**.

BST-дерево применяется для построения абстрактных структур, таких, как *множества*, *мультимножества*, *ассоциативные массивы (словари)*.

В реализациях, основанных на *хэш-таблицах*, среднее время поиска лучше, чем в реализациях, основанных на BST-деревьях (однако не гарантируется высокая скорость выполнения отдельной операции; например, операция вставки выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хэш-таблицы).

Кроме этого, на *хэш-таблицах* нельзя реализовать быстро работающие дополнительные операции *MIN*, *MAX* и алгоритм обхода всех хранимых пар в порядке возрастания или убывания ключей.

Поэтому использование BST-деревьев позволяет разрабатывать алгоритмы с высокой средней производительностью операций найти, вставить, выбрать и сортировать.

Стандартная реализация операции вставки нового узла в дерево → новый элемент становится внешним узлом (листом) по правилу: если ключ нового узла меньше ключа в корне, то элемент вставляется в левое поддерево, иначе элемент вставляется в правое поддерево.

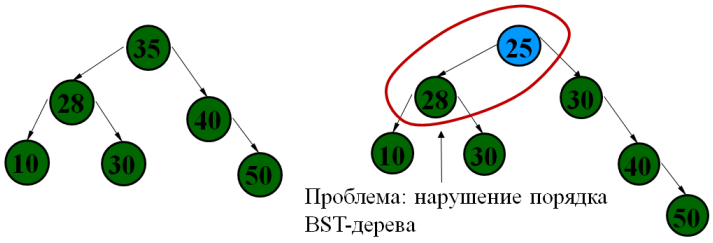
BST-дерево – это упорядоченная (отсортированная) структура данных, поскольку, используя последовательный обход, можно получить не убывающую последовательность узлов дерева.

Однако очень часто *BST-дерево* формируется на основе упорядоченных файлов или данных, что приводит к вырождению дерева в список.

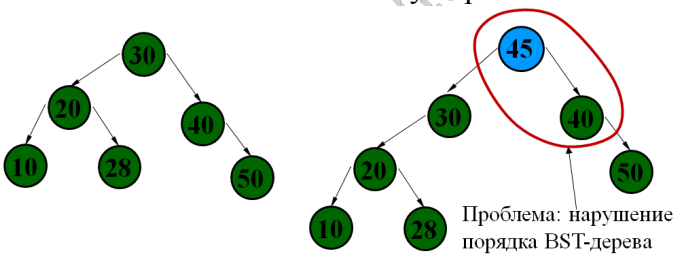
Существует другой метод вставки, при котором новый элемент становится корнем, т.е. все последние вставленные узлы находятся вблизи вершины.

Пример вставки в корень дерева:

1) Новый узел имеет значение меньше чем у корня



2) Новый узел имеет значение больше чем у корня

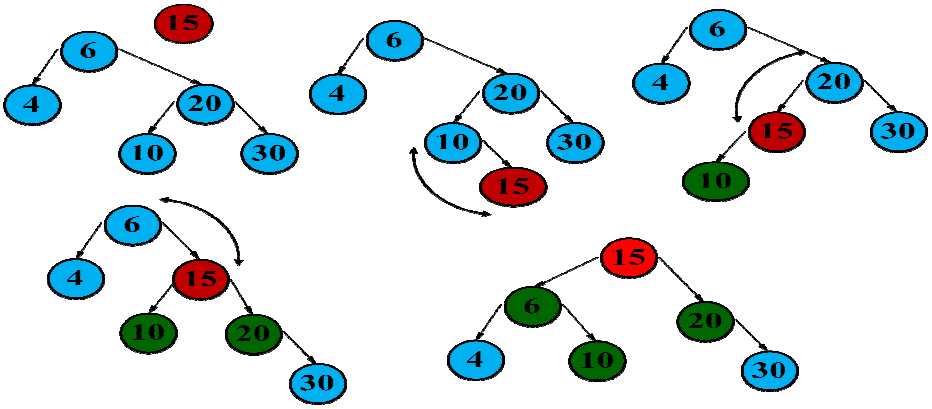


Для восстановления требуемого порядка применяется *ротация* – преобразование дерева, основанное на обмене узла с его дочерним узлом для сохранении порядка следования ключей в узлах BST-дерева.

а) Ротация вправо – местами меняются родитель и его левый узел (потомок):
- родитель становится правым узлом своего левого потомка;
- правый узел левого потомка становится левым узлом родителя.

б) Ротация влево – местами меняются родитель и его правый узел (потомок):
- родитель становится левым узлом своего правого потомка;
- левый узел правого потомка становится правым узлом родителя.

Чтобы обеспечить вставку именно в корень дерева:
- сначала новый узел делается листом дерева, рекурсивно спускаясь по дереву вниз;
- затем, рекурсивно поднимаясь, выполняются соответствующие ротации.



► Метод добавления нового элемента в BST-дерево

```
Insert_BST(Root, New_Elem)
    Root ← call Insert_Node(Root, New_Elem)
```

► Рекурсивный метод добавления нового элемента в корень BST-дерева

```
Insert_Node(Current, New_Elem)
    if Current = NULL then
        return New_Elem
    if data(New_Elem) < data(Current) then
        left(Current) ← call Insert_Node(left(Current), New_Elem)
        Current ← call Rotation_R(Current)
    else
        right(Current) ← call Insert_Node(right(Current), New_Elem)
        Current ← call Rotation_L(Current)
    return Current
```

► Метод – ротация вправо

```
Rotation_R(Current)
    Temp ← left(Current)
    left(Current) ← right(Temp)
    right(Temp) ← Current
    Current ← Temp
    return Current
```

► Метод – ротация влево

```
Rotation_L(Current)
    Temp ← right(Current)
    right(Current) ← left(Temp)
    left(Temp) ← Current
    Current ← Temp
    return Current
```