АЛГОРИТМЫ ПОИСКА ПОДСТРОКИ В СТРОКЕ

- **СТРОКА** в которой осуществляется поиск называется **ТЕКСТОМ**
- **СТРОКА**, которую необходимо найти называется **ПОДСТРОКОЙ ПОИСКА** (**ОБРАЗОМ**)

ПРЯМОЙ ПОИСК

- самый простой и не эффективный поиск.
- не проводит анализа подстроки
- наиболее эффективно работает при небольшом количестве совпадений при очередном сравнивании символов образа с символами текста

ПРЯМОЙ ПОИСК

п	р	и	м	е	р		т	е	к	с	т	а		д	л	я		п	о	и	с	к	а
т	е	к	с	т																			

п	р	и	м	е	р		т	е	к	с	т	а		д	л	я		п	о	и	с	к	а
	т	е	к	с	т																		

п	р	и	м	е	р		т	е	к	с	т	а		д	л	я		п	о	и	с	к	а
		т	е	к	с	т																	

ПРЯМОЙ ПОИСК

п	р	и	м	е	р		т	е	к	с	т	а		д	л	я		п	о	и	с	к	а
			т	е	к	с	т																	
				т	е	к	с	т																
					т	е	к	с	т															
						т	е	к	с	т														
							т	е	к	с	т													

ПРЯМОЙ ПОИСК

S – текст, n – количество символов текста

O – образ, m – количество символов образа

```
i =0; j=0;
пока (i<n) нц
    j=0; f=0;k=i;
    пока (j<m и k<n и S[k]==O[j]) нц
        k++; j++;f++;
    кц
```

ПРЯМОЙ ПОИСК

```
    если f==m то УСПЕХ, вернуть i;
    i++;
кц
```

ПРЯМОЙ ПОИСК

Самый неэффективный случай:

а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	а	в
а	а	а	а	а	в																	

Оценка времени работы алгоритма O(n*m)

Алгоритм Кнута, Морриса и Пратта

- Описан Кнутом и Праттом и независимо от них Моррисом
- Результаты опубликованы совместно в 1977 г.
- Алгоритм назвали КМП-алгоритмом
- Временная характеристика алгоритма O(n)

Алгоритм Кнута, Морриса и Пратта

- для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге алгоритма был максимально возможным

а	**в**	**с**	**д**					
а	**в**	**с**	**е**					
			а	**в**	**с**	**е**		

  - для вычисления этого сдвига алгоритм разбит на две части:

КМП - АЛГОРИТМ

1) предварительная обработка подстроки

2) поиск подстроки

а	в	с	д					
а	в	с	е					
			а	в	с	е		

- обозначим j - количество совпадений, текущего сравнения (j = 3)
  - значение j зависит только от вида образа, никак не зависит от текста

КМП – АЛГОРИТМ. Предварительная обработка образа

- Для образа строится таблица сдвигов d[m]
- d[0] = -1, d[1] = 0
- d[j] – длина самой длинной последовательности символов образа, предшествующих j, которая полностью совпадает с началом образа
- При j совпадениях образа с текстом можно выполнить сдвиг на **j – d[j]**

КМП – АЛГОРИТМ. Предварительная обработка образа

Примеры таблиц сдвигов для различных образов:

АБВГ	АБАБВ	АБАБВА
d[0]=-1	d[0]=-1	d[0]=-1
d[1]=0	d[1]=0	d[1]=0
d[2]=0	d[2]=0	d[2]=0
d[3]=0	d[3]=1	d[3]=1
	d[4]=2	d[4]=2
		d[5]=2

КМП – АЛГОРИТМ. Предварительная обработка образа

```
m = длина образа o;
j = 1, k=0
d[0] = -1;
d[1] = 0;
Пока j<m-1  нц
    Пока o[k]!=o[j] и j<(m-1) нц
        d[j+1] = d[j]; j++;  кц
```

КМП – АЛГОРИТМ. Предварительная обработка образа

```
Если j==m-1 то закончить выполнение
        цикла
k++;
d[j+1]=k;
j++;
кц
```

КМП – АЛГОРИТМ. Пример

a	b	c	a	b	d

d	-1	0	0	0	1	2

сдвиг на j – d[j]

a	b	c	a	b	c	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d		
a	b	c	a	b	d	5 совпадений, сдвиг 5 – 2 = 3																	
			a	b	c	a	b	d	4 совпадения, сдвиг 4 – 1 = 3													
						a	b	c	a	b	d	1 совпадение, сдвиг 1 – 0 = 1										
							a	b	c	a	b	d	5 совпадений, сдвиг 3									
										a	b	c	a	b	d	2 совпадения, сдвиг 2						

КМП – АЛГОРИТМ. Пример

a	b	c	a	b	c	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d		
												a	b	c	a	b	d	сдвиг 3					
															a	b	c	a	b	d	сдвиг 1		
																a	b	c	a	b	d	!!!	

Чем больше совпадений по тексту и меньше совпадений по строке

d	-1	0	0	0	1	2

Алгоритм Боуэра-Мура

Предложен в 1977

a	b	c	a	b	d

a	b	c	a	b	f	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d		
a	b	c	a	b	d																		
						a	b	c	a	b	d												
							a	b	c	a	b	d											

Алгоритм Боуэра-Мура

- Сравнение символов образа с символами текста осуществляется с конца образа
- Если несовпадение произошло на символе, не встречающемся в образе выполняется сдвиг на длину образа минус количество совпадений

Алгоритм Боуэра-Мура

- Если несовпадение произошло на символе, встречающемся в образе, то выполняется сдвиг на величину, равную расстоянию от совпавшего символа до конца образа минус количество совпадений
- Если получен отрицательный или нулевой сдвиг, осуществляется сдвиг на 1

Алгоритм Боуэра-Мура

- Перед работой создается массив d, размерность которого равна размеру использованного алфавита

Алгоритм Боуэра-Мура. Пример

a	b	c	a	b	d

d	2	1	3	0	6	6
	a	b	c	d	e	…

a	b	c	a	b	c	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d		
a	b	c	a	b	d																		
			a	b	c	a	b	d															
				a	b	c	a	b	d														
							a	b	c	a	b	d											

Алгоритм Боуэра-Мура. Пример

a	b	c	a	b	c	a	a	b	c	a	b	a	b	c	b	a	b	c	a	b	d		
									a	b	c	a	b	d									
												a	b	c	a	b	d						
													a	b	c	a	b	d					
																a	b	c	a	b	d		

Чем меньше совпадений, тем быстрее

ПОИСК ПОДСТРОКИ В СТРОКЕ

- Определить таблицы сдвигов для КМП-алгоритма и алгоритма Боуэра – Мура
- На каждом шаге выбирать лучший сдвиг

ПОИСК В МАССИВЕ

- Неупорядоченный массив – прямой поиск (O(n))
- Упорядоченный массив – бинарный поиск ($O(\log_2 n)$)

Бинарный поиск

Вход – X[n], a

1. F = 0, L = n-1
2. Если F>L то вернуть -1
3. M =1/2* (F+L)
4. Если X[M] = a, то вернуть M
   иначе если X[M]>a то L = M-1
   иначе F = M+1
5. Перейти на шаг 2

Интерполяционный поиск

M =1/2* (F+L) ☐ F + 1/2*(L-F)

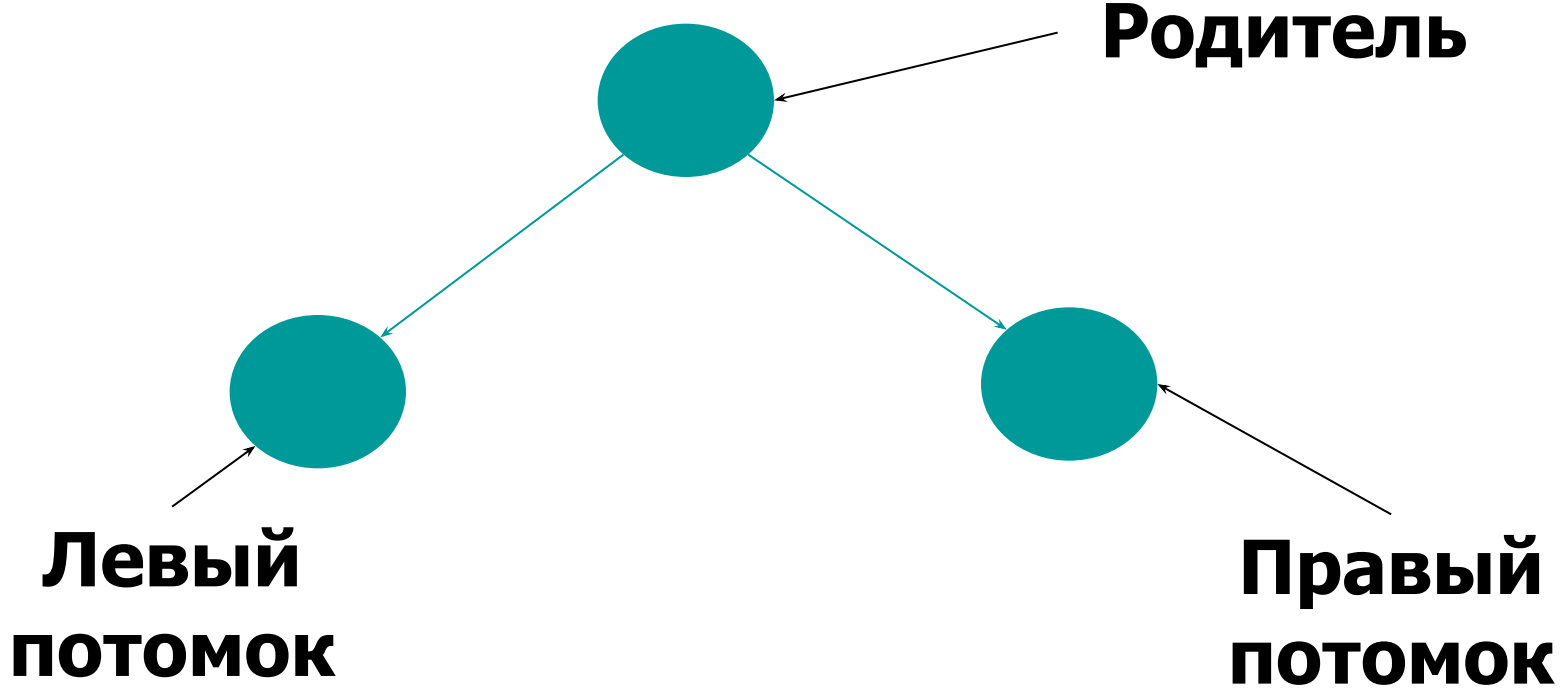
Если искомый элемент S:

1/2 ☐ (S-X[F])/(X[L] – X[F])

Используется алгоритм бинарного поиска

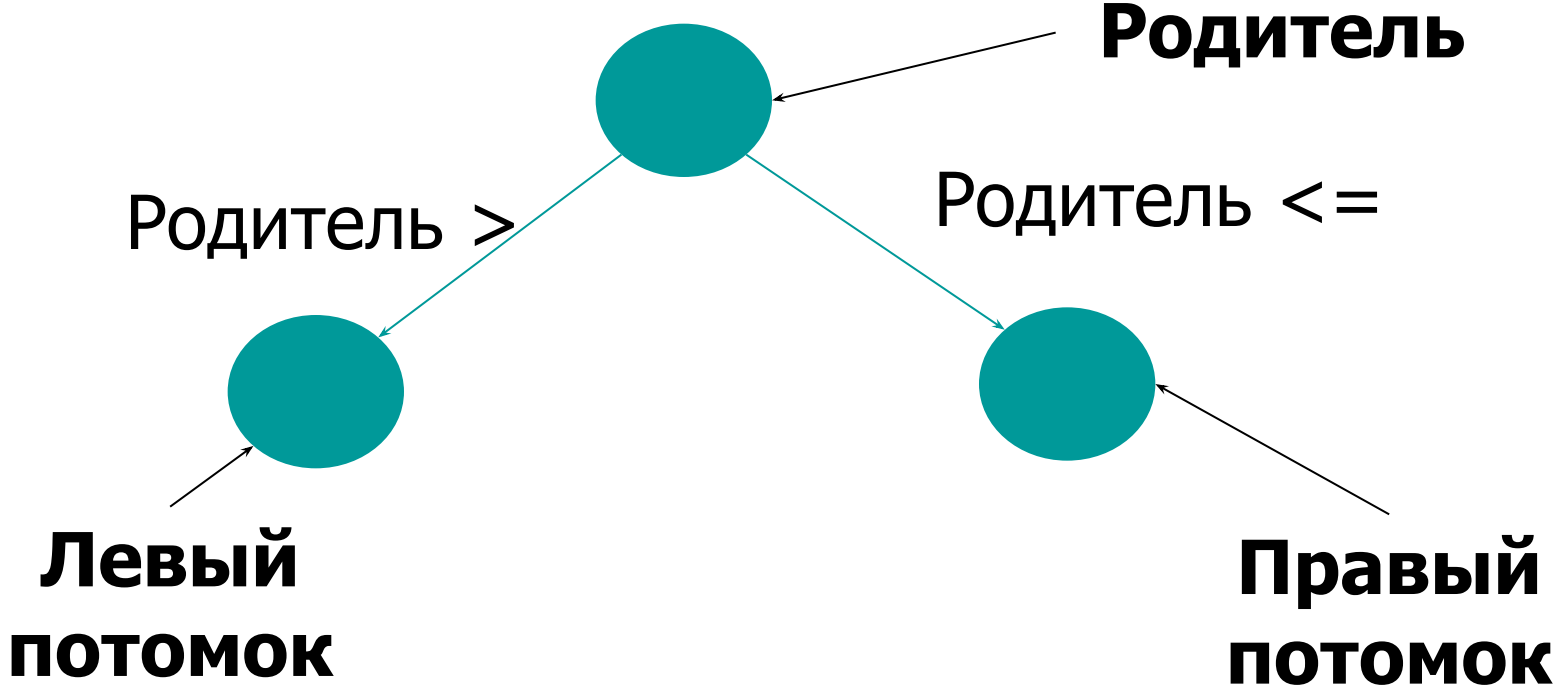
BST - деревья

Binary **S**earch **T**ree - деревья двоичного поиска.



BST - деревья

Правила организации элементов:



BST - деревья

Реализация описания узла дерева в Си

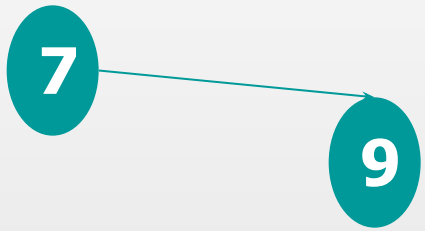
```
typedef struct node{
        int data;
        struct node *left;
        struct node *right;
        } Node;
```

BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16

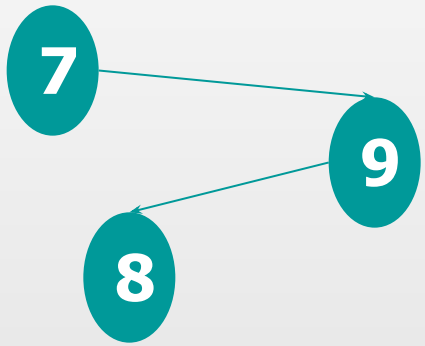
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



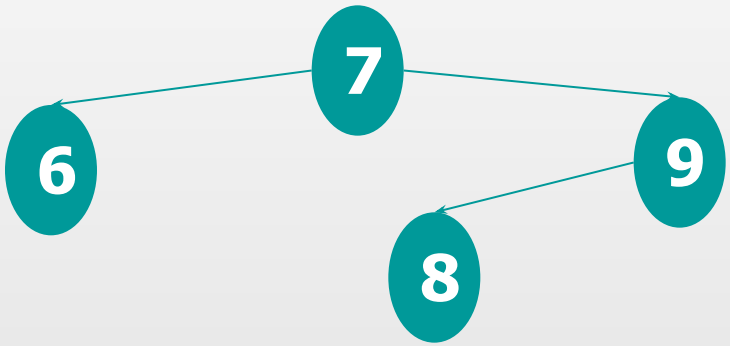
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



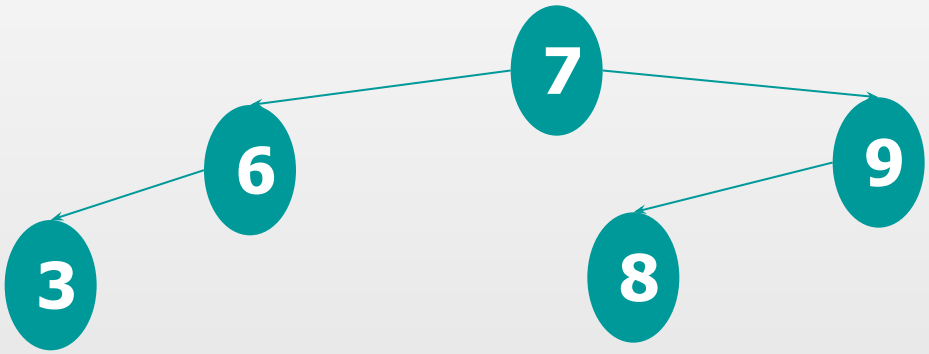
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



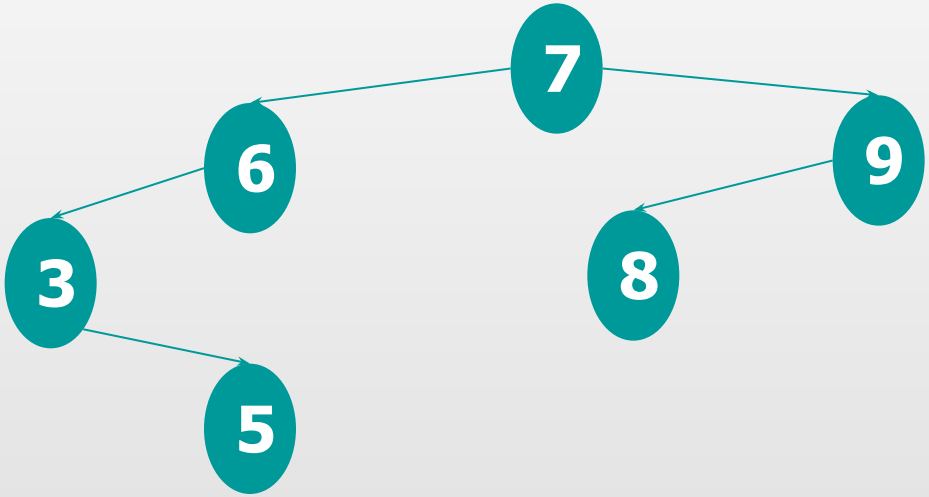
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



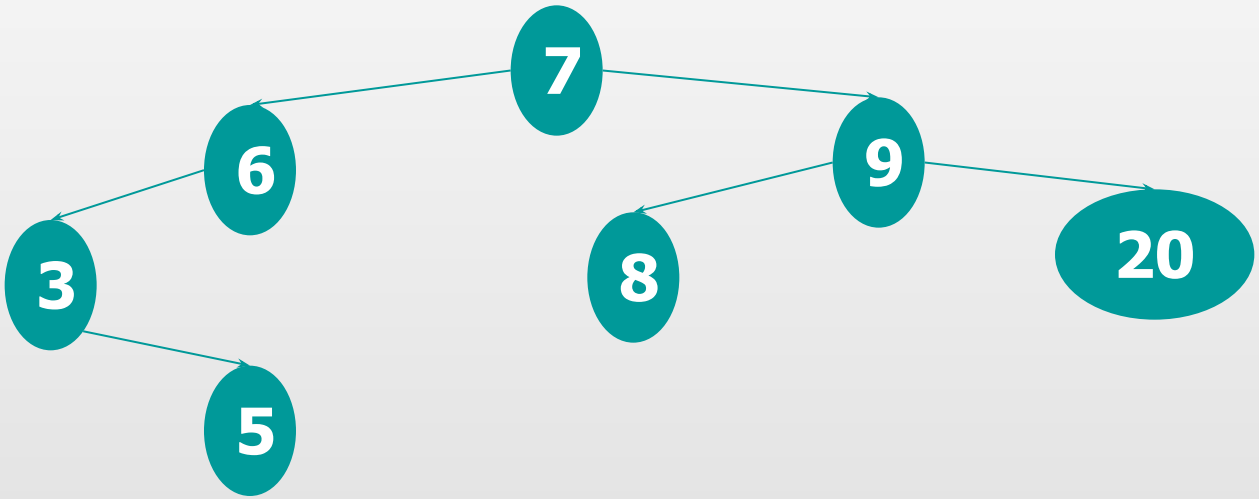
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



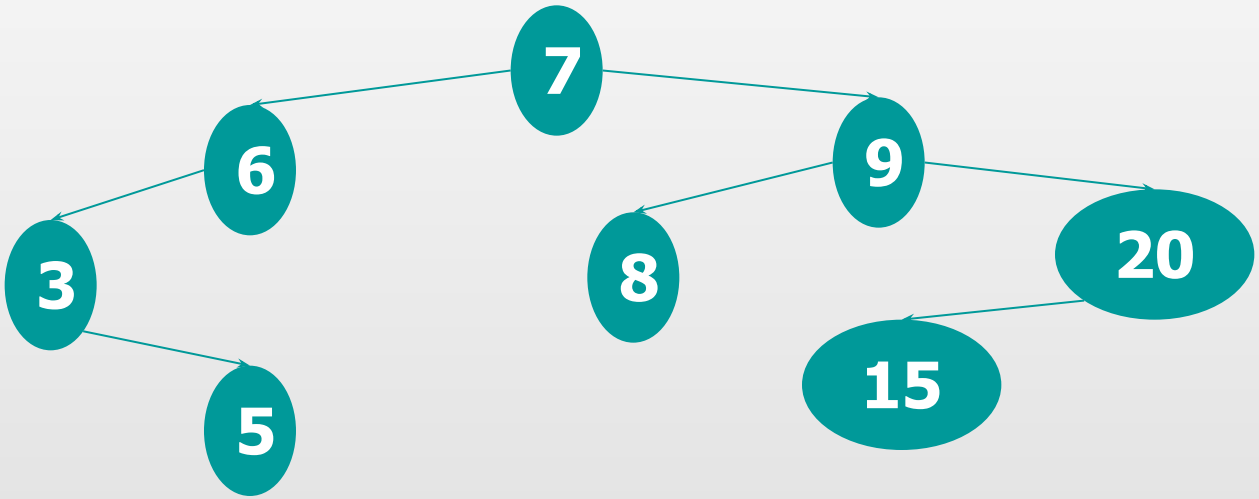
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



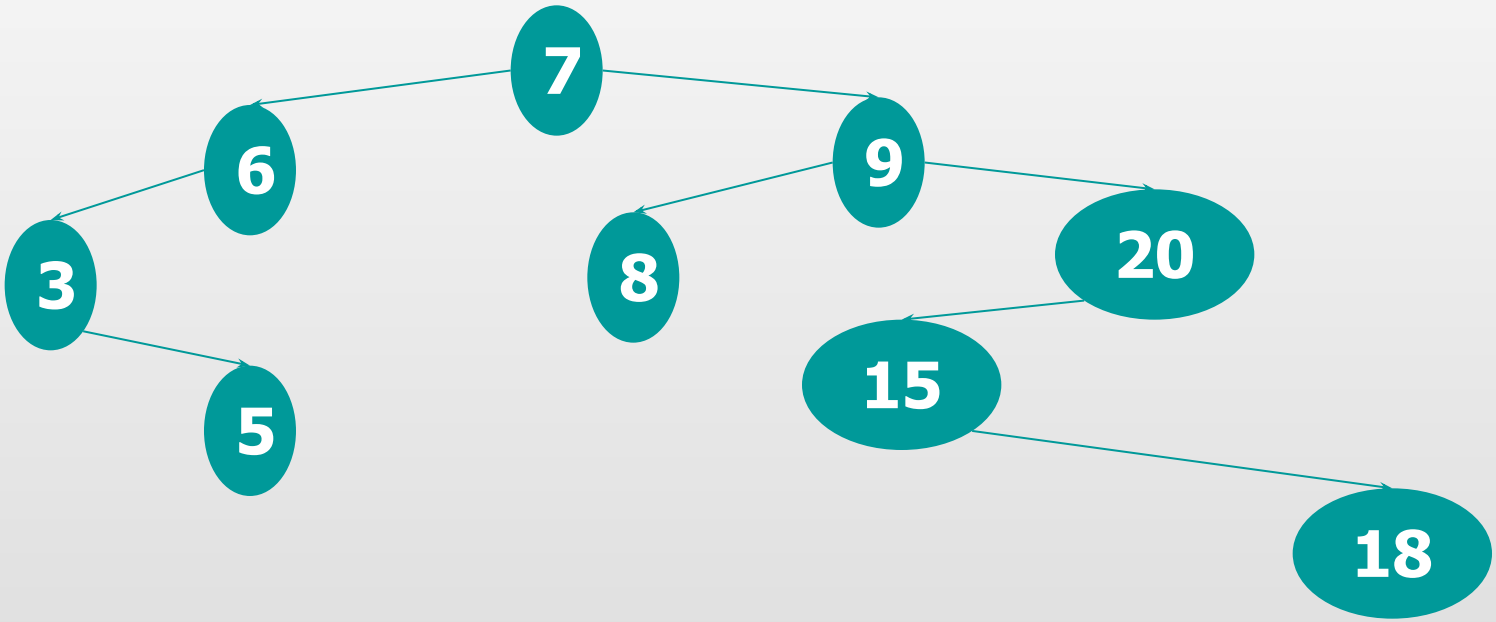
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



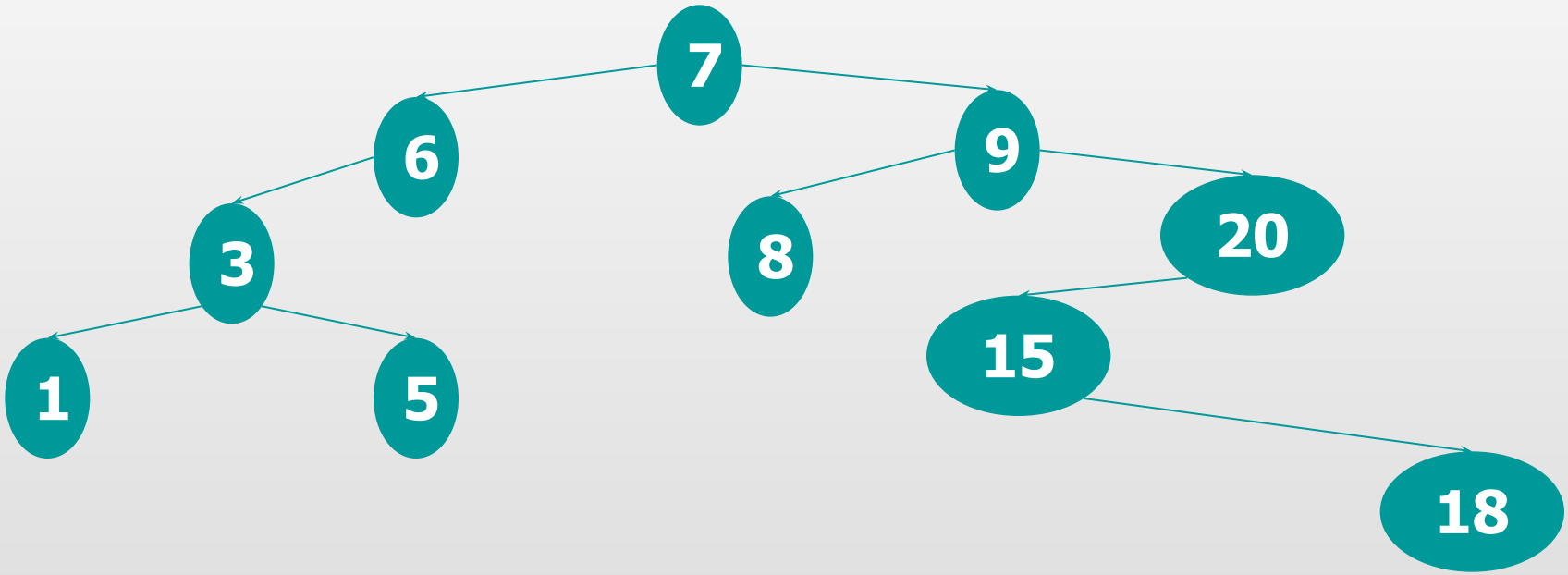
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



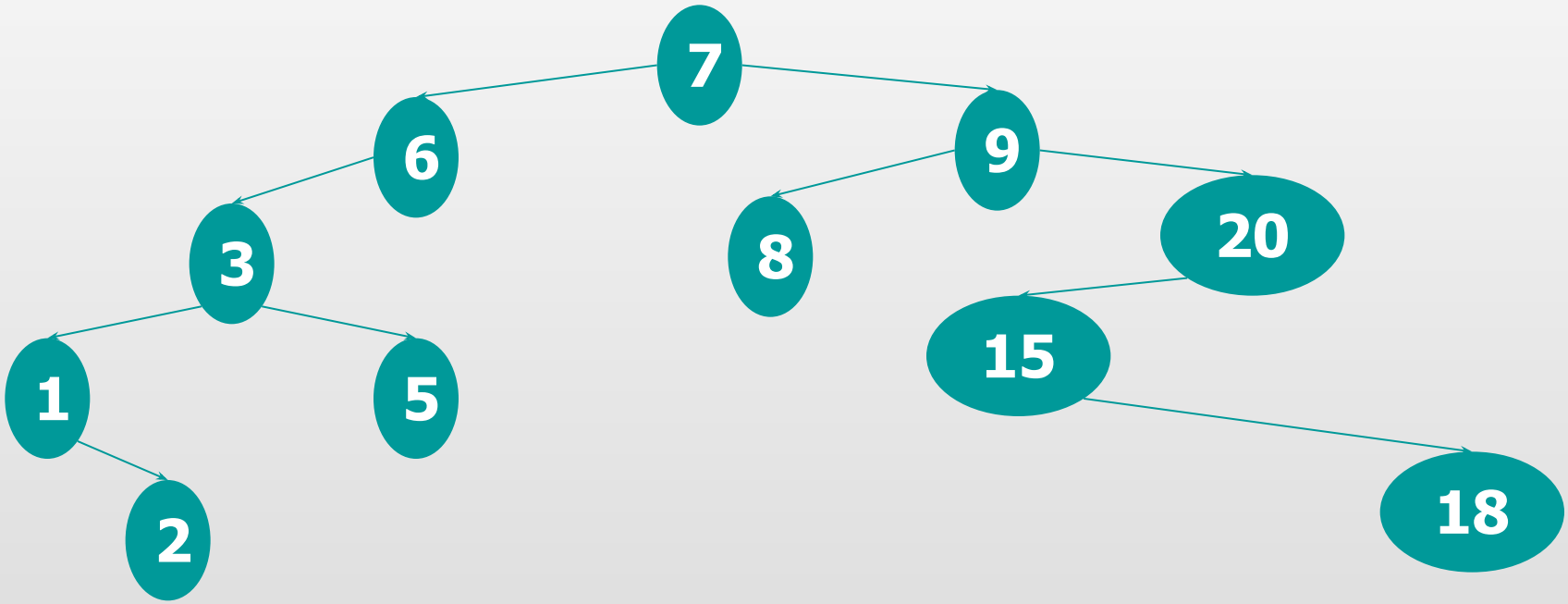
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



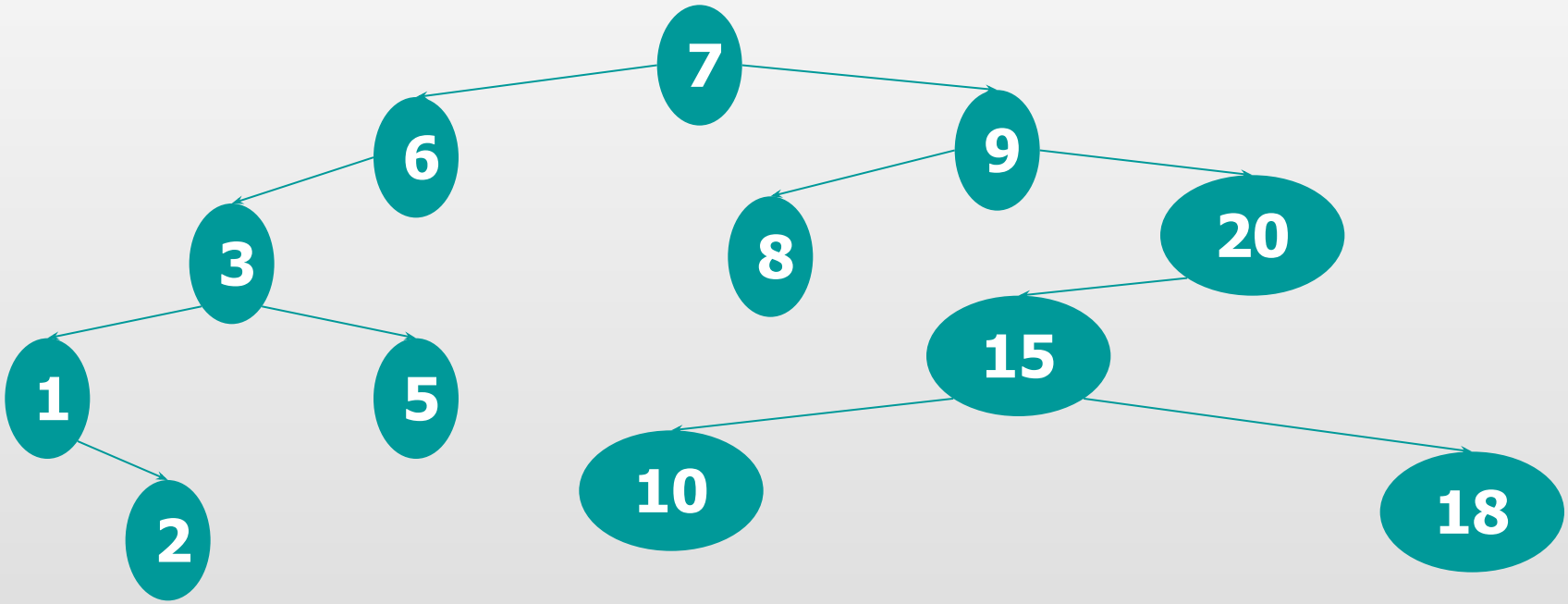
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



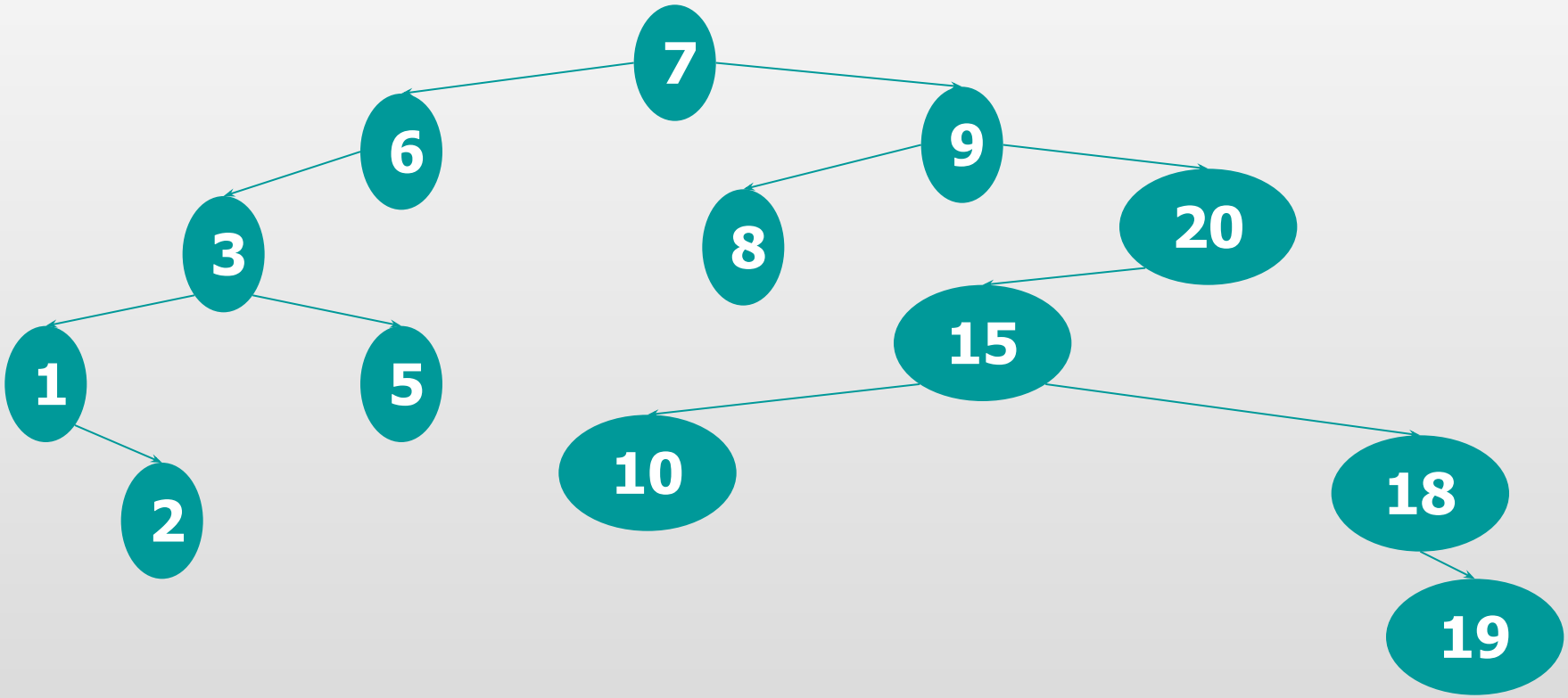
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



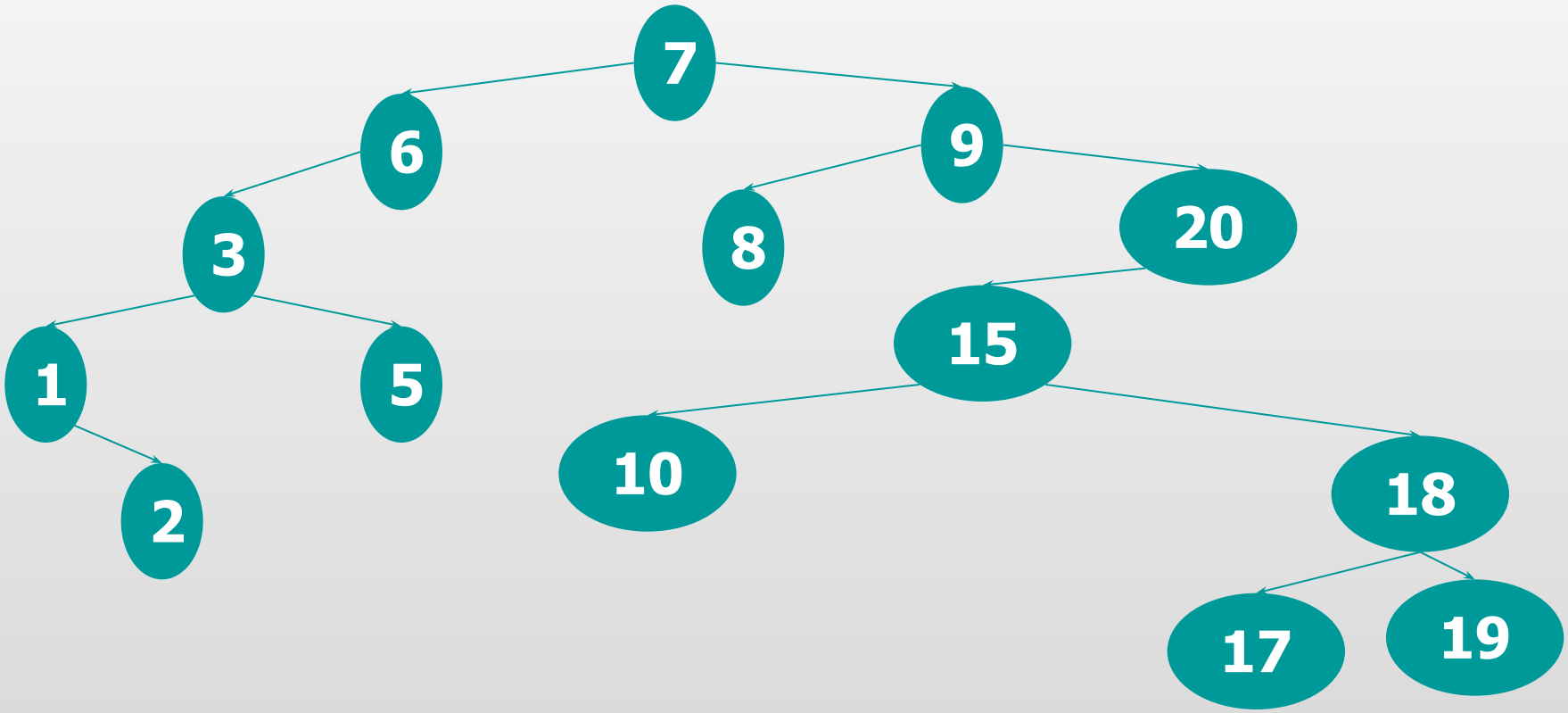
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



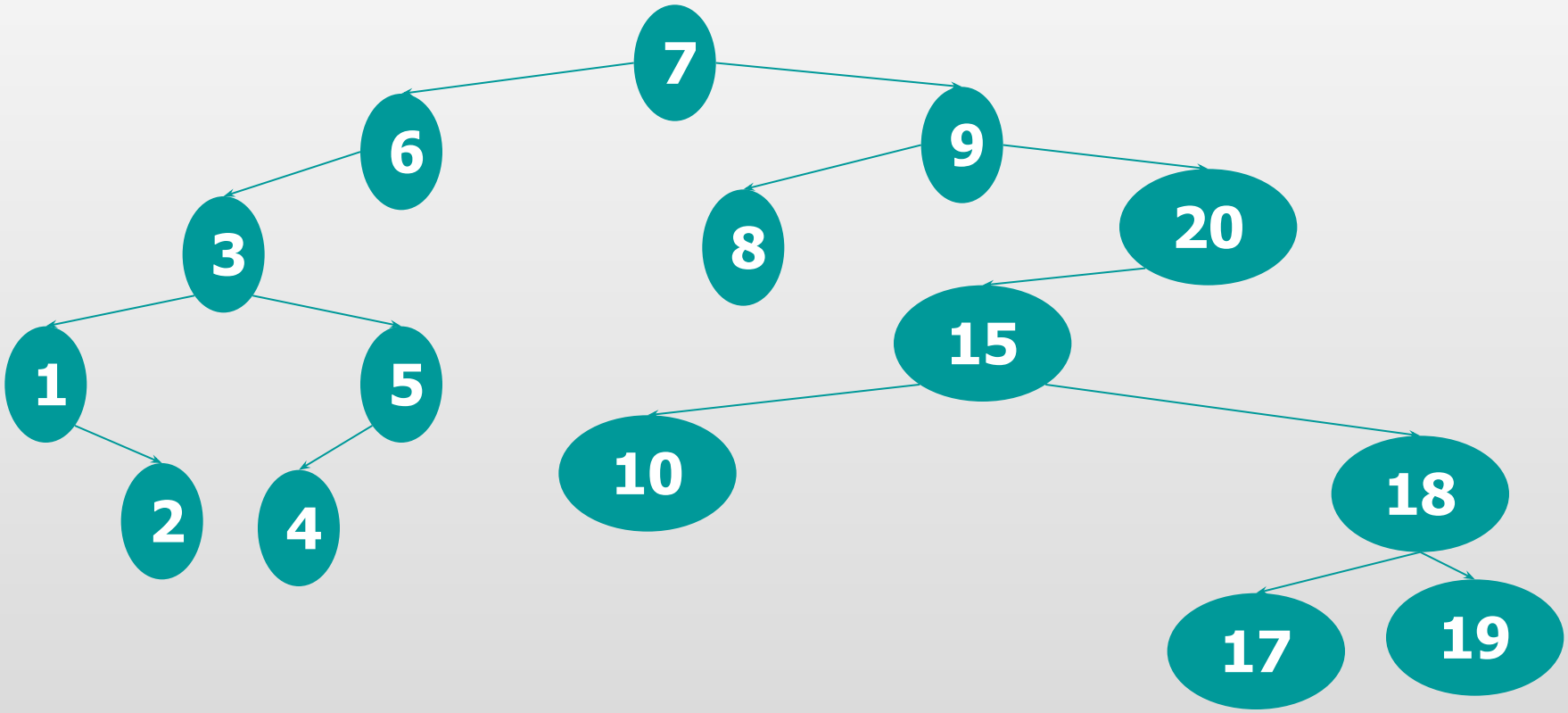
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



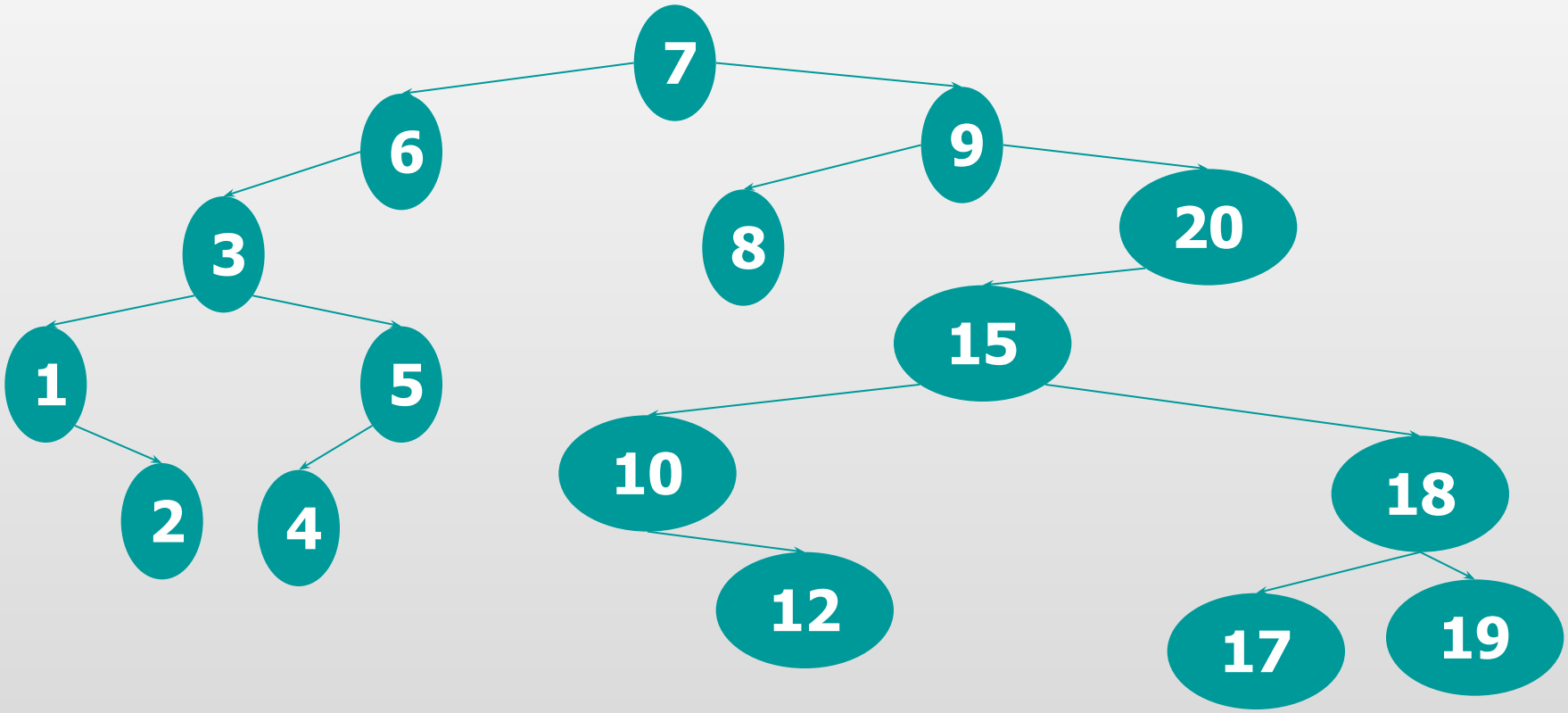
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



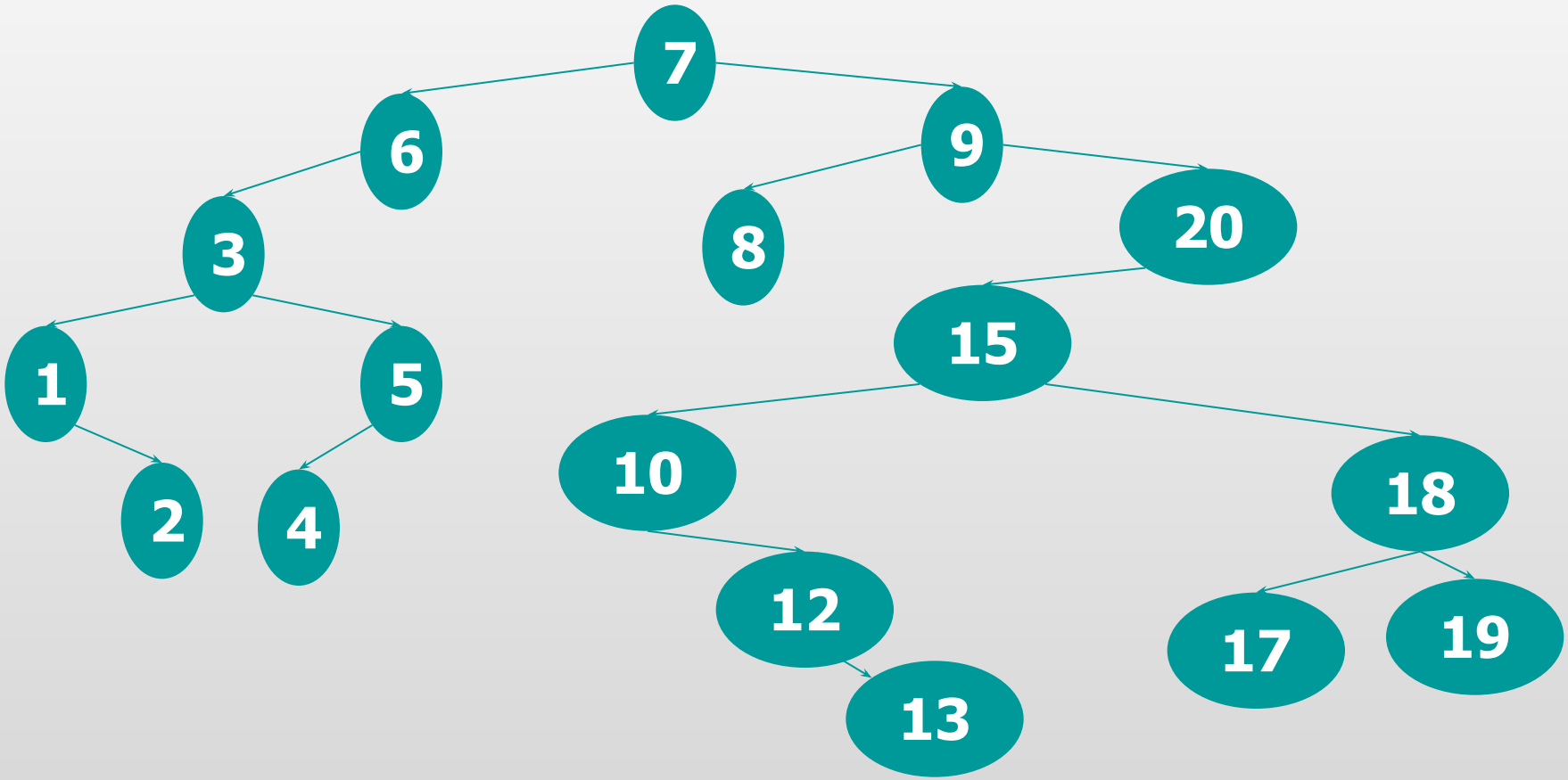
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



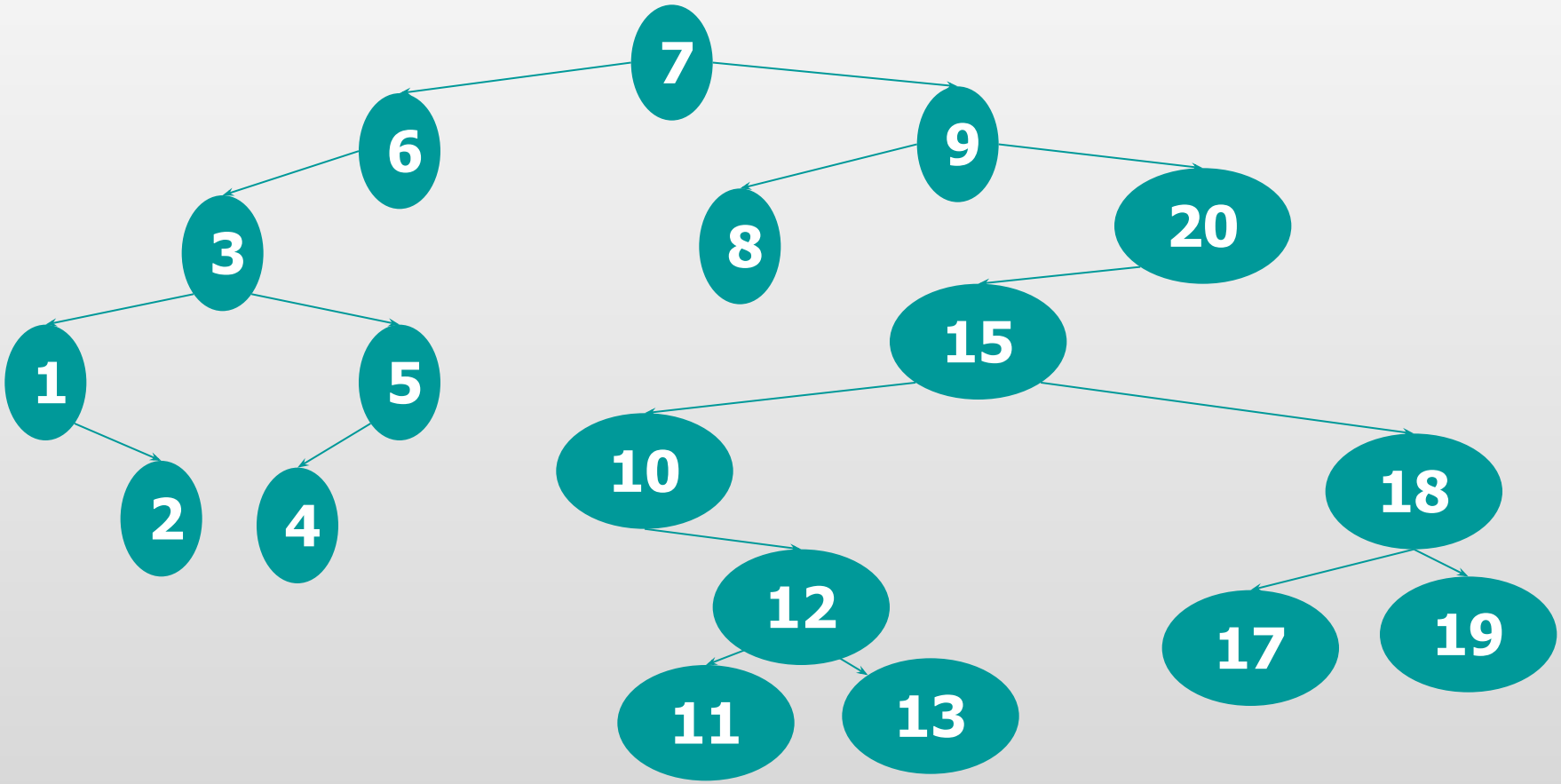
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



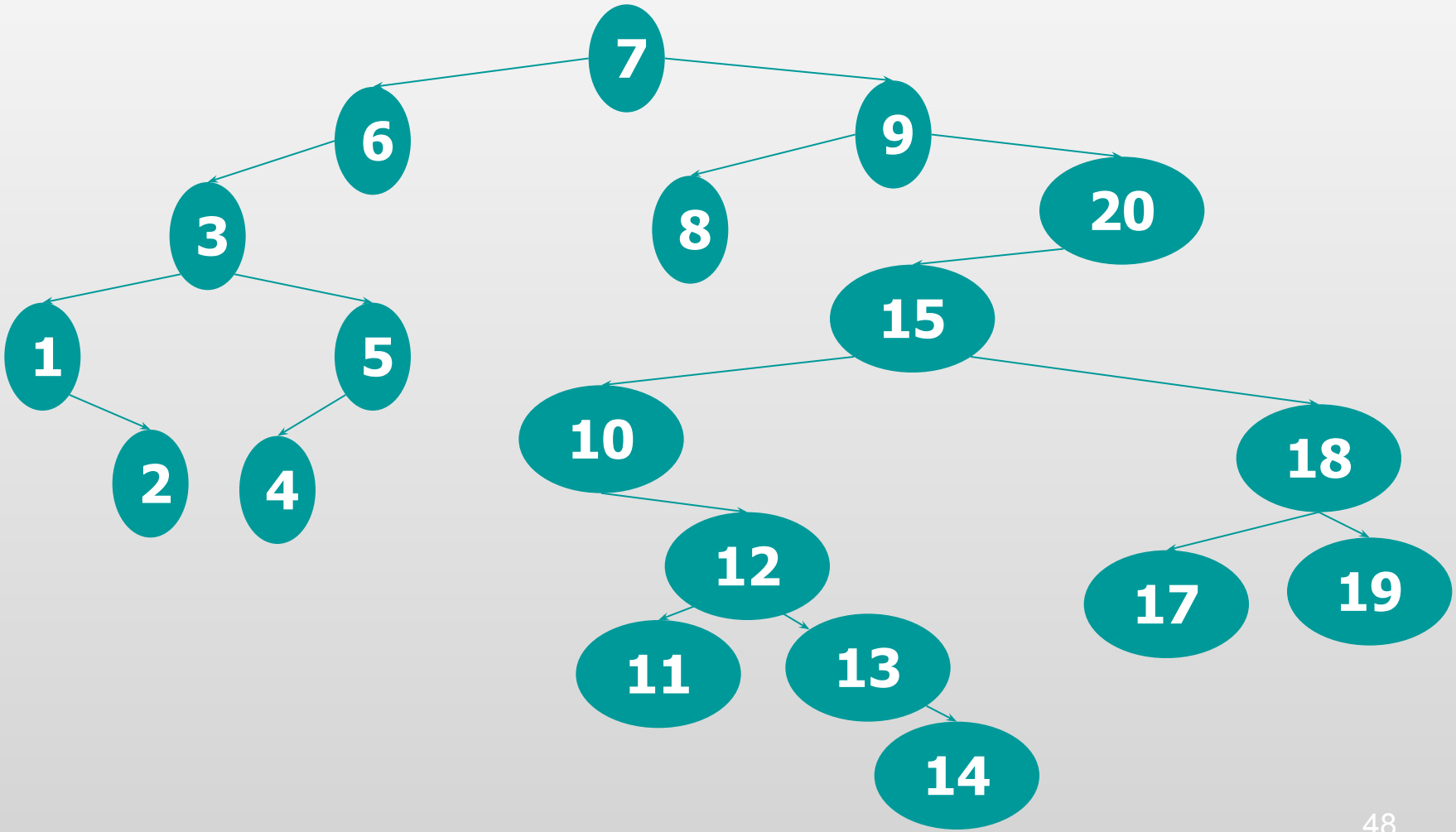
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



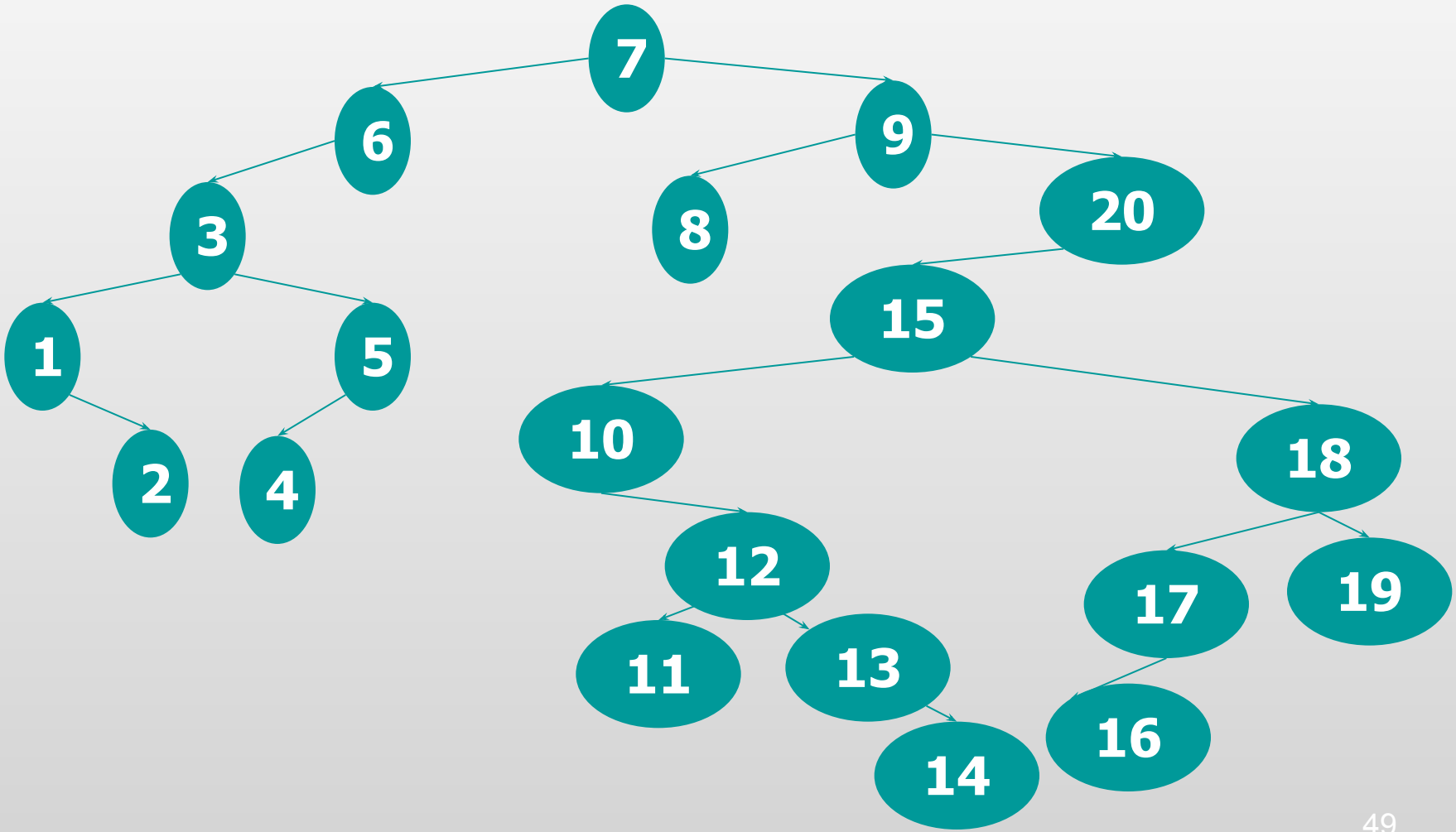
BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16



BST - деревья

7	9	8	6	3	5	20	15	18	1	2	10	19	17	4	12	13	11	14	16

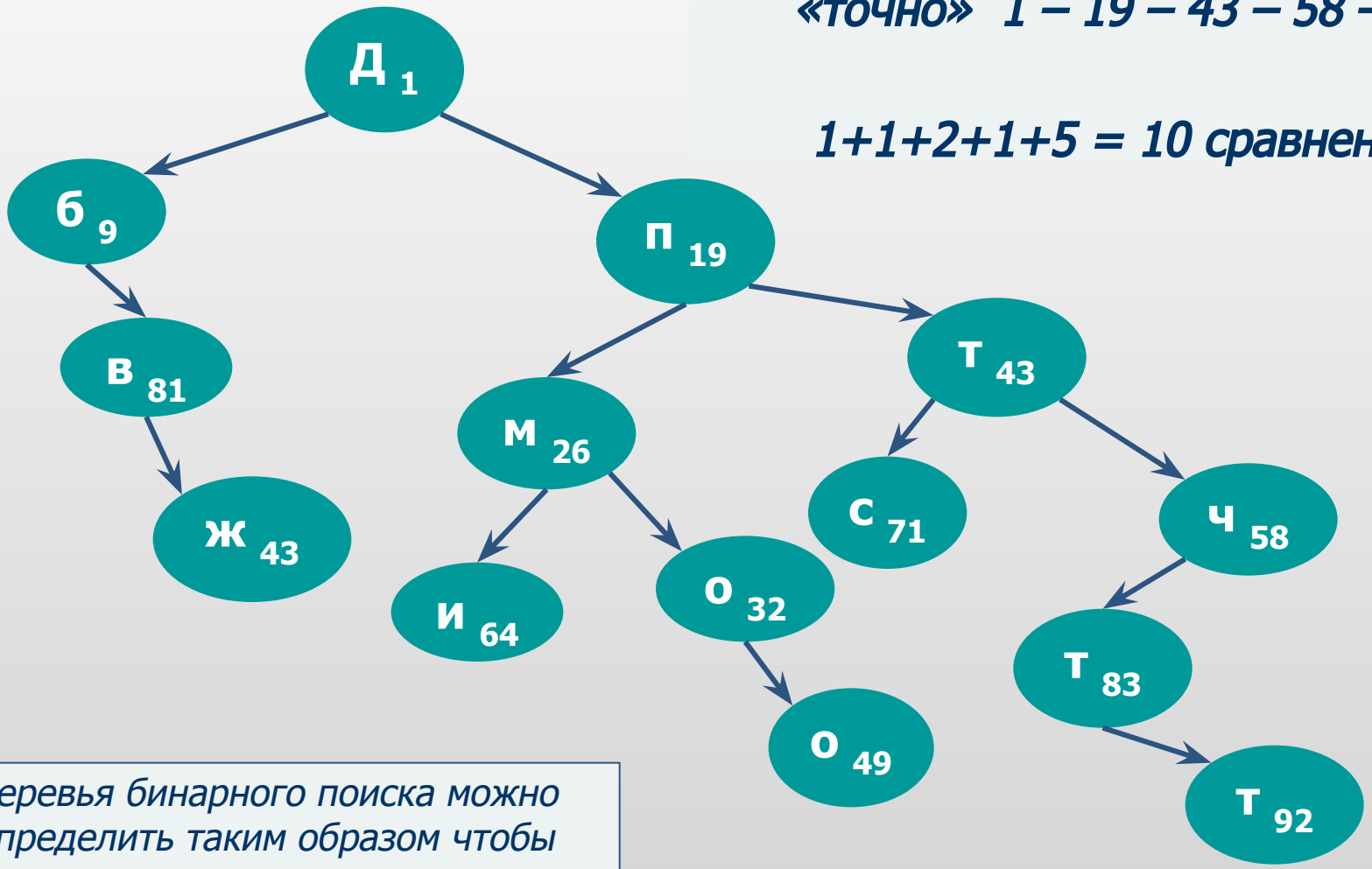


Пример использования BST-деревьев для поиска слов

Деревья бинарного поиска можно определить таким образом чтобы индексы строились в точности так же…

Деревья – 1	бинарного – 9	поиска – 19	можно – 26
определить – 32	таким – 43	образом – 49	чтобы – 58
индексы – 64	строились – 71	в – 81	точности – 83
так – 92	же – 96		

Пример использования BST-деревьев для поиска слов



«точно» 1 – 19 – 43 – 58 – 83

1+1+2+1+5 = 10 сравнений

Деревья бинарного поиска можно определить таким образом чтобы индексы строились в точности так же…

BST – деревья. Алгоритм вставки элемента

Вставка (Node ** Root, int key)

Если *Root - пустой, то

выделить память
под *Root
*Root -> data = key
*Root -> left = Null
*Root->right = Null
return

BST – деревья. Алгоритм вставки элемента

иначе

Если key >= *Root->data то

Вставка (*Root->right, key)

иначе

Вставка (*Root->left, key)

конец

BST – деревья. Алгоритм вставки элемента. Не рекурсивная реализация

Вставка1 (Node ** Root, int key)

Если *Root – пустой то выделить память
под *Root
*Root -> data = key
*Root -> left = Null
*Root->right = Null
return

BST – деревья. Алгоритм вставки элемента. Не рекурсивная реализация

```
Node *temp = *Root;
Пока  (temp не пуст) нц
        Если (key>=temp->data) то
            Если (temp->right - пуст) то
                выделить память под temp->right
                temp->right ->left = NULL
                temp->right->right = NULL
              temp->right->data = key
              return
            иначе temp = temp->right;
```

BST – деревья. Алгоритм вставки элемента. Не рекурсивная реализация

```
иначе  Если temp->left – пуст то
       выделить память под temp->left
       temp->left->left = NULL;
       temp->left->right = NULL;
       temp-> left->data = key;
       return;
       иначе temp = temp->left;
кц
конец
```