Searching Algorithms

Data Structures & Algorithms

Dr. Sambit Bakshi
Dept. of CSE, NIT Rourkela

Searching

Finding the location of an given item in a collection of item

Linear Search with Array

2	**7**	**9**	**12**
1	2	3	4

Algorithm

[1] i = 1

[2] If K = A[i] , Print "Search is Successful" and Stop

[3] i = i + 1

[4] If (i <= n) then Go To Step [2]

[5] Else Print "Search is Unsuccessful" and Stop

[6] Exit

Complexity Of Linear Search Array

Case 1: Key matches with the first element

$T(n)$ = Number of Comparison

$T(n) = 1$, Best Case $= O(1)$

Case 2: Key does not exist

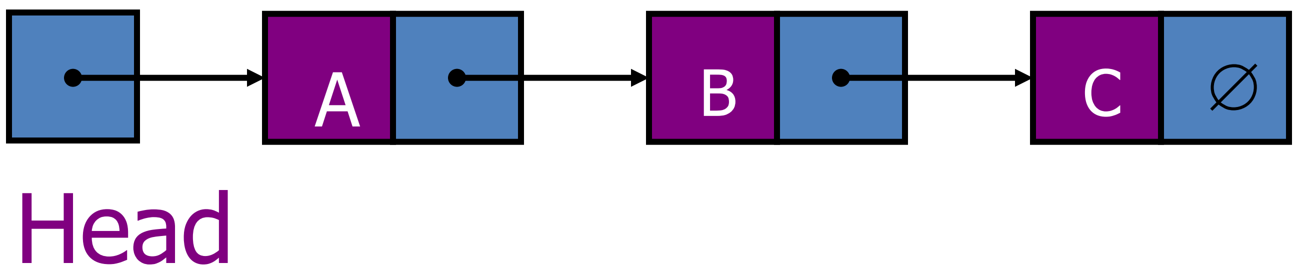
$T(n) = n$, Worst Case $= O(n)$

Case 3: Key is present at any location with same probability

$T(n) = (1+2+3+\ldots+n)/n = (n+1)/2$,

Average Case $= O(n)$

Linear Search with Linked List



Best Case
Worst Case
Average Case

Linear Search with Ordered List

5	**7**	**9**	**10**	**13**	**56**	**76**	**82**	**110**	**112**
1	2	3	4	5	6	7	8	9	10

Algorithm

[1] i = 1

[2] If K = A[i] , Print "Search is Successful" and Stop

[3] i = i + 1

[4] If (i <= n) and (A[i] <= K) then Go To Step [2]

[5] Else Print "Search is Unsuccessful" and Stop

[6] Exit

Complexity

Case 1: Key matches with the first element

$T(n)$ = Number of Comparison

$T(n) = 1$, Best Case $= O(1)$

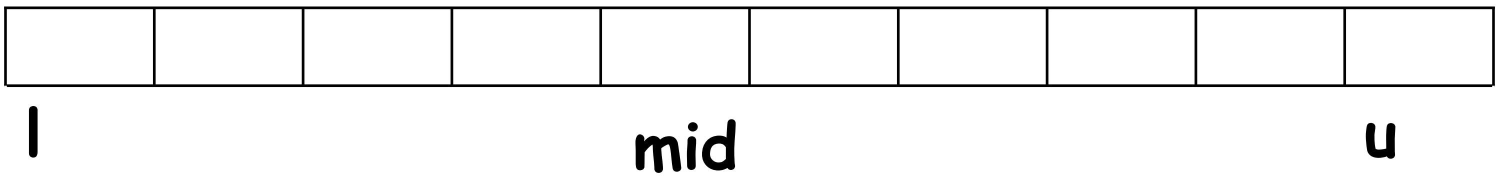
Case 2: Key does not exist

$T(n) = n$, Worst Case $= O(n)$

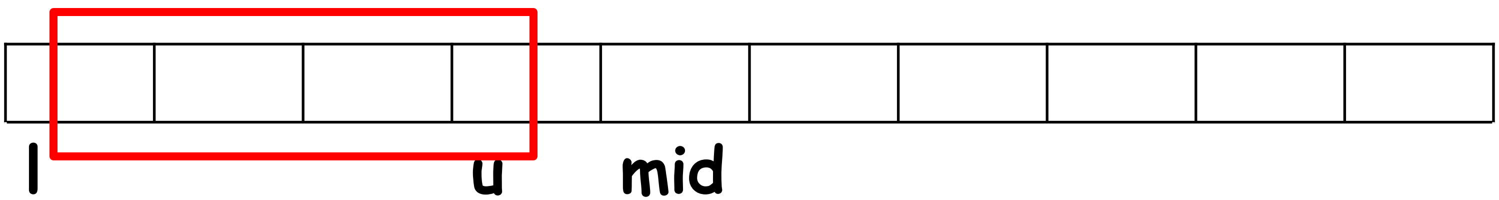
Case 3: Key is present at any location

$T(n) = (n+1)/2$, Average Case $= O(n)$

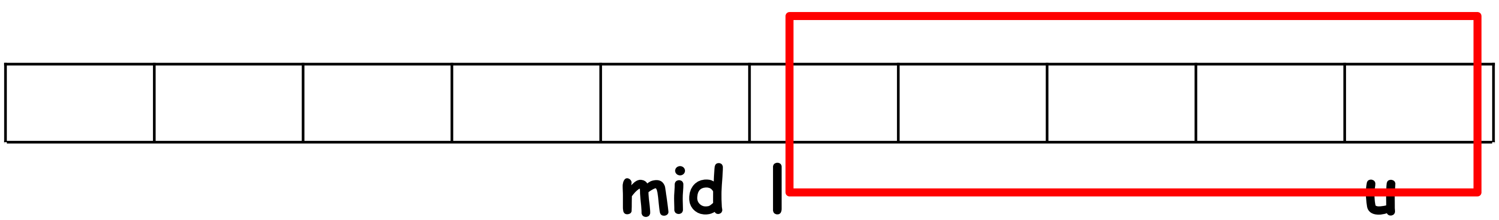
Binary Search



mid = (l + u) /2 If K = A[mid] then done



If K < A[mid] u = mid -1



If K > A[mid] l = mid +1

Algorithm

[1] l =1, u =n

[2] while (l <= u) repeat steps 3 to 7

[3] mid = (l + u) / 2

[4] if K = A[mid] then print Successful and Stop

[5] if K < A[mid] then

[6] u = mid -1

[7] else l = mid + 1

[8] Print Unsuccessful and Exit

Example

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

l u

K = 75
l = 1
u =8

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

l = 4 + 1 u

l = 1
u =8
mid = 4
K = 75 > A[4]

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

l = 4 + 1 u

l = 5
u =8
mid = 6
K = 75 = A[6]

Example

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

$K = 55$
$l = 1$
$u = 8$

l u

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

$l = 4 + 1$ u

$l = 1$
$u = 8$
$mid = 4$
$K = 55 > A[4]$

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

$l = 4 + 1$

$u = 6 - 1$

$l = 5$
$u = 8$
$mid = 6$
$K = 55 < A[6]$

1	2	3	4	5	6	7	8
15	25	35	45	65	75	85	95

l = 5

u= 5-1

l = 5

u =5

mid = 5

K = 55 < A[5]

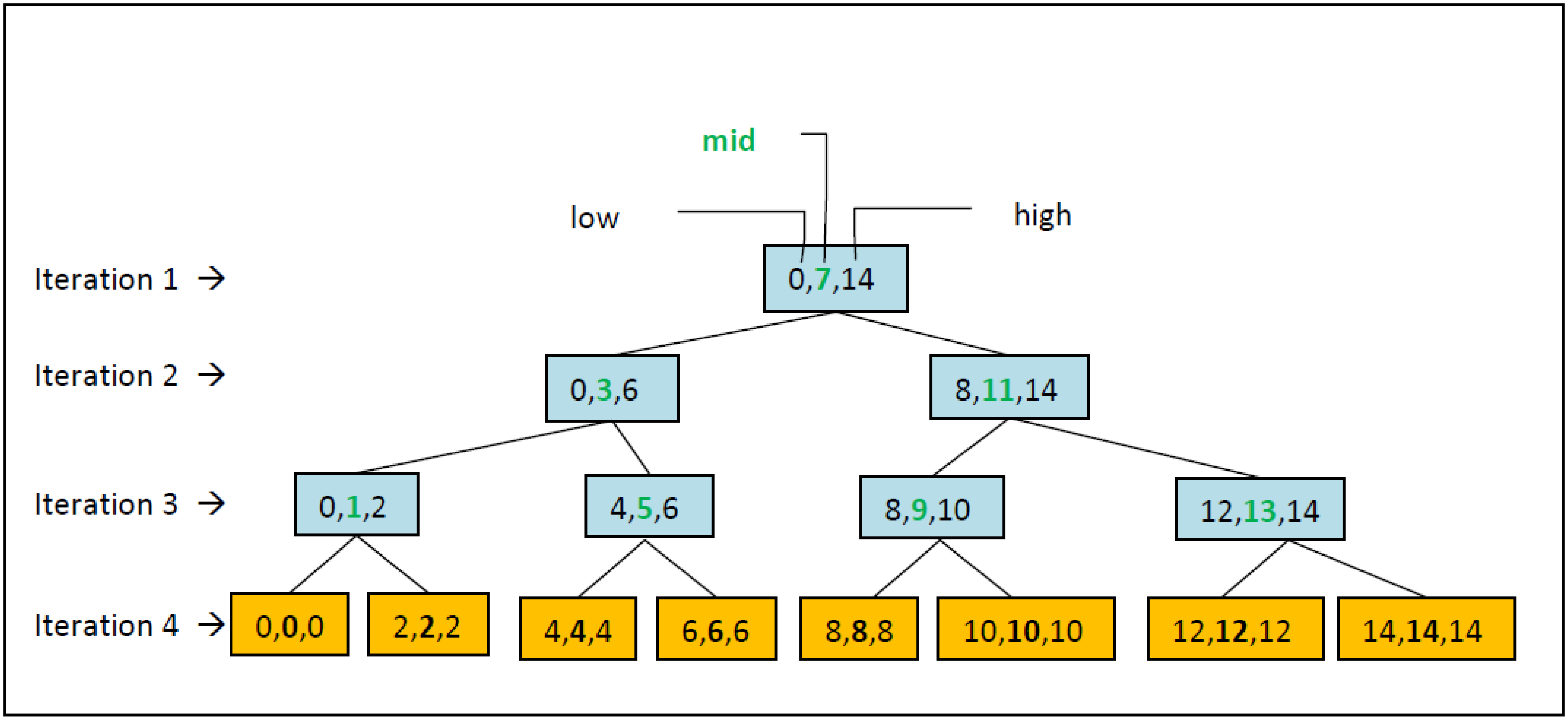
Binary Search Complexity Analysis

$T(n)$ = Number of Comparisons

Case 1: Key found in the first comparison

$T(n) = 1$, Best Case $= O(1)$

Index →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Values →	5	15	25	35	45	55	65	75	85	95	105	115	125	135	145



No. of iterations	Array Elements	No of Array Elements
1	A[7]	1
2	A[3],A[11]	2
3	A[1],A[5],A[9],A[13]	4
4	A[0],A[2],A[4],A[6],A[8],A[10],A[12],A[14]	8
.		.
.		.
k		2^{k-1}

k : max no. of iterations

n : size of array

$\Rightarrow 1+2+4+\ldots+2^{k-1} = n$

$\Rightarrow 2^k-1=n$

$\Rightarrow k = \log_2(n+1)$

Binary Search Complexity Analysis

Case 2: Key does not exist

$T(n) = \log_2(n+1)$

Worst Case = $O(\log_2(n))$

Case 3: Key is present at any location

No of Comparisons →	4	3	4	2	4	3	4	1	4	3	4	2	4	3	4
	5	15	25	35	45	55	65	75	85	95	105	115	125	135	145

Total. no. of comparisons, $\mathbf{A = 1.1 + 2.2 + 3.4 + 4.8 + \ldots + k.2^{k-1}}$

Which can be put as

$$\mathbf{1 + 2 + 4 + 8 + \ldots + 2^{k-1}}$$
$$\mathbf{+ 2 + 4 + 8 + \ldots + 2^{k-1}}$$
$$\mathbf{+ 4 + 8 + \ldots + 2^{k-1}}$$
$$\mathbf{+ 8 + \ldots + 2^{k-1}}$$
$$\vdots$$
$$\mathbf{+ 2^{k-1}}$$

$$\Rightarrow A = \mathbf{(2^k-1) + (2^k-2) + (2^k-4) + (2^k-8) + \ldots + (2^k-2^{k-1})}$$

$$A = k.2^k - (1+2+4+\ldots+2^{k-1})$$
$$= k.2^k - (2^k - 1)$$
$$= k(n+1) - n$$

Avg. no of comparisons, T(n) = A/n = $\boldsymbol{k + \frac{k}{n} - 1}$

$$= \log_2(n+1) + \frac{\log_2(n+1)}{n} - 1$$

Average Case = $O(\log_2(n))$

Fibonacci Search

$F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0$ and $F_1 = 1$

If we expand the nth Fibonacci number into the form of a recurrence tree, it results into a binary tree and we can term this as Fibonacci tree of order n.

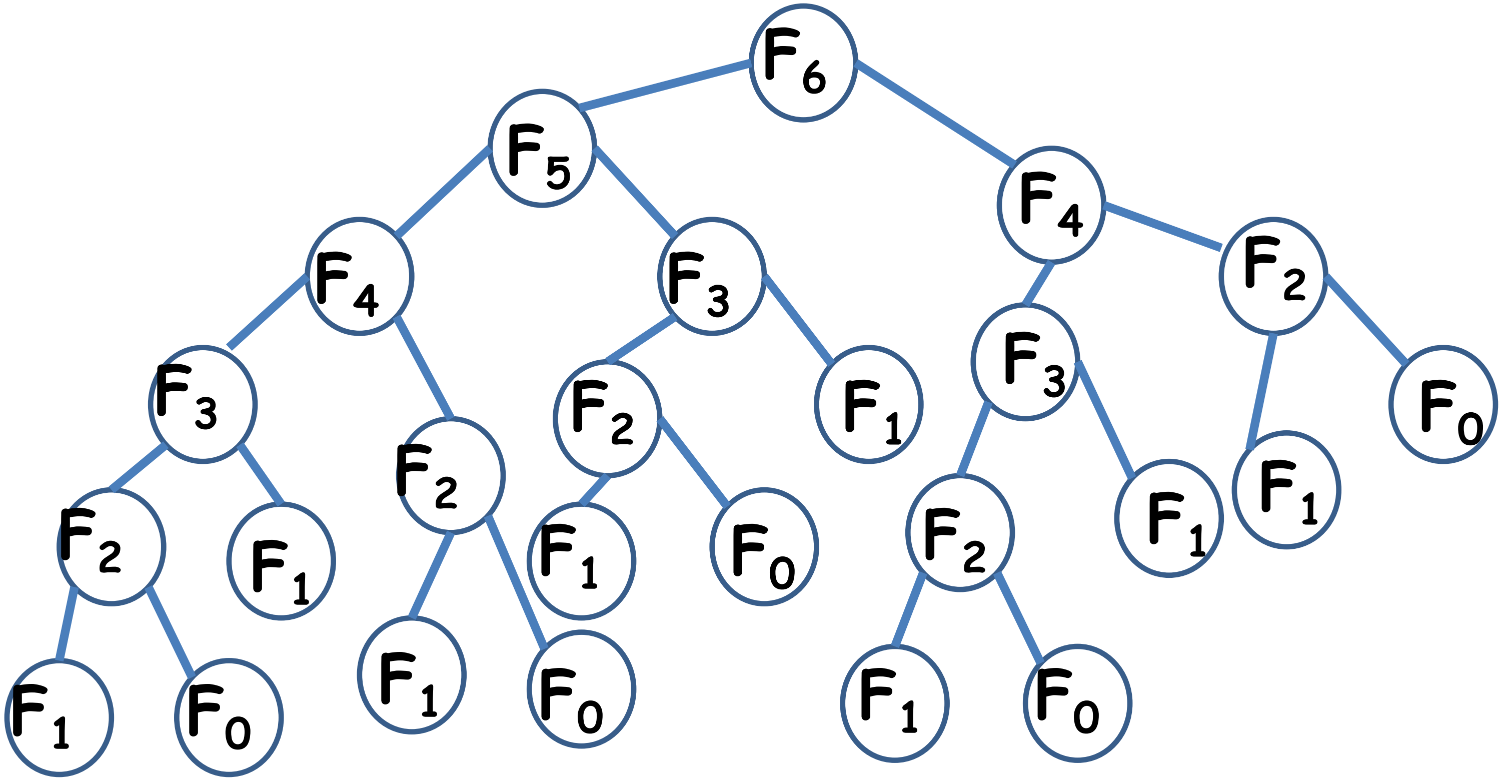
In a Fibonacci tree, a node correspond to F_i the ith Fibonacci number

F_{i-1} is the left subtree

F_{i-2} is the right subtree

F_{i-1} and F_{i-2} are further expanded until F_0 and $F_{1.}$

Fibonacci Tree



A Fibonacci Tree of Order 6

Apply the following two rules to obtain the Fibonacci search tree of order n from a Fibonacci tree of order n

Rule 1: For all leaf nodes corresponding to F_1 and F_0, we denote them as external nodes and redraw them as squares, and the value of each external node is set to zero. Value of each node is replaced by its corresponding Fibonacci number.

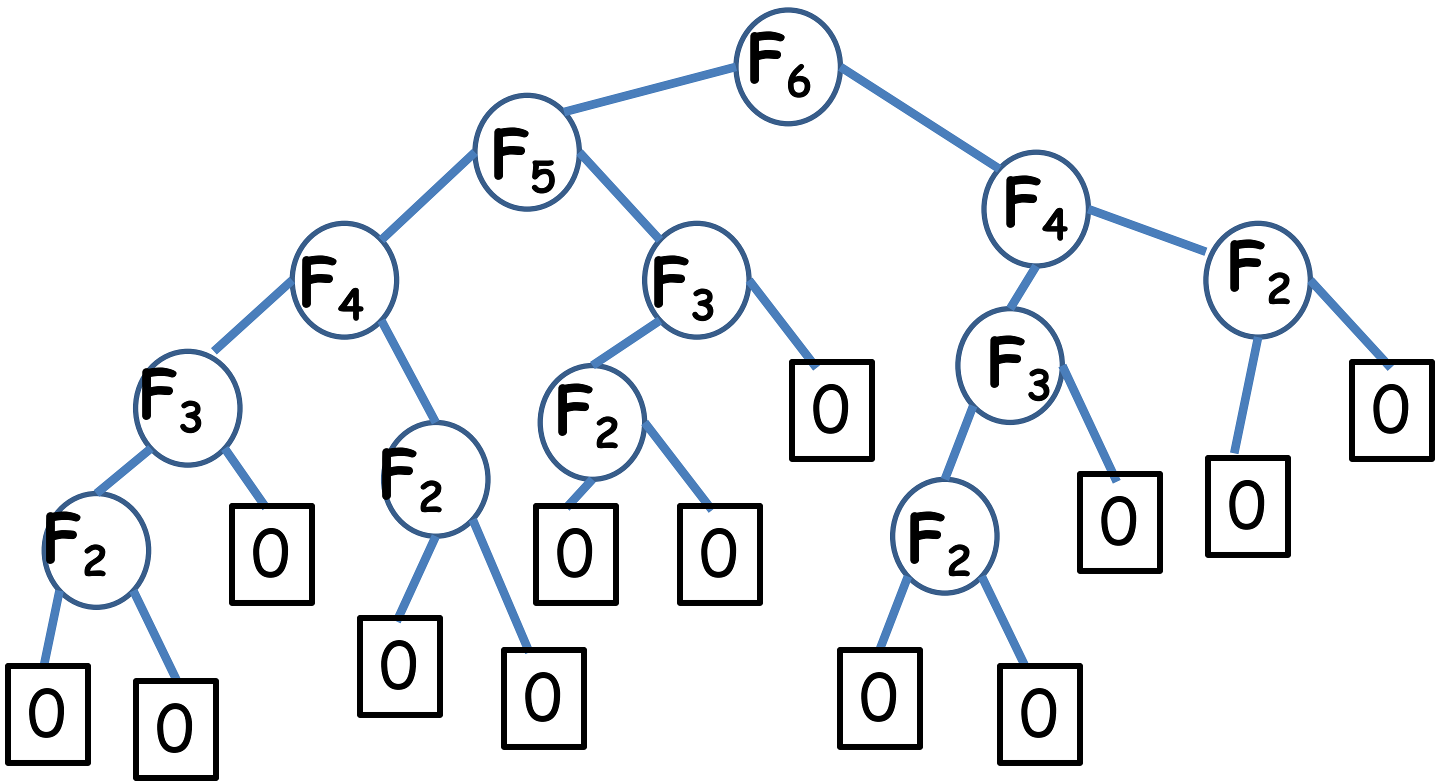
Rule 2: For all nodes corresponding to F_i ($i>=2$), each of them as a Fibonacci search tree of order i such that

(a) the root is F_i

(b) the left-subtree is a Fibonacci search tree of order i -1

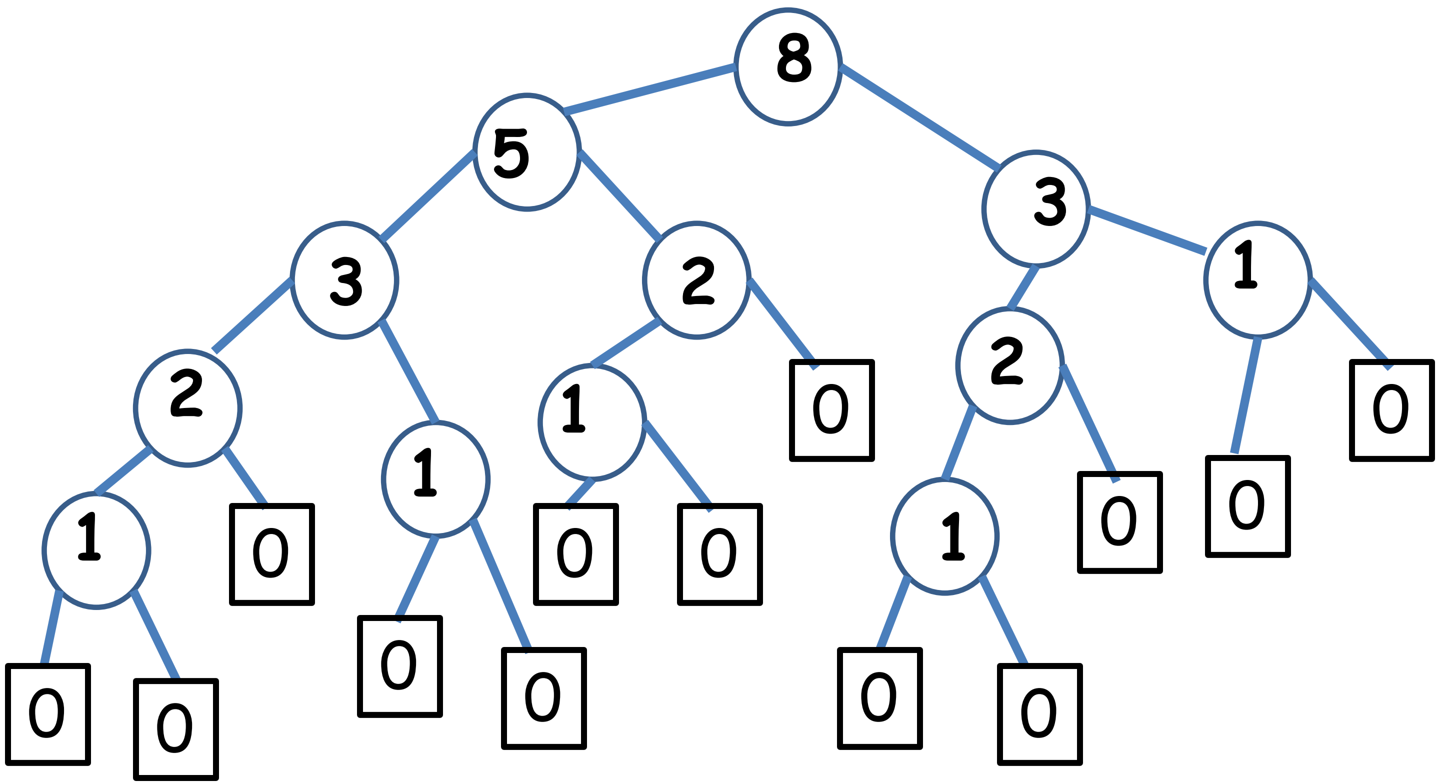
(c) the right-subtree is a Fibonacci search tree of order i -2 with all numbers increased by F_i

Fibonacci Search Tree



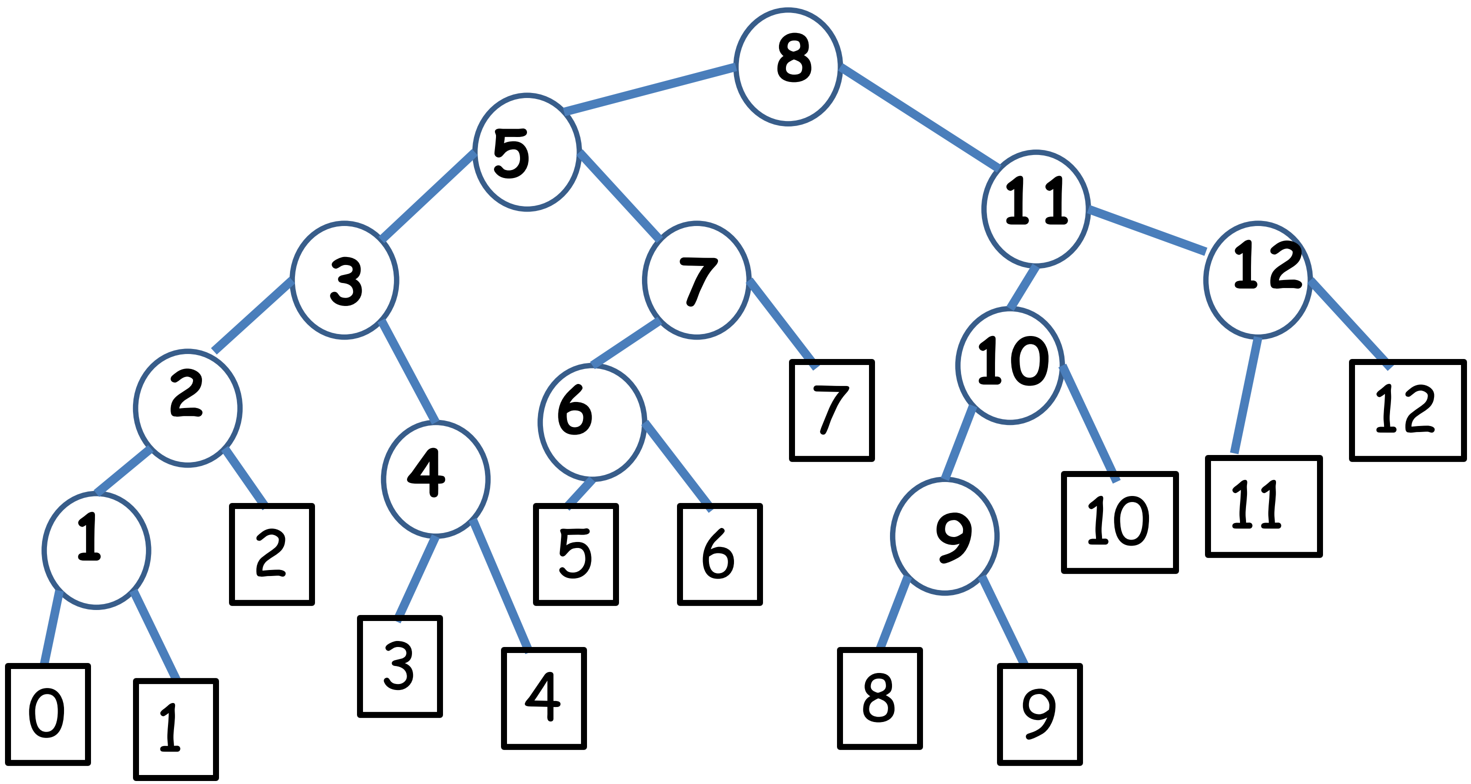
Rule 1 applied to Fibonacci Tree of Order 6

Fibonacci Search Tree



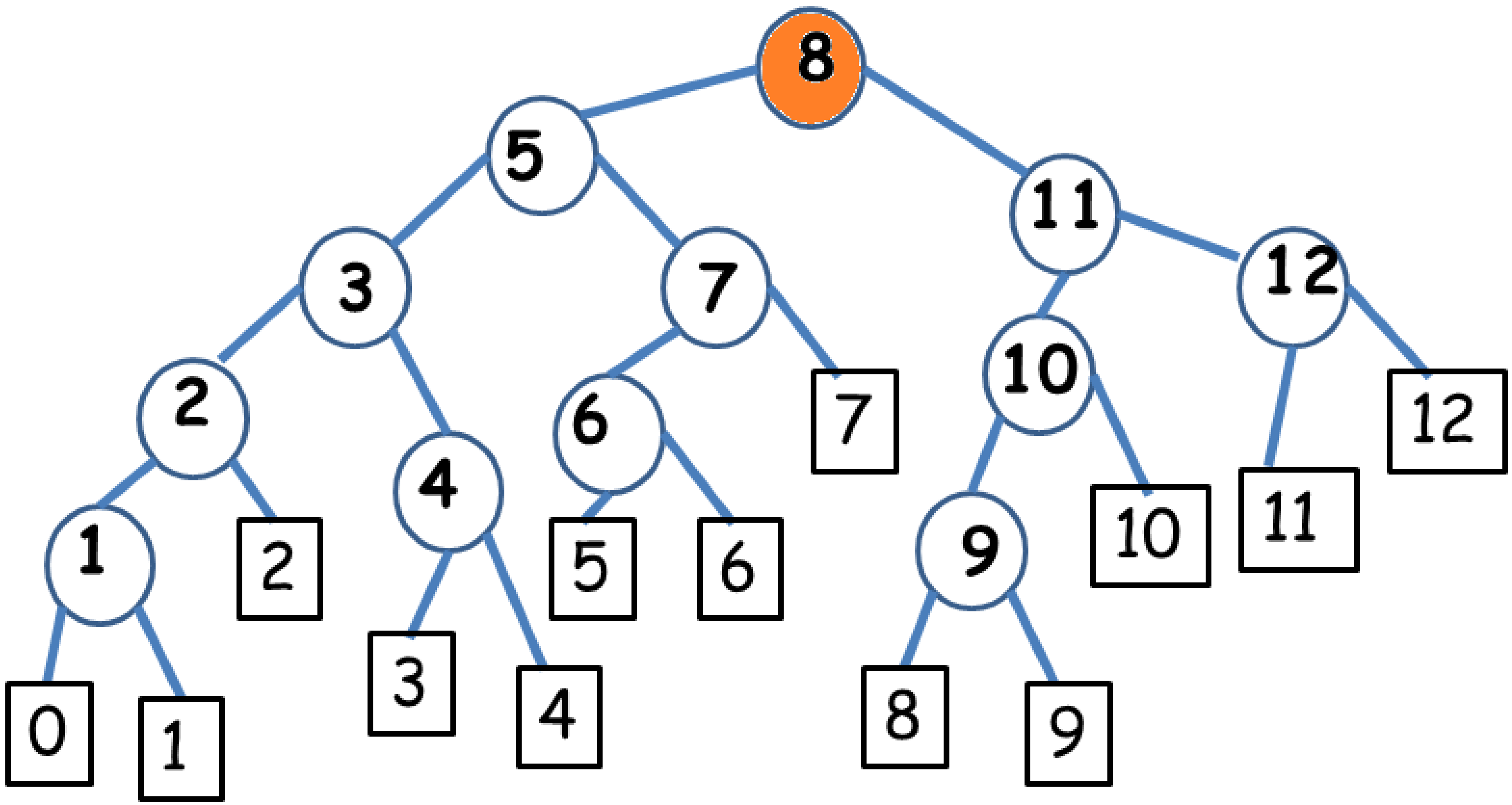
Rule 1 applied to Fibonacci Tree of Order 6

Fibonacci Search Tree

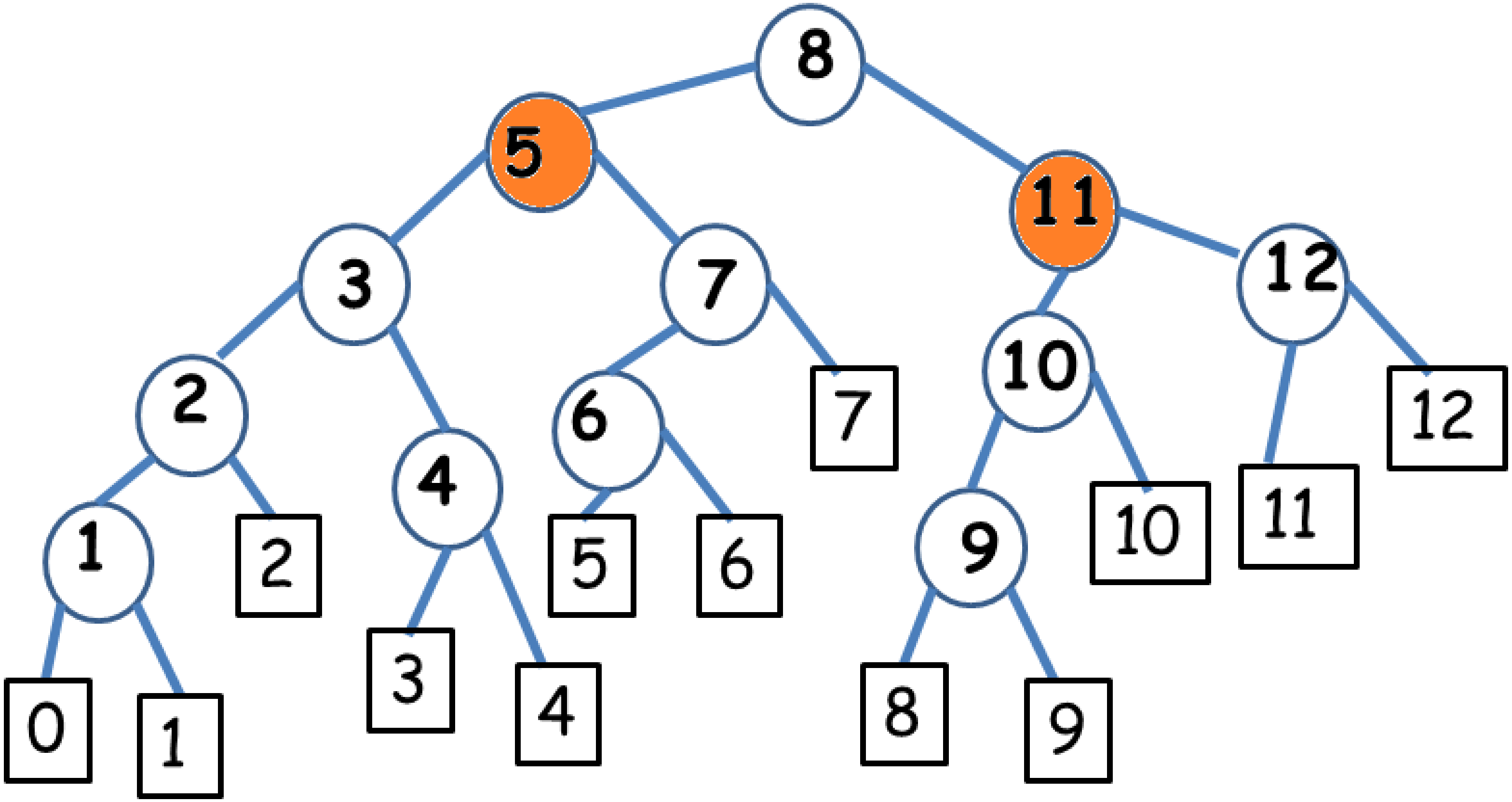


Rule 2 applied to Fibonacci Tree of Order 6

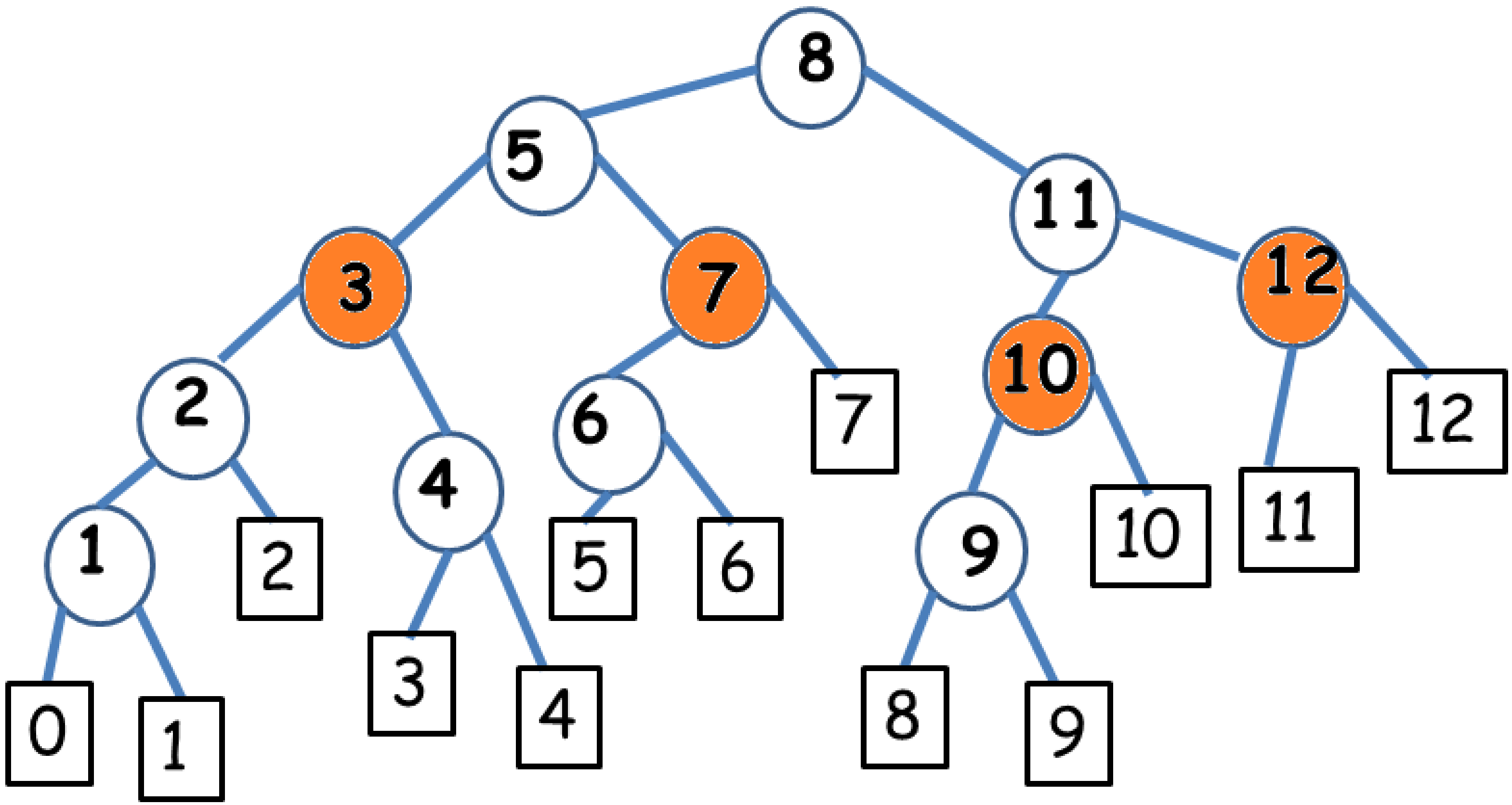
Evaluation #1



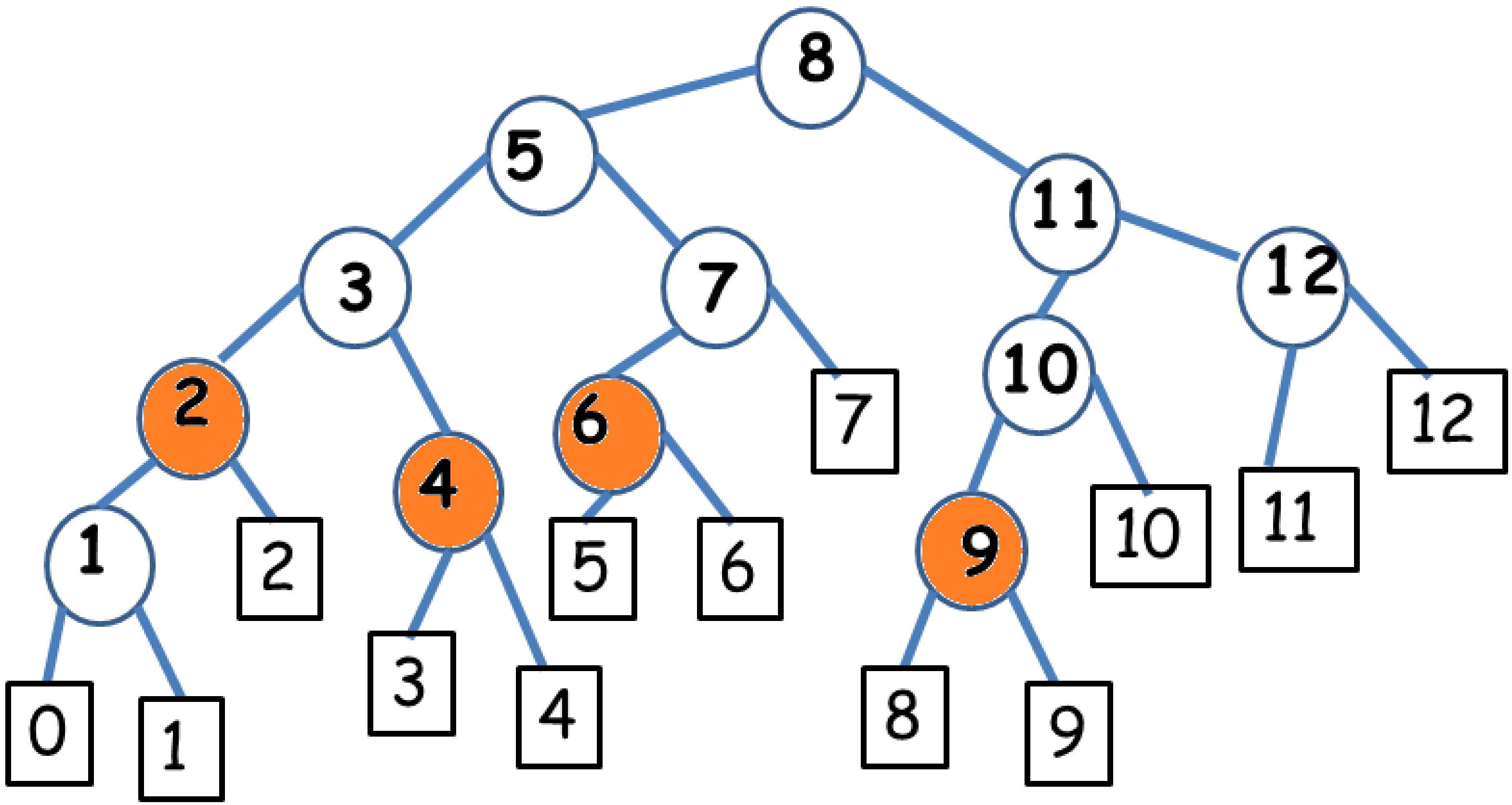
Evaluation #2



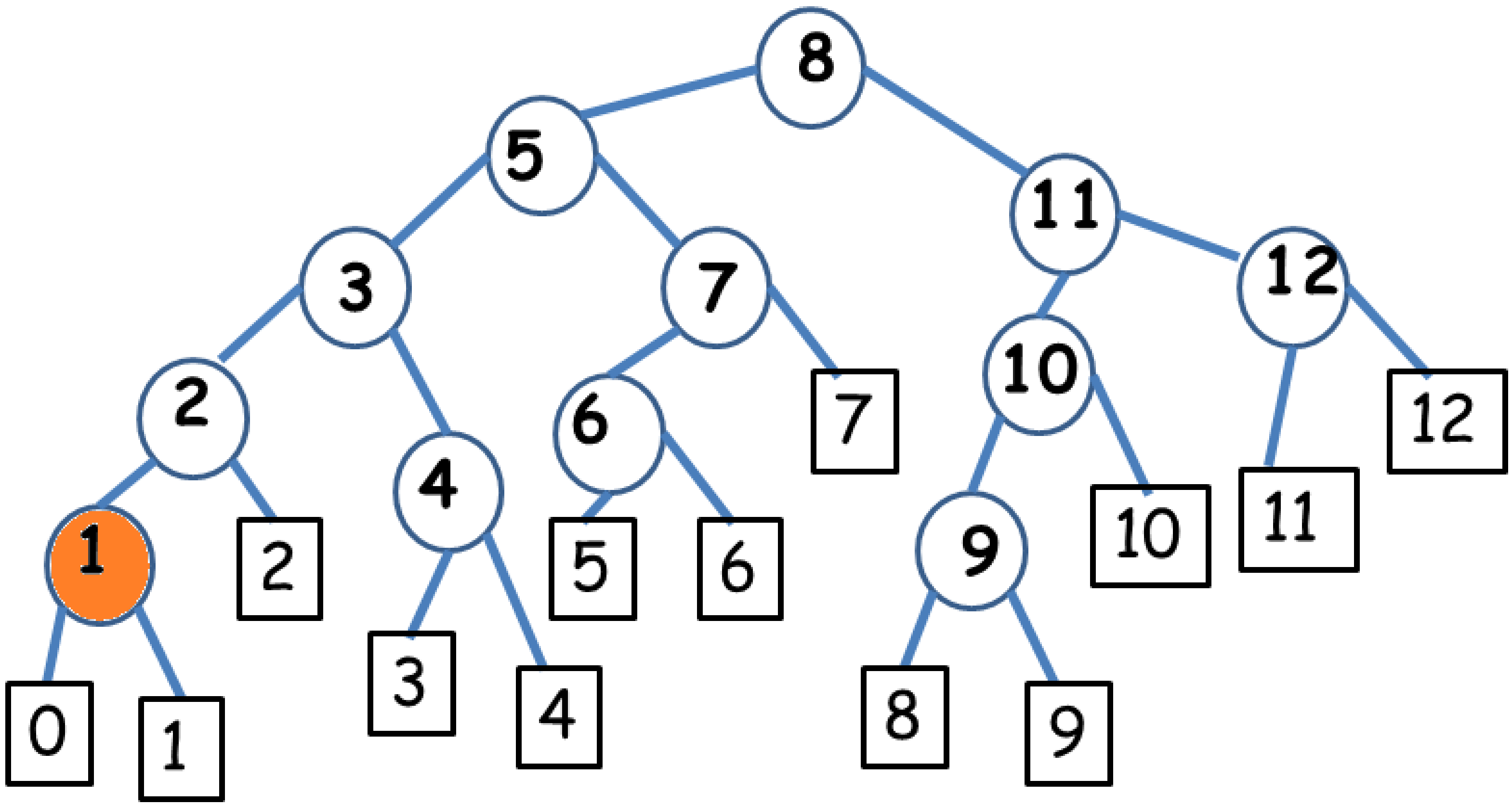
Evaluation #3



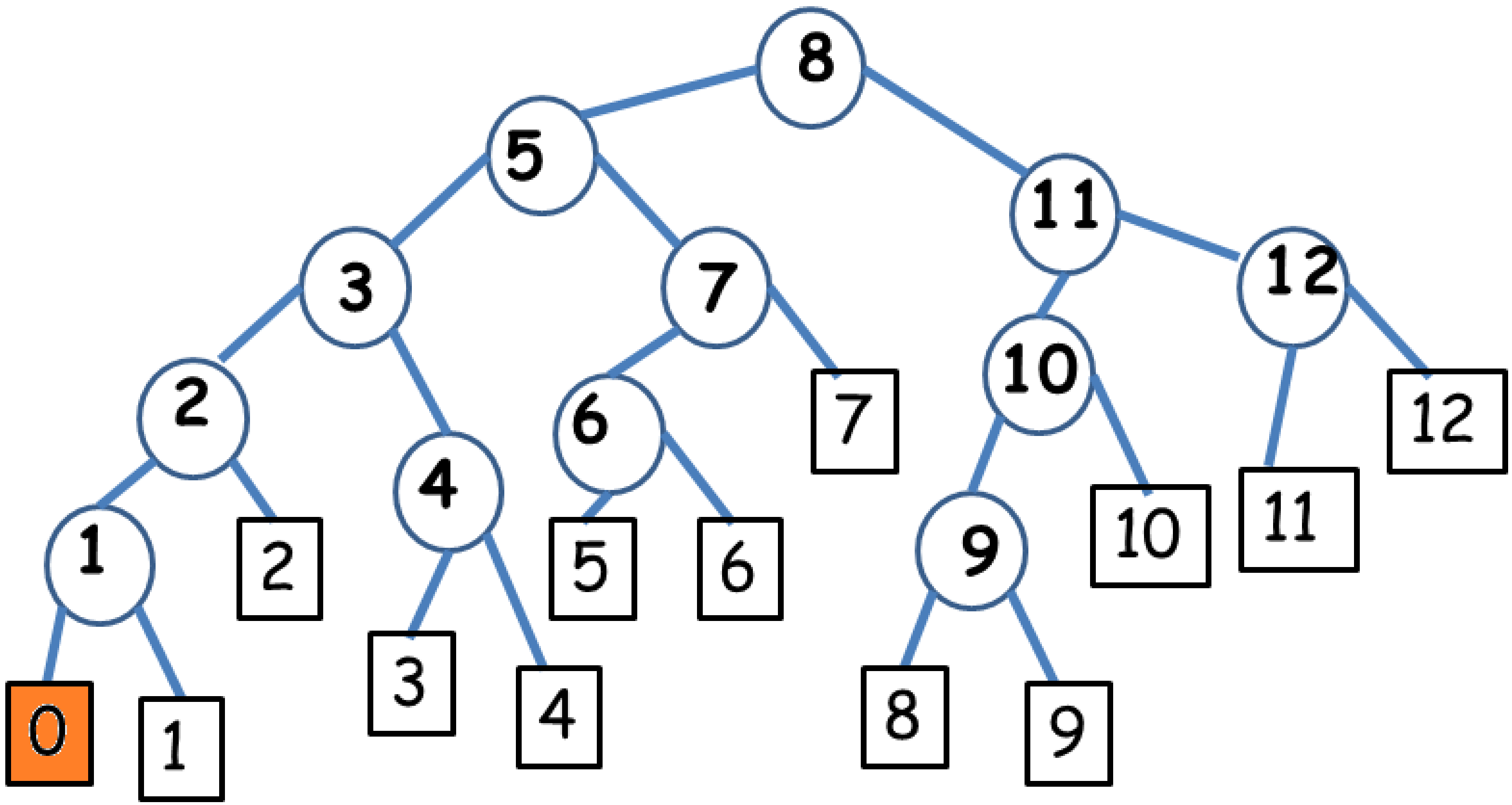
Evaluation #4



Evaluation #5



Evaluation #6



FibSearch(int arr[], int x, int n)

//Initialize fibonacci numbers

```
fibMMm2 = 0                         // (m-2)'th Fibonacci No.
fibMMm1 = 1                         // (m-1)'th Fibonacci No.
fibM = fibMMm2 + fibMMm1  // m'th Fibonacci
```

//fibM is going to store the smallest Fibonacci Number greater than or equal to n

```
while (fibM < n)
  {
     fibMMm2 = fibMMm1;
     fibMMm1 = fibM;
     fibM  = fibMMm2 + fibMMm1;
  }
```

```
// Marks the eliminated range from front
offset = -1;
// finding 'i' which is the comparison point
  i = min(offset+fibMMm2, n-1)
```

```
/* If x is greater than the value at index
fibMm2,
Eliminate the subarray array from offset
upto i */

   if (arr[i] < x)
      {
         fibM  = fibMMm1;
         fibMMm1 = fibMMm2;
         fibMMm2 = fibM - fibMMm1;
         offset = i;
      }
```

```
/* If x is less than the value at index
fibMm2,
Eliminate the subarray after i+1  */

   else if (arr[i] > x)
     {
        fibM  = fibMMm2;
        fibMMm1 = fibMMm1 - fibMMm2;
        fibMMm2 = fibM - fibMMm1;
     }
```

This goes iteratively until either arr[i] = x or fibM>1

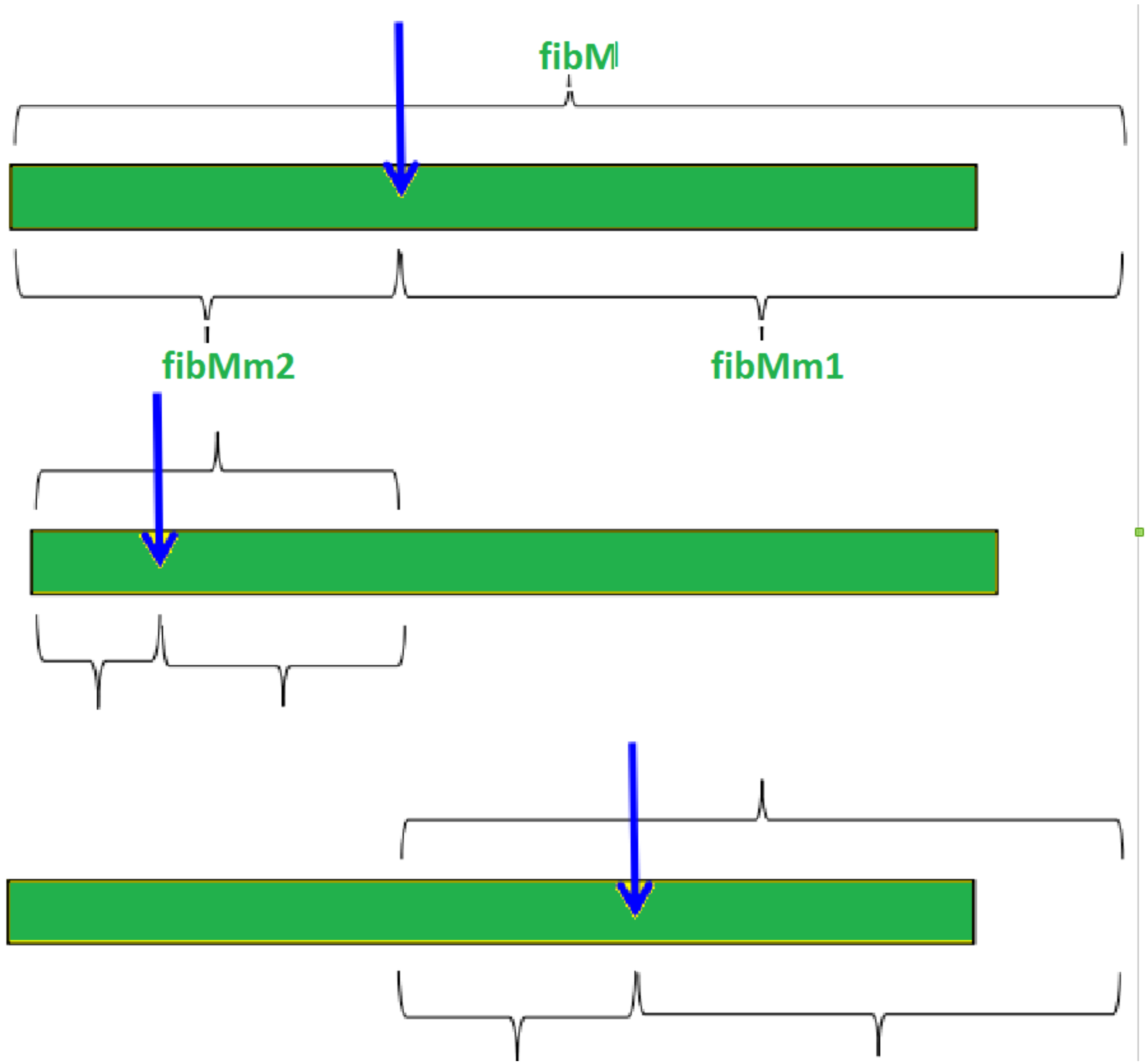

fibM
fibMm2
fibMm1

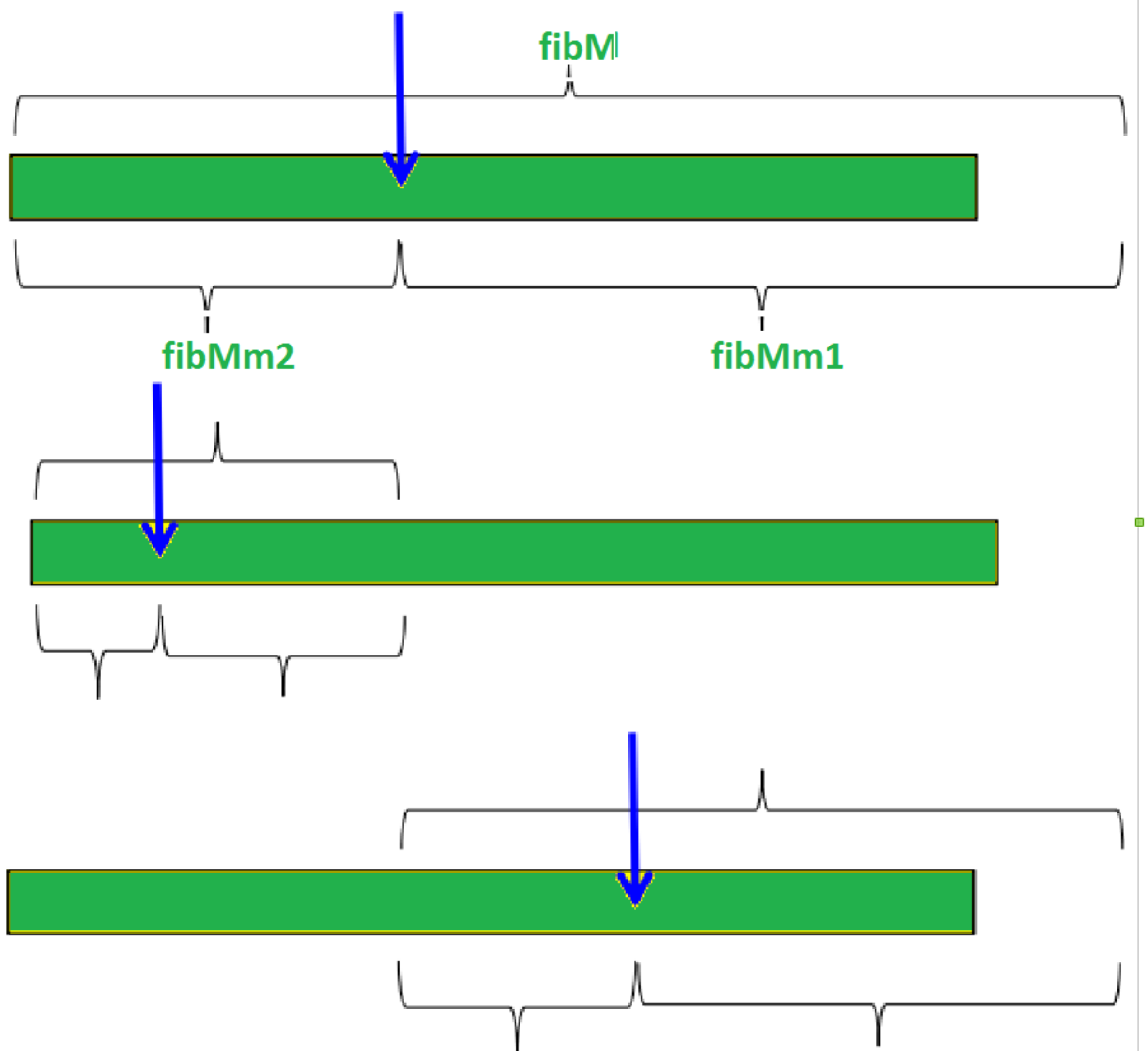
Illustration assumption:

1-based indexing. Target element x is 85. Length of array n = 11.

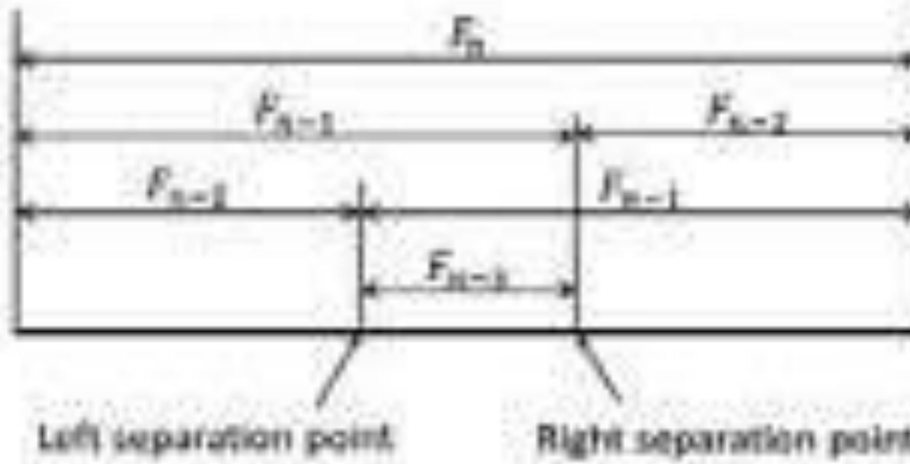
i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	**10**	**22**	**35**	**40**	**45**	**50**	**80**	**82**	**85**	**90**	**100**	**-**	**-**

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	**10**	**22**	**35**	**40**	**45**	**50**	**80**	**82**	**85**	**90**	**100**	**-**	**-**

fibMm2	***fibMm1***	***fibM***	***offset***	**i**=min(offset+fibL n)	***arr[i]***	***Consequence***
5	8	13	0	5	45	Move one down, reset offset
3	5	8	5	8	82	Move one down, reset offset
2	3	5	8	10	90	Move two down
1	1	2	8	9	85	Return i


fibM
fibMm2
fibMm1

Fibonacci search method (F_x:Fibonacci number)



Algorithm

Elements are in sorted order. Number of elements **n** is related to a perfect Fibonacci number F_{k+1} such that $F_{k+1} = n+1$

[1] $i = F_k$

[2] $p = F_{k-1}$, $q = F_{k-2}$

[3] If ($K < K_i$) then

[4] If (q==0) then Print "unsuccessful and exit"

[5] Else $i = i - q$, $p_{old} = p$, $p = q$, $q = p_{old} - q$

[6] Goto step 3

[7] If ($K > K_i$) then

[8] If ($p == 1$) then Print "Unsuccessful and Exit"

[9] Else $i = i + q$, $p = p-q$, $q=q-p$

[10] Goto step 3

[11] If ($k == K_i$) then Print "Successful at ith location"

[12] Stop

Example

1	2	3	4	5	6	7	8	9	10	11	12
15	20	25	30	35	40	45	50	65	75	85	95

$K = 25$

Initialization: $i = F_k = 8$, $p = F_{k-1} = 5$, $q = F_{k-2} = 3$

Iteration 1;

$K_8 = A[8] = 50$, $K < K_8$, $q == 3$

$i = i - q = 8 - 3 = 5$, $p_{old} = p = 5$

$p = q = 3$, $q = p_{old} - q = 5-3 = 2$

Iteration 2;

$K_5 = A[5] = 35$, $K < K_5$, $q == 2$

$i = i - q = 5 - 2 = 3$, $p_{old} = p = 3$

$p = q = 2$, $q = p_{old} - q = 3-2 = 1$

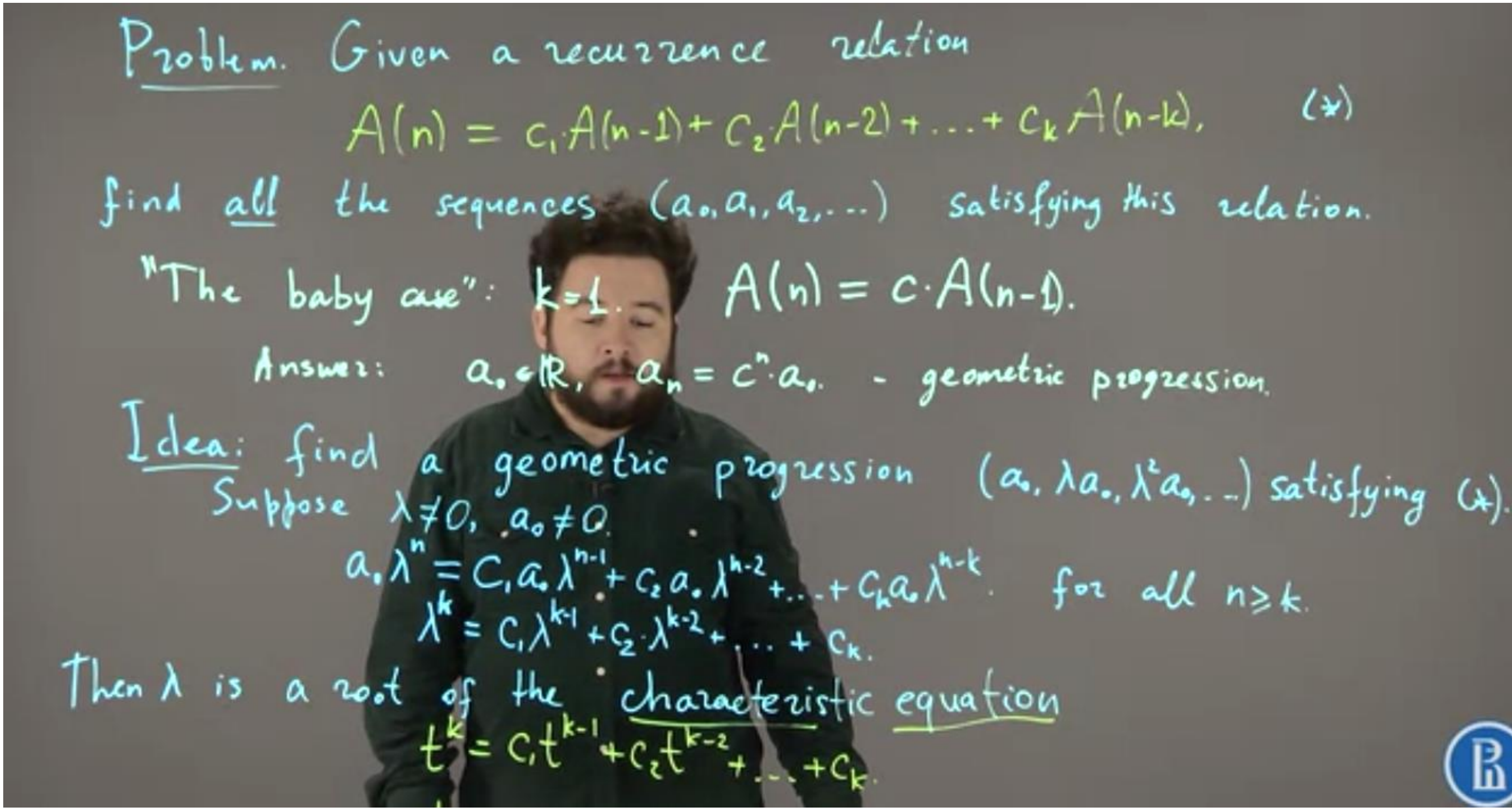
1	2	3	4	5	6	7	8	9	10	11	12
15	20	25	30	35	40	45	50	65	75	85	95

Iteration 2;

K_3 = A[3] = 25,   K = K_3

Search is successful

Finding time complexity of Fibonacci search



Finding time complexity of Fibonacci search

Now Fibonacci is defined as

$F(n) = F(n-1) + F(n-2)$

The characteristic equation for this function will be

$x^2 = x + 1$

$x^2 - x - 1 = 0$

Solving this by quadratic formula we can get the roots as

$x = (1+\sqrt{5})/2$ and $x = (1 - \sqrt{5})/2$

Now we know that solution of a linear recursive function is given as

$F(n) = (\alpha_1)^n + (\alpha_2)^n$

where α_1 and α_2 are the roots of the characteristic equation.

So for our Fibonacci function $F(n) = F(n-1) + F(n-2)$ the solution will be

$F(n) = ((1+\sqrt{5})/2)^n + ((1-\sqrt{5})/2)^n$

Clearly $T(n)$ and $F(n)$ are asymptotically the same as both functions are representing the same thing.

Hence it can be said that

$T(n) = O(((1+\sqrt{5})/2)^n + ((1-\sqrt{5})/2)^n)$

or we can write below (using the property of Big O notation that we can drop lower order terms)

$T(n) = O(((1+\sqrt{5})/2)^n$

$T(n) = O(1.6180)^n$

This is the tight upper bound of fibonacci.\

Comparison of Fibonacci search with binary search

1. Fibonacci Search divides given array in unequal parts

2. Binary Search uses division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.

3. Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.