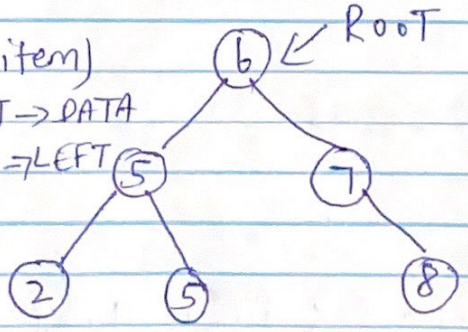
Mahesh Koppala
1001764522

(1) (a) It is a binary search tree because in this tree, the key of any node is greater than all keys occuring in it's left sub tree and less than all keys occuring in it's right subtree.

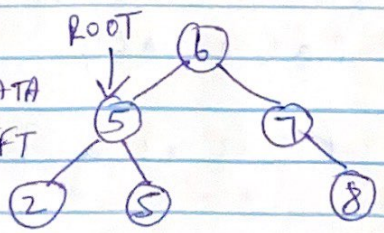
(b)
(1) Start searching from root.
(2) Compare the element with root, if less than root, then recurse for left otherwise recurse for right (if element is greater than root)
(3) If Element is found return true otherwise return false.

Key = 2 (item)
item < ROOT → DATA
ROOT = ROOT → LEFT



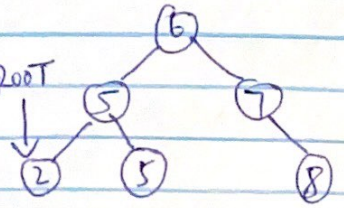
Step1

ITEM = 2
ITEM < ROOT → DATA
ROOT = ROOT → LEFT



Step2

Item = 2
Item = Root → DATA
Return root

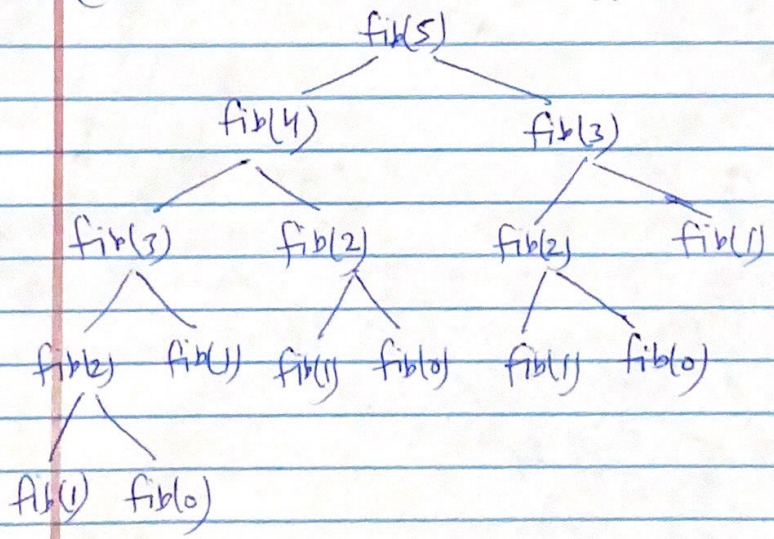


Step3

(2) Given fibonacci sequence is

$$fib(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ fib(n-2) + fib(n-1) & \text{for } n > 1 \end{cases}$$

(a) Recursion tree for fib(5)



(b) Dynamic Programming is a technique to solve the recursive problems in more effecient manner in the fibonacci sequence.

Each number in the fibonacci sequence is the sum of the two previous numbers in the sequence.

By not computing the full recursive tree on each iteration, we're essentially reduced the running time. New fibonacci number function can compute the additional values in linear time vs exponential time.

For fibonacci numbers, the answer is pretty obvious, so it is fairly difficult to do a brute-force solution to this problem. The dynamic programming solution is much more concise and a natural fit for the problem definition, so we'll skip creating an unnecessarily complicated naive solution and jump straight to the Dynamic programming solution

(c) Top down algorithm for fib(n)

top down approach breaks the larger problem into multiple subproblems

```
Fib(n)
    if n==0 || n==1 return n;
  otherwise, compute subproblem results recursively
  return Fib(n-1) + Fib(n-2);
```

example:
If we want to compute Fibonacci(4), the top down approach will do the following

Fibonacci(4) → Compute Fibonacci(3) & Fibonacci(2) and return results
Fibonacci(3) → compute Fibonacci(2) & Fibonacci(1) and return results
Fibonacci(2) → compute Fibonacci(1) & Fibonacci(0) and return results.

finally fibonacci(1) will return 1 and
fibonacci(0) will return 0

(d) Bottom up algorithm for fib(n)

computing result for the subproblem. Using the subproblem result, solve another subproblem and finally solve the whole problem.

Bottom-up approach algorithm:

1. Set Fib[0] = 0
2. Set Fib[1] = 1
3. From index 2 to n, Compute result using the below formula.
   Fib[index] = Fib[index-1] + Fib[index-2]
4. The final result will be stored in Fib[n].

Let's find the n^{th} member of a fibonacci series

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(2) = 1 (Fibonacci(0) + Fibonacci(1))

Fibonacci(3) = 2 (Fibonacci(1) + Fibonacci(2))

We can solve the problem step by step

1. Find 0^{th} member
2. Find 1^{st} member
3. Calculate the 2^{nd} member using 0^{th} and 1^{st} member.
4. Calculate the 3^{rd} member using 1^{st} and 2^{nd} member.
5. By doing this, we can easily find the n^{th} member.

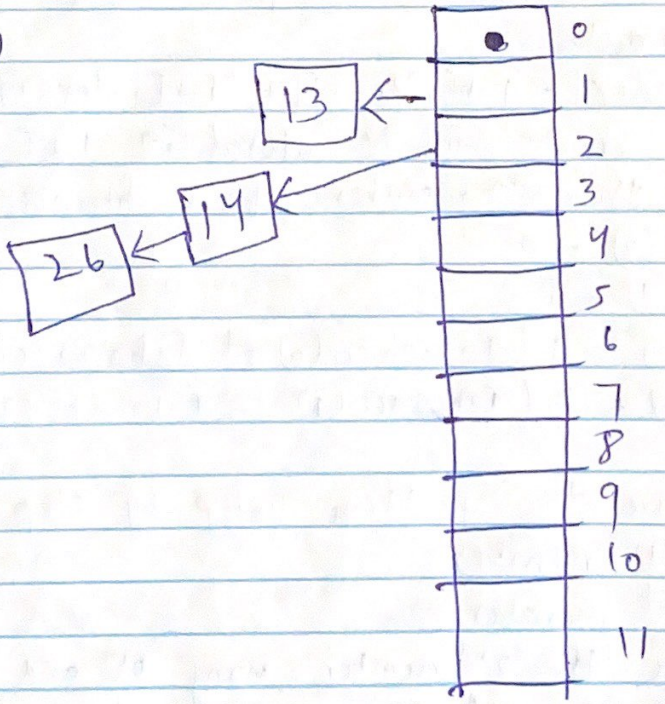
(3) $h(k) = k \,\%\, 12$

(a) $h(13) = 13 \,\%\, 12$
$= 1$

(b) $h(14) = 14 \,\%\, 12$
$= 2$

(c) $h(26) = 26 \,\%\, 12$
$= 2$ (collision)

(d)



If two items hash to same slot, we must have a systematic method for placing the second item in hash table. This is known as collision resolution. h(14) and h(26) is an example of this.

To handle collision problem, we need to allow each slot to hold a reference to collection of items chaining allows many items to exist at the same location in hash table when collision happen, item is still placed in proper slot of hash table.

While searching for an item, hash function is used to generate slot where it should reside. Each slot holds a collection, we use a searching technique to decide whether item is present.

(4) Step 1:
sort the activities in ascending order w.r.t their finish times.

Activity	a3	a1	a2	a7	a8	a4	a6	a5
start time	1	1	0	3	4	4	5	2
finish time	2	3	4	5	5	6	8	9

Add the least finishing time to the solution {a3}

step2:

Activity	a1	a2	a7	a8	a4	a6	a5
start time	1	0	3	4	4	5	2
finish time	3	4	5	5	6	8	9

Now check start time of the next activity with the finish time of the a3. The start time of the next activity must be greater than or equal to previously added activity to the solution.

start time(a1) < finish time(a3) because 1<2
a1 is not added to the solution.
solution set {a3}

step3:

Activity	a2	a7	a8	a4	a6	a5
start time	0	3	4	4	5	2
finish time	4	5	5	6	8	9

Now check start time of the next activity a_2 with finish time of a_3.

start time(a_2) < finish time(a_3) because $0 < 2$
a_2 is not added to the solution.
solution set {a_3}

step 4:

Activity	a_7	a_8	a_4	a_6	a_5
Start time	3	4	4	5	2
Finish time	5	5	6	8	9

start time(a_7) > finish time(a_3) because $3 > 2$
a_7 is added to the solution.
Solution set {a_3, a_7}

step 5:

Activity	a_8	a_4	a_6	a_5
Start time	4	4	5	2
finish time	5	6	8	9

start time(a_8) < finish time(a_7) because $4 < 5$
a_8 is not added to the solution
solution set {a_3, a_7}

step 6:

Activity	a_4	a_6	a_5
Start time	4	5	2
finish time	6	8	9

start time(a4) < finish time(a7) because 4<5
a4 is not added to the solution.
solution set: {a3, a7}

Step7:

Activity	a6	a5
Start time	5	2
Finish time	8	9

start time(a6) >= finish time(a7) because 5>=5
a6 is added to the solution.
solution set {a3, a7, a6}

Step8:

Activity	a5
start time	2
finish time	9

start time(a5) < finish time(a6) because 2<8
a5 is not added to the solution.
solution set {a3, a7, a6}

final solution set {a3, a7, a6}