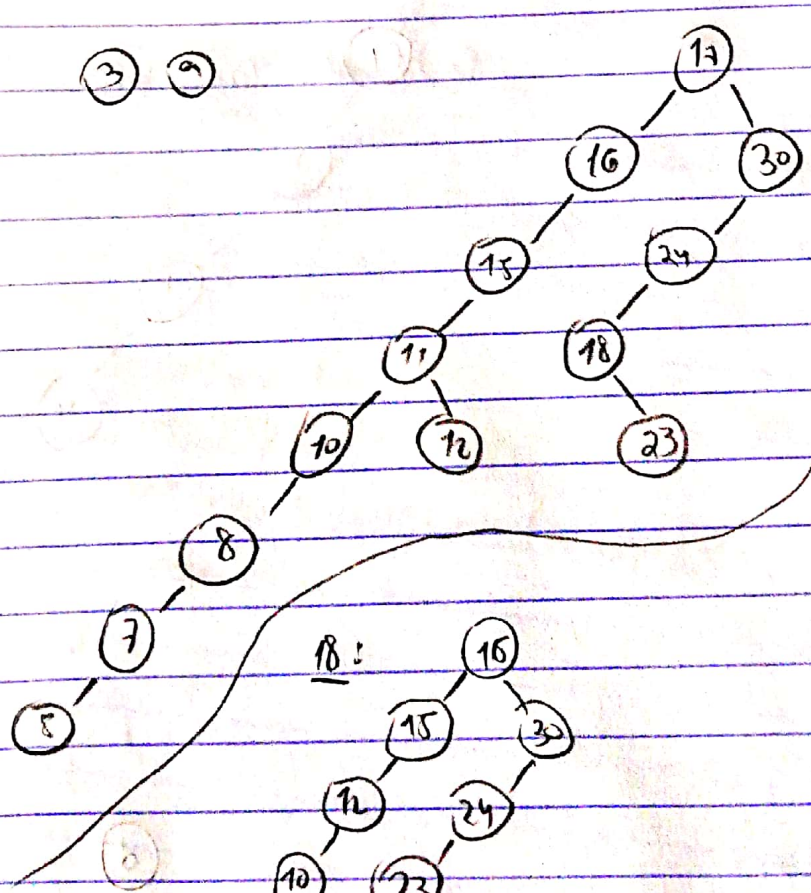
Data Structures #6

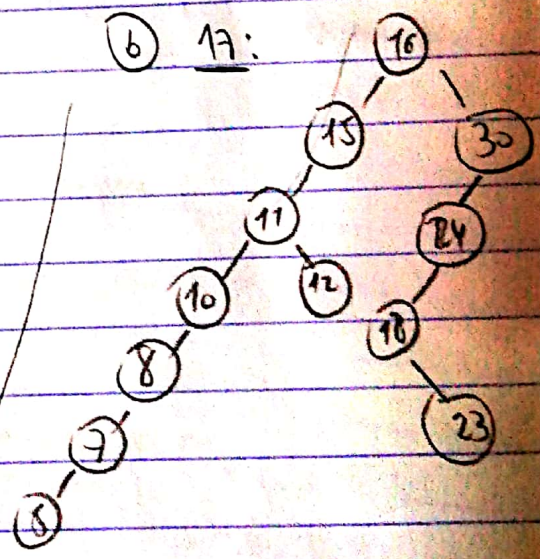
② nTree (n)

```
{ if n=1
    return 1;
  if n=0
    return 1
  sum=0;
  for i=0 to n-1
  sum = sum + nTree(i) * nTree(n-i-1)
  return sum
}
```

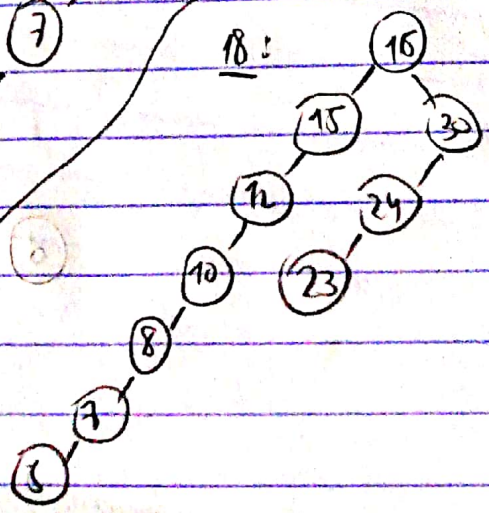
③ a



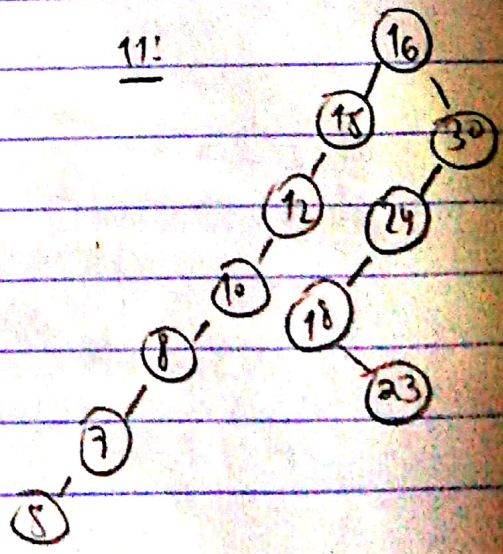
b 17:



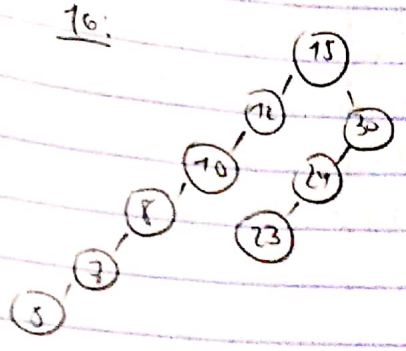
18:



11:



16:



(4) (a) $O(n^2)$ if you don't balance the tree each time you insert.
However, if you do balance the tree, it's $O(n\log(n))$

(b) the height of the binary tree will not exceed $\log(n)$
Since the array is sorted the nodes can be inserted from top to bottom, and the tree will always be complete, so to insert all nodes complexity is $O(n\log(n))$

(5) calling the treeMinimum(T) is $O(\log n)$
calling TreeSuccessor in a balanced tree, half of them will be ancestors, and from them half will be from the left $(\frac{1}{4}n)$ → only one step.
The pattern is $\sum_{i=1} \frac{1}{2^{i+1}}$ → which goes to 1. when it will search the right, it will also be going to one
so if we call $O(1)$, $n-1$ times, the total runtime is $O(n)$
therefore: total Runtime $\Theta(n)$

(1) (a)
```
int largeUnifunc (T)
{ int L, R, num = value (T)
  int max = biggestpath (T, num, L, R)
  return max
}
```

```
int biggestpath (T, num, L, R)
{ temp = 0
  if (temp == 1) L = 0, R = 0
  if (value(T) != num)
  { temp = 1
    return -1
  }
  L = biggestpath (Left (T)) + 1
  R = biggestpath (Right (T)) + 1
  if (max == 0) max = L + R
  else Return max (max, L + R);
}
```

algorithm my code take the value at the root of the tree and calls the function biggest path and finds the biggest unified path.
the runtime of the algorithm is O(n).

(b) bool cunified_path (T)
{ int num = value(T);
bool temp = false;
bool phase = search (T, num, temp)
return phase;
}

bool search (T, num, temp)
{ if (value (T) != num)
return false
else if (T == NULL)
return true;
Temp = search(LeftT(T))
Temp = search(Right (T))
return = Temp;
}

algorithm: The function cunified_path gives the value of the root of the tree and calls the function search to see if there exists a complete unified path
runtime → is O(n)