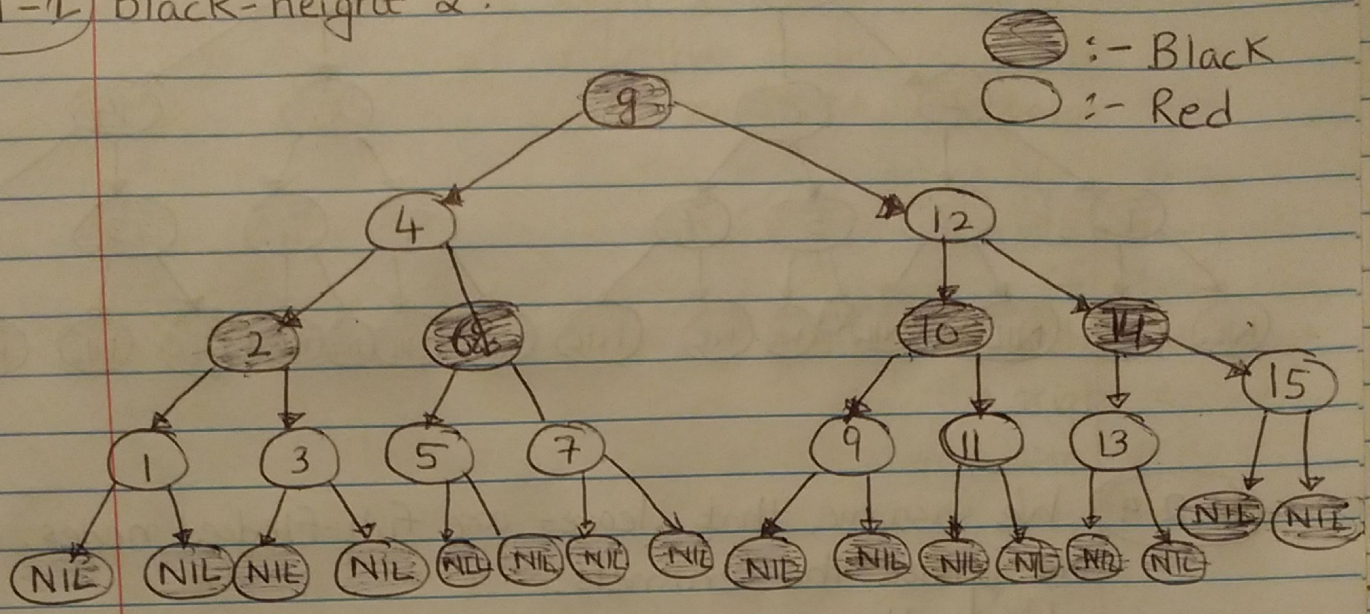
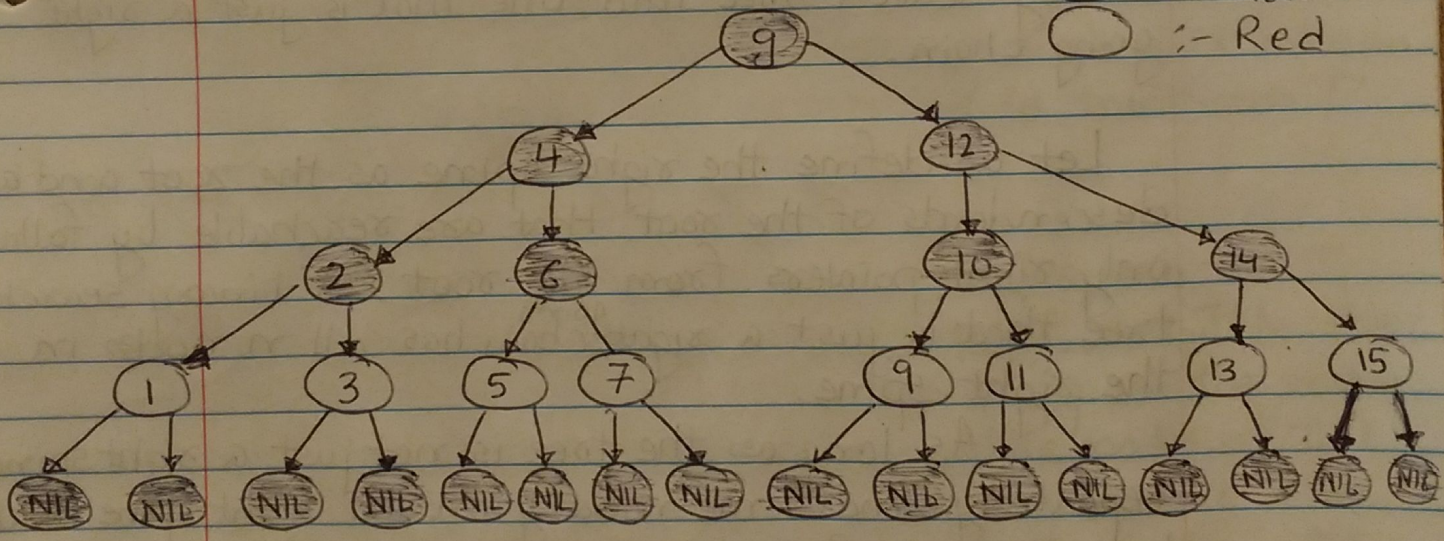
Assignment - 6

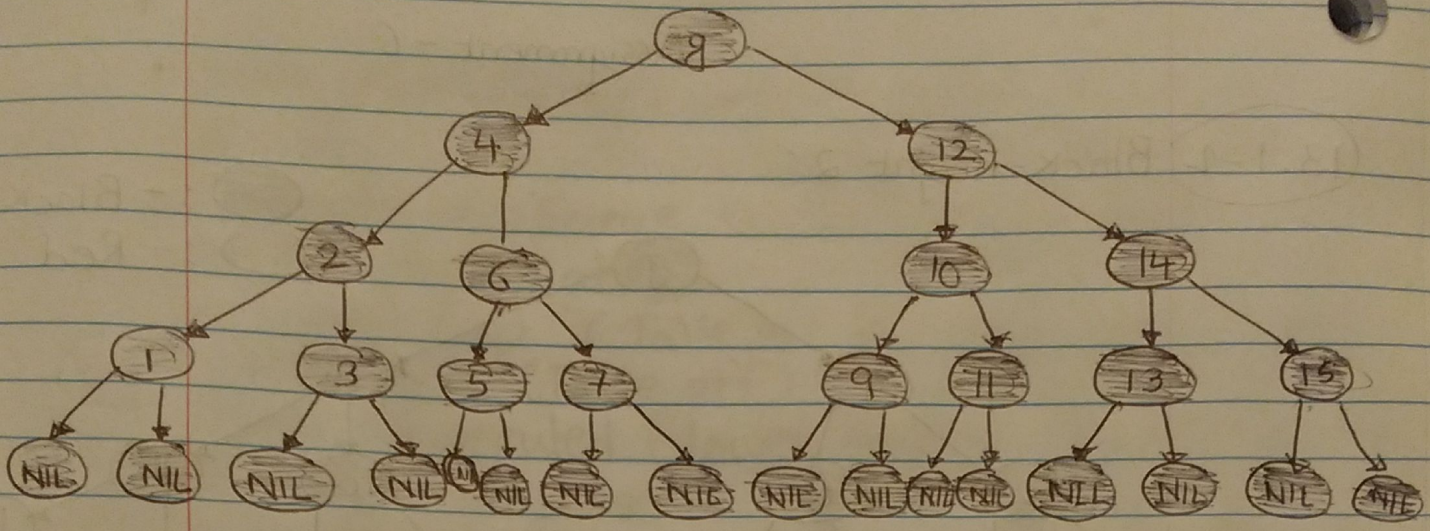
13.1-1 Black-height 2:



Black-height 3:



Black-height 4:

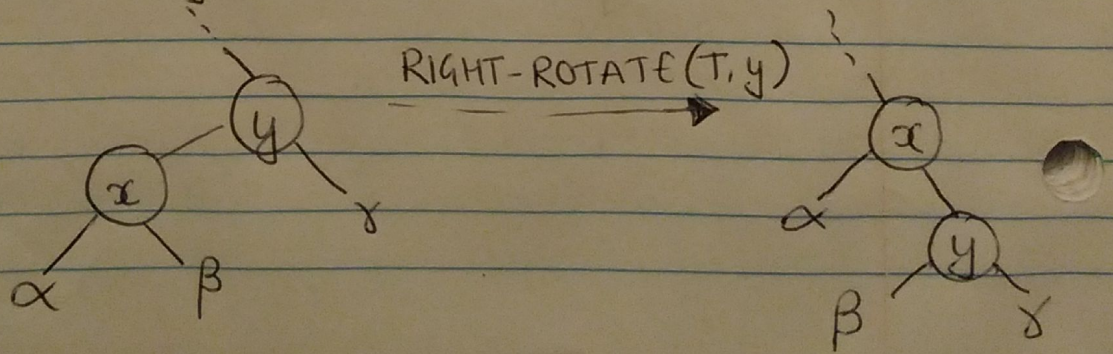


(13.2.4) We assume that leaves are full-fledged nodes, and we ignore the sentinels.

Using Hint give, we start by showing that with at most $n-1$ right rotations, we can convert any binary search tree into one that is just a right going chain.

Let us define the right spine as the root and all descendants of the root that are reachable by following only right pointers from the root. A binary search tree that is just a right going chain has all n nodes in the right spine.

As long as the tree is not just a right spine, repeatedly find some node y on the right spine that has a non-leaf child x and then perform a right rotation on y:



α, β and γ can be an empty subtree

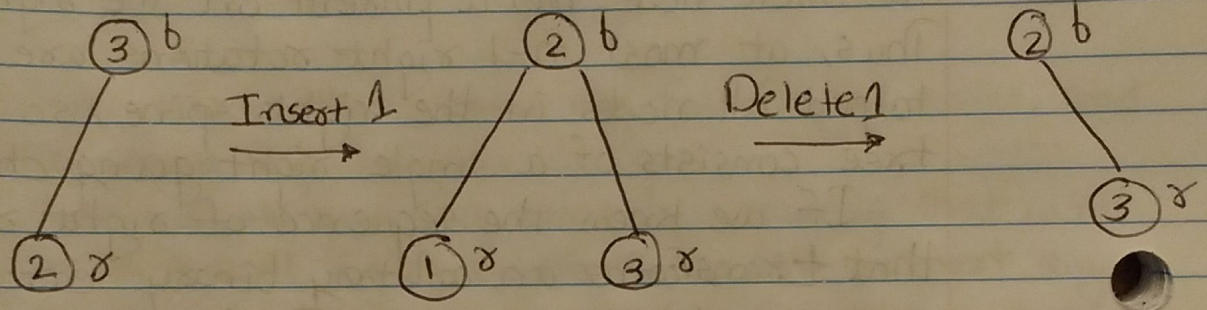
Observe that this right rotation adds x to the right spine, and no other nodes leave the right spine. Thus, this right rotation increases the number of nodes in the right spine by 1. Any binary search tree starts out with at least one node - the root in the right spine. Moreover, if there are any nodes not on the right spine, then at least one such node has a parent on the right spine. Thus, at most $n-1$ right rotations are needed to put all nodes in the right spine, so that the tree consists of a single right going chain.

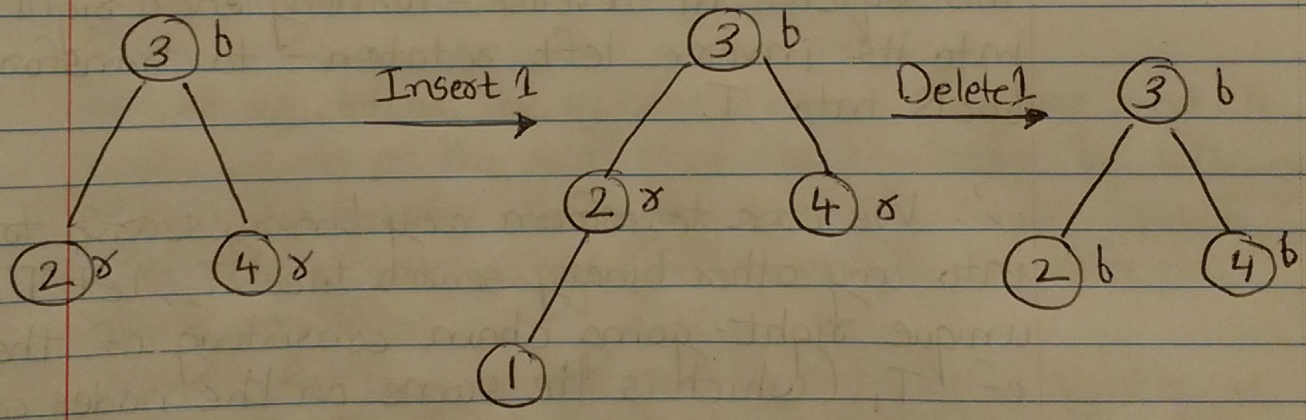
If we knew the sequence of right rotations that transforms an arbitrary binary search tree T to a single right-going chain T', then we could perform this sequence in reverse - turning each right rotation into its inverse left rotation - to transform T' back into T.

∴ We can transform any binary search tree T_1 into any other binary search tree T_2, Let T' be the unique right-going chain consisting of the nodes of T_1 (which is the same as the nodes of T_2). Let $\gamma = \langle \gamma_1, \gamma_2, \ldots \gamma_k \rangle$ be a sequence of ~~right-going chain consisting of the nodes of T_1 (which is the~~ right rotations that transform T_2 to T'. ~~We know~~ and let $\gamma' = \langle \gamma_1', \gamma_2' = \ldots \gamma_k' \rangle$ be a sequence of right rotations that transforms T_2 to T'. We know that these exist sequences γ and γ'

with $k, k' \leq n-1$. For each right rotation r_i, let l'_i be the corresponding inverse left rotation. Then the sequence $\langle r_1, r_2 \ldots r_k, l'_{k-1}, \ldots l'_2, l'_1 \rangle$ transforms T_1 to T_2 in at most $2n-2$ rotations.

(13.4.7) Inserting and Immediately deleting need not yield the same tree. Here is an example that alters the structure and one that changes the color





(15.1.1) We can verify that $T(n) = 2^n$ is solution to the given recurrence by the substitution method. We note that for $n = 0$, the formula is true since $2^0 = 1$. For $n > 0$, substituting into the recurrence and using the formula for summing geometric series yields

$$T(n) = 1 + \sum_{j=0}^{n-1} 2^j$$
$$= 1 + (2^n - 1)$$
$$= 2^n$$

(15.1.3) MODIFIED-CUT-ROD(p, n, c)

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = p[j]
    for i = 1 to j - 1
        q = max(q, p[i] + r[j - 1] - c)
    r[j] = q
return r[n]
```

The major modification required is in the body of the inner loop, which now reads $q = \max(q, p[i] + r[j-i] - c)$. This change reflects the fixed cost of making the cut, which is deducted from the revenue. We also have to handle the case in which we make no cuts (when i equals j); the total revenue in this case is simply $p[j]$. Thus, we modify the inner for loop to run from i to $j-1$ instead of to j. The assignment $q = p[j]$ takes care of the case of no cuts. If we did not make these modifications, then even in the case of no cuts, we would be deducting c from the total revenue.

(15.1.4) Modify MEMORIZED-CUT-ROD to return not only the value but the actual solution too.

```
MODIFIED-MEMOIZED-CUT-ROD(p,n)
let r[0..n] and s[0..n] be a new array
for i = 0 to n
  r[i] = -INF
return MODIFIED-MEMOIZED-CUT-ROD-AUX(p,n,r,s)

MODIFIED-MEMOIZED-CUT-ROD-AUX(p,n,r,s)
if r[n] >= 0
   return r[n]
if n == 0
   q = 0
else q = -INF
  for i = 1 to n
     if q < p[i] + MODIFIED-MEMOIZED-CUT-ROD-AUX(p,n-i,r,s)
     q = p[i] + MODIFIED-MEMOIZED-CUT-ROD-AUX(p,n-i,r,s)
     s[n] = i
r[n] = q
return q and s

PRINT-CUT-ROD-SOLUTION(p,n)
(r,s) = MODIFIED-CUT-ROD(p,n)
while n > 0
  print s[n]
  n = n - s[n]
```

(15.2) We assume that no word is longer than will fit into a line, i.e $l_i \leq M$ for all i.

First, we will make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a

sequence of words fits in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

→ Define $extras[i,j] = M - j + i - \sum_{k=i}^{j} l_k$ to be the number of extra spaces at the end of a line containing words i through j. (Extras may be negative)

→ Now define the cost of including a line containing words i through j in the sum we want to minimize

$$lc[i,j] = \begin{cases} \infty & \text{if } extras[i,j] < 0 \text{ (i.e. words } i, \ldots, j \text{ don't fit)}, \\ 0 & \text{if } j = n \text{ and } extras[i,j] \geq 0 \\ & \text{(last line costs 0)}, \\ (extras[i,j])^3 & \text{otherwise} \end{cases}$$

By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of lc over all lines of the paragraph.

Our subproblems are how to optimally arrange words $1, \ldots, j$ where $j = 1, \ldots, n$.

Consider an optimal arrangement of words $1, \ldots, j$

Suppose we know that the last line, which ends in word j, begins with word i. The preceding lines, therefore, contains words $1, \ldots, i-1$. In fact, they must contain an optimal arrangement of words $1, \ldots, i-1$

Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains word $i, \ldots, j$, then $c[j] = c[i-1] + lc[i,j]$
As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = lc[1,1]$ which is what we want.

We have to figure out which word begins the last line for the sub problems of words $1, \ldots, j$. So we try all possibilities for word i, and we pick the one that gives the lowest cost. Here, i ranges from 1 to j. Thus, we can define $c[j]$ recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \le i \le j} (c[i-1] + lc[i,j]) & \text{if } j > 0 \end{cases}$$

Note that the way we defined lc ensures that

→ All choices made will fit on the line (since an arrangement with $lc = \infty$ cannot be chosen as the minimum), and

→ The cost of putting words $i, \ldots, j$ on the last line will not be ∞ unless this really is the last line of the paragraph ($j = n$) or words $i \ldots j$ fill

the entire line.

We can compute a table of c values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel p table that points to where each c value came from. When $c[j]$ is computed if $c[j]$ is based on the value of $c[k-1]$, set $p[j]=k$ Then after $c[n]$ is computed, We can trace the pointers to see where to break the lines. The last line starts at word $p[n]$ and goes through word n. The previous line starts at word $p[p[n]]$ and goes through word $p[n]-1$ etc.

Pseudocode

```
PRINT-NEATLY(l, n, M)
▷ Compute extras[i,j] for 1≤i≤j≤n
 for i ← 1 to n
     do extras[i,i] ← M-l_i
       for j ← i+1 to n
       do extras[i,j] ← extras[i,j-1] - l_j - 1
▷ Compute lc[i,j] for 1≤i≤j≤n
  for i ← 1 to n
     do for j ← 1 to n
          do if extras[i,j] < 0
            then lc[i,j] ← ∞
          else if j = n and extras[i,j] ≠ 0
                then lc[i,j] ← 0
```

```
        else lc[i,j] ← (extras[i,j])^3
▷ Compute c[j] and p[j] for 1 ≤ j ≤ n
c[0] ← 0
for j ← 1 to n
    do c[j] ← ∞
       for i ← 1 to j
           do if c[i-1] + lc[i,j] ≤ c[j]
              then c[j] ← c[i-1] + lc[i,j]
                   p[j] ← i
return c and p
```

Quite clearly, both the time and space are $\Theta(n^2)$

In fact, we can get both the time and space down to $\Theta(nM)$. The key observation is that at most $\lceil M/2 \rceil$ words can fit on a line. (Each word is at least one character long, and there's a space between words.) Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i,j] = \infty$. We need only compute and store $extras[i,j]$ and $lc[i,j]$ for $j - i + 1 \le \lceil M/2 \rceil$. And the inner for loop header in the computation of $c[j]$ and $p[j]$ can run from $\max(1, j - \lceil M/2 \rceil + 1)$ to j.

We can reduce the space even further to $\Theta(n)$. We do so by not storing the lc and extras tables, and instead computing the value of $lc[i,j]$ as needed in the last loop. The idea is that we could compute $lc[i,j]$ in $O(1)$ time if we knew the value of $extras[i,j]$. And if we scan for minimum value in descending order of i, we can compute that as $extras[i,j] = extras[i+1,j] - l_i - 1$ (Initially, $extras[j,j]$

$= M - l_j$). This improvement reduces the space to $\Theta(n)$ since now the only tables we store c and p.

This is how we print which words are on which line. The printed output of GIVE-LINES(p, j) is a sequence of triples (k, i, j) indicating that words $i, \ldots, j$ are printed on line k. The return value is the line number k.

```
GIVE-LINES (p, j)
i ← p[j]
if i = 1
 then k ← 1
 else k ← GIVE-LINES (p, i-1) + 1
 print (k, i, j)
 return k
```

The initial call is GIVE-LINES(p, n). Since the value of j decreases in each recursive call, GIVE-LINES takes a total of $O(n)$ time.