University of Waterloo
CS240 Winter 2018
Assignment 3 – Solutions

version:
2018-03-07 18:16

Problem 1 [5+3=8 marks]

Let T be a binary tree with a set of n nodes in which each node has a pointer to its parent node. Choosing two arbitrary nodes u and v from the tree, we execute the following algorithm:

Algorithm `Nameless` (u, v)

```
1: P_u, P_v ← empty stack of references to nodes
2: do
3:       P_u.push(u)
4:       u ← u.parent
5: while (u != null )
6: do
7:       P_v.push(v)
8:       v ← v.parent
9: while (v != null )
10: create a new node c
11: while (!P_u.isEmpty() and !P_v.isEmpty() and P_u.top()=P_v.top()) do
12:       c ← P_u.top()
13:       P_u.pop()
14:       P_v.pop()
15: return (c)
```

y47bai

a) Describe what `Nameless` returns.

Solution: In the `Nameless` (u, v) algorithm, we use two stacks P_u and P_v to maintain the path from nodes u and v to the root of the tree, respectively. In line 10, at the top of both stacks you will necessarily find a reference to the root of the tree. Then, we extract an element from both stacks simultaneously; the last equal node extracted from both will be the lowest common ancestor (LCA). It is necessary to pay attention to the fact that, during this process, one of the two stacks could be empty (in particular this happens if one of the two nodes is ancestor of the other, and therefore it is itself the LCA of both).

This document is for the exclusive use of y47bai.

b) Analyze the run-time of `Nameless` in both worst case and best case using asymptotic notation. Briefly justify your answer.

Solution: The time complexity of `Nameless` (u, v) is dominated by the time necessary to fill the stacks, which in the worst case corresponds to the maximum height of the tree. Since in the worst case a tree of n nodes has height $n-1$ (the tree degenerates into a chain), the cost of the algorithm in the worst case is $O(n)$. The best case occurs when one of the two nodes is the root and the other is one of its children; in this case the cost will be $O(1)$. In the case that $u = v$, the best case is $u = v = root$.

Problem 2 [4+4+6=14 marks]

I. Consider the following AVL trees T_1 and T_2,

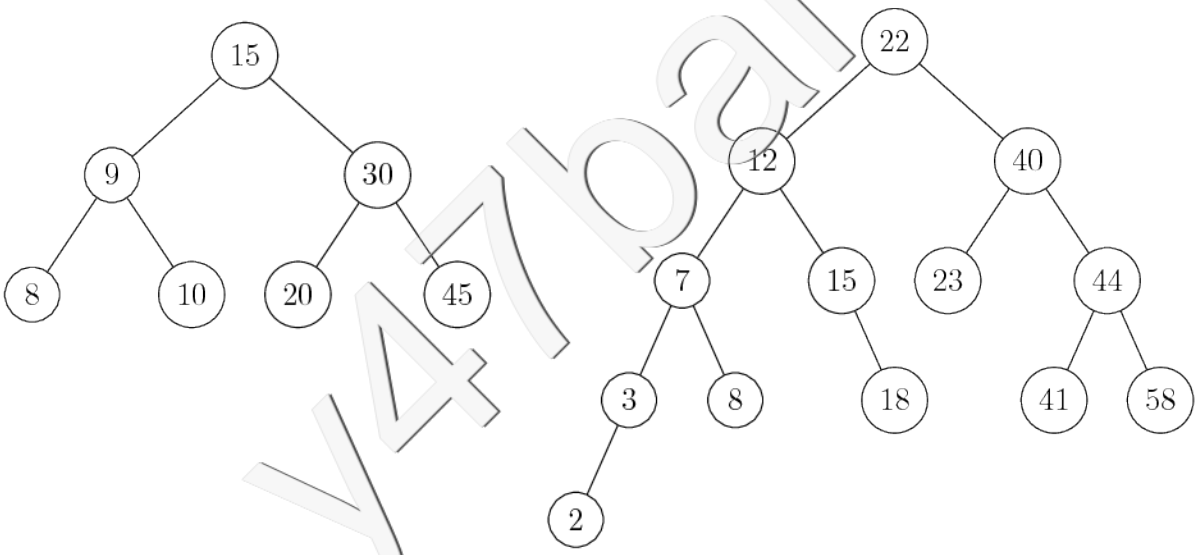


Figure 1: T_1 and T_2

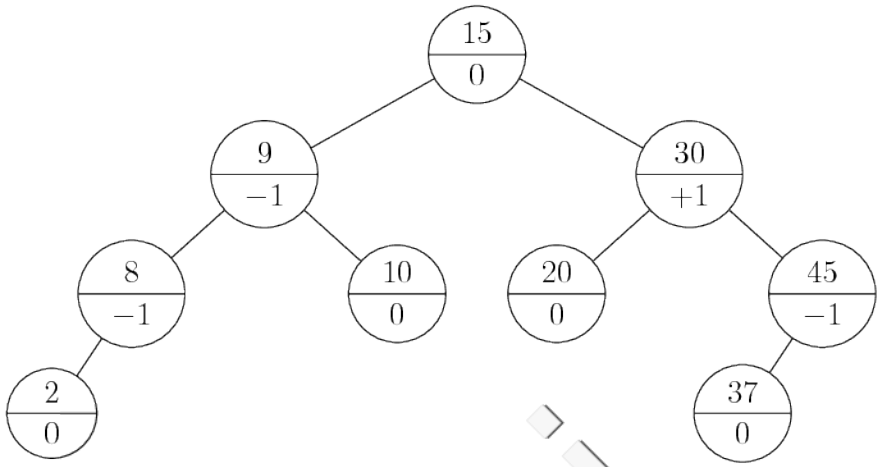
y47bai

a) Show the result of inserting 2, 37, 7, 12, 3, 36 into the tree T_1. Draw the tree before and after every insertion that results in rebalancing (similar to slide 25 of Module 4).

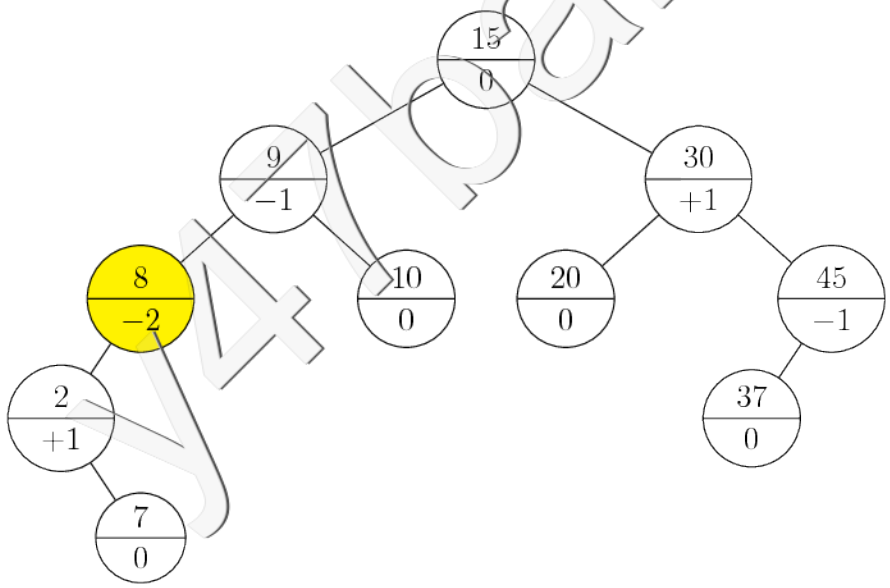
This document is for the exclusive use of y47bai.

Solution:

a1) After inserting 2, 37



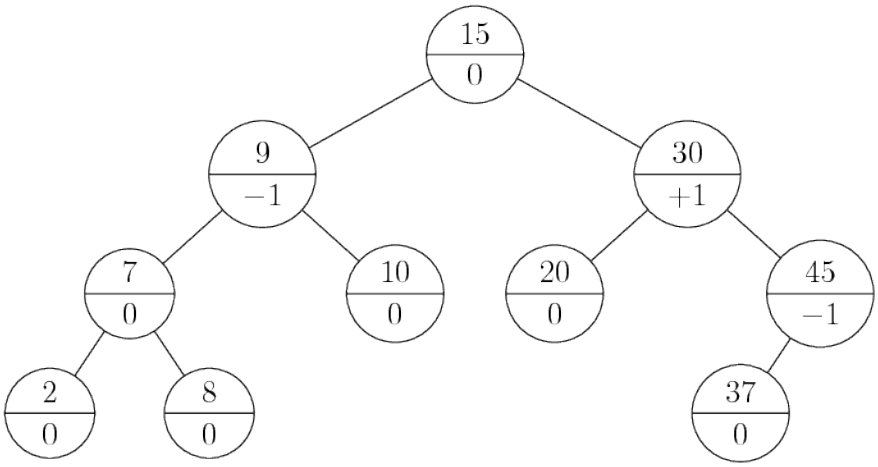
a2) After inserting 7



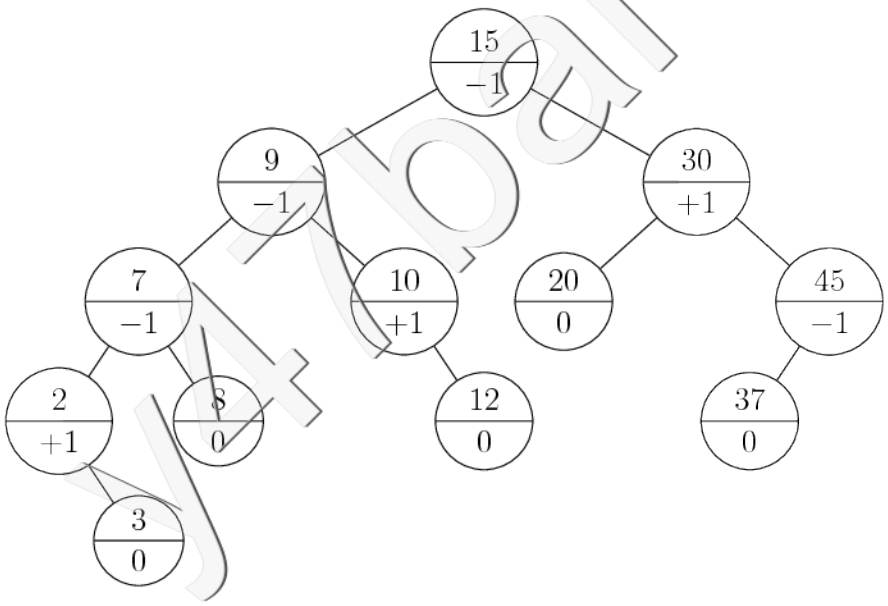
y47bai

This document is for the exclusive use of y47bai.

a3) Double rotation: left rotation in 2, right in 8



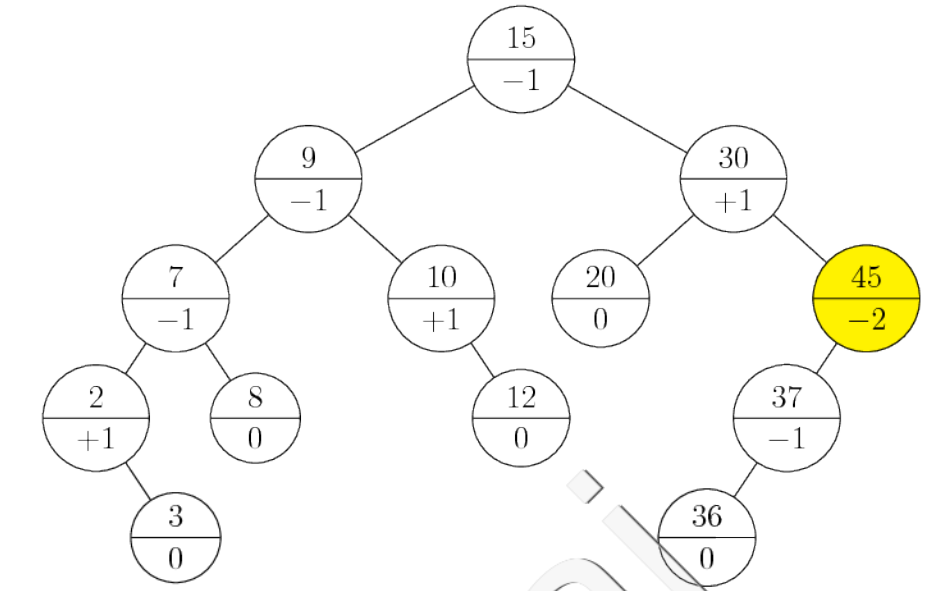
a4) After inserting 12, 3



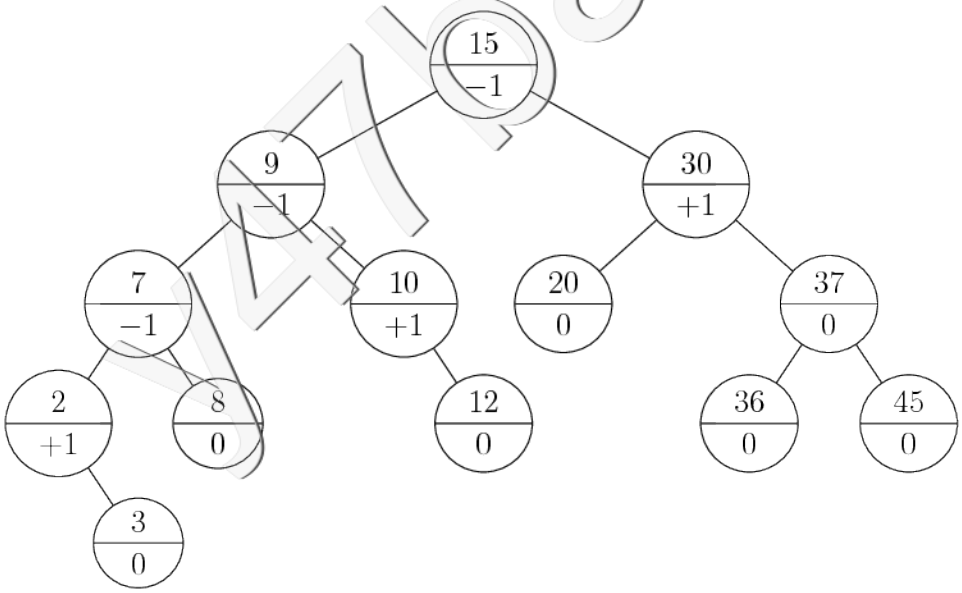
y47bai

This document is for the exclusive use of y47bai.

a5) After inserting 36



a6) Rotation: right rotation in 45

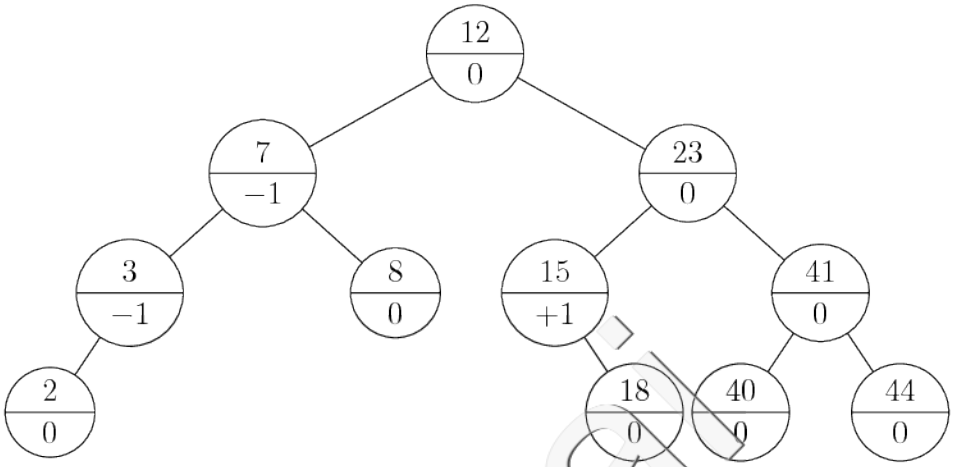


y47bai

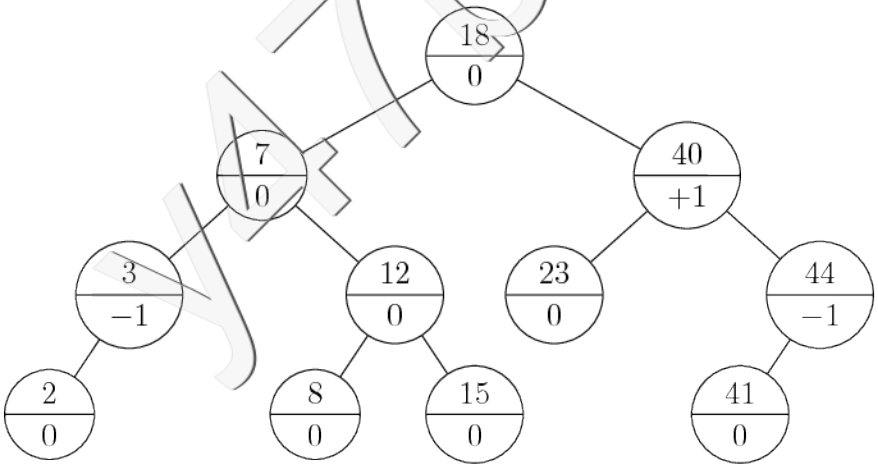
This document is for the exclusive use of y47bai.

b) Draw your final AVL tree after removing the keys 22 and 58, in the order given, from the tree T_2.

First Solution: Replace the deleted node with its successor.



Second Solution: Replace the deleted node with its predecessor.



This document is for the exclusive use of y47bai.

II. Consider an AVL tree containing n distinct keys. Design an algorithm to compute a function called `IncKey`(k, d) whose purpose is to increase the key k (if a node containing key k is present) by the given value $d \geq 0$ (we assume the key $k + d$ is unique in the tree). At the end, the resulting tree must still be AVL tree. Analyze the run-time of your algorithm using asymptotic notation. For full credit, the asymptotic run-time of your algorithm must be as small as possible.

Solution:

Idea. If the value d is such that $k + d$ has the same position of key k in the key ordering, then it is sufficient to increase the key k to $k + d$ and stop (lines 4-6). Otherwise, we can delete the node with key k and insert a node with key $k + d$, using the procedures for deleting and inserting nodes in AVL trees.

Any correct solution using simple *delete* and *re-insert* is acceptable.

Code.

Algorithm `IncKey` (k, d)

1: $r \leftarrow$ // reference to the root of AVL tree
2: $u \leftarrow r$ // r root node
3: $(u, P) \leftarrow$ `AVL-search`(r, k)
4: $u' \leftarrow$ `AVL-Minimum` $(u.right)$ // find the minimum key in the subtree rooted at $u.right$
5: **if** $k + d < u'.key$
6: $\quad u.key \leftarrow u.key + d$ // keys already respect the BST property
7: **else**
8: $\quad val \leftarrow u.value$
9: $\quad$ `AVL-delete2`(r, u)
10: $\quad$ `AVL-insert`$(r, k + d, val)$

Algorithm `AVL-delete2` (r, u)

1: $(r', P) \leftarrow$ `BST-delete2`(r, u)
2: The same as `AVL-delete` (Module 4).

Correctness.

`AVL-delete2`: unlike `AVL-delete`, which takes as argument the key to be deleted, this procedure takes as input the node containing the key to be deleted, thus saving the time for searching the key. Specifically, the two procedures differ just at line 1 of the pseudo code of `AVL-delete`: instead of calling the procedure `BST-delete(`r`,` k`)`, it calls `BST-delete2(`r`,` u`)`, which skips searching for the node containing the key, unlike `BST-delete(`r`,` k`)`. The algorithm is correct since the procedures it uses from slides (`BST-delete` and `AVL-delete`) are correct as well.

This document is for the exclusive use of y47bai.

Analysis.

The run-time of this algorithm is $O(\log n)$, where n is the number of nodes in the AVL tree, since all the procedures `AVL-Minimum`, `AVL-delete2`, `AVL-insert` have running time $O(\log n)$.

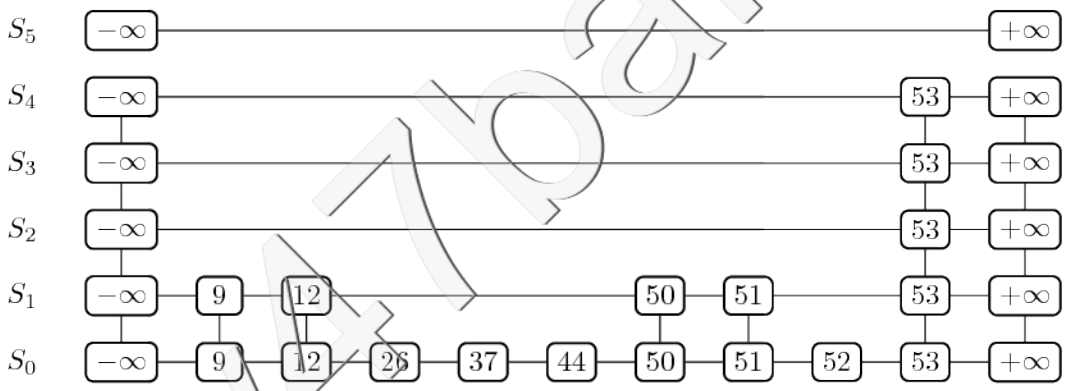
Problem 3 [6+4+8+2=20 marks]

I. Given an empty skip list L,

a) Show the result of inserting the following keys,

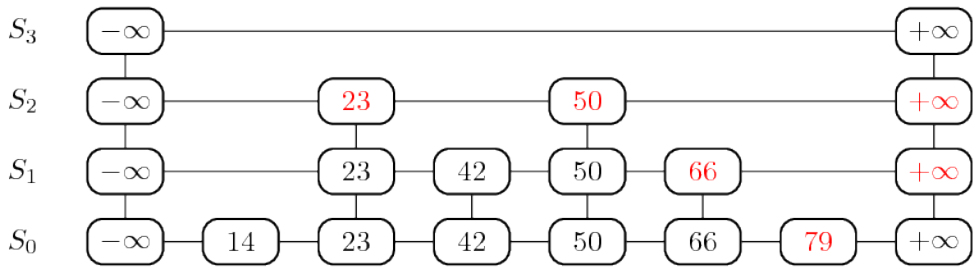
Keys: 44, 9, 26, 50, 12, 37, 51, 52, 53
Coin flip sequence: THTTHTHTTHTTHHHHT

Solution:



b) Count the number of key-comparisons needed to successfully search the key 79 in the following skip list and maintain the track of your searches in a stack called P (similar to slide 4 of Module 5).

Solution. The stack P contains: $-\infty$, 50, 66, 66, $top(P)$=66 and $after(66)$=79. 6 comparisons are required to search the 79 (Slide 4 of Module 5).



This document is for the exclusive use of y47bai.

II. Consider a skip list L that contains n distinct keys. Associated with each key x at each level i is a field $dist[x, i]$ containing the number of keys skipped by the pointer at x to the next element, including the key x itself. For example, in the above skip list the key 23 at level S_0 has $dist[23, 0] = 1$ and at level S_2 has $dist[23, 2] = 2$.

a) Design an algorithm in pseudocode which uses the minimum number of visited nodes (i.e., after/below calls) to compute $Select(L, k)$. The task of $Select(L, k)$ is to return the kth smallest key in the skip list L. If no such key exists, it returns *null*. For example in the above skip list, 14 is the 1st smallest key, 23 is the 2nd, etc. You may assume the skip list L has a variable $L.depth$ that stores the number of levels in L.

Solution.

Idea.

The idea is to use the field *dist* to know if we should continue the current level or we should go one level down for more refined search. Moreover, *dist* field can be used in the case in which we are at the last node of a level and we want to understand if k is greater than the number of nodes in skip list.

The original formulation of the problem uses $dist[x, i]$, where we use the key x and the current level i to obtain the *dist* value. This formulation was meant to emphasize the fact, that the distance entry is different on each level, but it is actually more convenient to store *dist* as part of the node objects (along with *after* and *below* pointers), so our suggested solution here uses the notation $dist(p)$ for a node p instead of $dist[x, i]$ (the same syntax is as for $after(p)$ and $below(p)$ used on the slides). By counting down from the topmost level (stored in $L.depth$), we can always maintain the information about the current level (as done in the code below for illustration), so both options for accessing *dist* are acceptable.

Code.

Algorithm `Select` (L, k)
L: a skip list, k

1: $p \leftarrow$ topmost left position
2: $r \leftarrow k$
3: $level \leftarrow L.depth - 2$
4: $p \leftarrow below(p)$
5: **while** $r > 0$
6: **if** $key(after(p)) = +\infty$ **and** $r - dist(p) \geq 0$ // $k > size(L)$
7: $p =$null
8: **break**
9: **if** $dist(p) \leq r$

This document is for the exclusive use of y47bai.

10: $r \leftarrow r - dist(p)$
11: $p \leftarrow after(p)$
12: **else**
13: $p \leftarrow below(p)$
14: $level \leftarrow level - 1$
15: **return** (p)

Correctness.
Suppose $k > n$, where n is the number of elements in the list. In this case, at line 10, r will be reduced according to the *dist* field of the current node (number of nodes skipped), until (iterating on the **while** loop) p points to the last element of that level and the current value of $r \geq dist(p)$, since $r > n$. The equality in $r \geq dist(p)$ holds just in the case $k = n + 1$, since, by definition of *dist*, $-\infty$ is also included in $dist(-\infty)$ at the first iteration of the **while** loop. Hence, condition at line 6 will be **true** and $p = null$ is returned.
If $k \leq n$, to show the correctness it is enough to show that if $r > 0$ in one **while** iteration, meaning that the searched key follows the current key in the S_0 level, it happens: 1) the algorithm will continue searching on the current level if $r > dist(p)$, and this is ensured by lines 9-11, after which $r > 0$ and the while loop condition still **true** (line 5); 2) the search will go to the level below if $r < dist(p)$, and this is ensured by lines 12-14; 3) if $r = dist(p)$, lines 9-11 set $r = 0$ and move to the next node of the same level (line 11), which is the searched node (by definition of *dist*), and stop because in next iteration of the while r is not positive, and p is returned.

Analysis. The running time is the same as *Search* in skip list, which is $O(\log n)$ expected time.

b) Describe the update operations that are necessary after insertion or deletion.

Solution. Update operations after inserting or deleting a *key* x:

- **Insert:** insert a new key x is as in skip list. During search, we compute the rank of all nodes on the stack. After inserting x, we increment *dist* for all nodes on the stack higher than the new tower i.e., those nodes whose after pointer now spans a new element. For the lower levels that point to the new tower, we use $dist(y) \leftarrow rank(x) - rank(y)$, where y is the node on the stack. The *dist* field of x is set to $dist(x) \leftarrow rank(y) + dist(y) + 1 - rank(x)$.
- **Delete:** delete a key x is as in skip list. During search, we compute the *rank* of all nodes on the stack. After deleting x, we decrement *dist* for all nodes on the stack higher than the old tower. For the lower levels that point to the old tower, we update the *dist* field as $dist(y) \leftarrow dist(y) + dist(x) - 1$, where

This document is for the exclusive use of y47bai.

y is the node on the stack.

Problem 4 [5+5+4=14 marks]

Suppose a linked list (or an array) with dynamic ordering contains the items $ABDCEFGH$,

a) Show a possible sequence of searches using the Move-To-Front (MTF) heuristic that leads to the sequence $EHGDABCF$.

Solution: A possible search sequence leading to list content $EHGDABCF$ is $DGHE$.

b) Using the Transpose heuristic, give a sequence of searches that leads to the sequence $ADBCEGHF$.

Solution: A possible search sequence leading to list content $ADBCEGHF$ is DGH.

c) Having the sequence of searches like $DHHGHEGH$, compute the total number of operations (key-pair comparison and swap) for both the Move-To-Front (MTF) and the Transpose heuristics.

Solution: We assume the linked list structure is not modified by both heuristics MTF and Transpose when an operation search is performed, that is node headers are not changed. This implies that to move to front the searched key (suppose found at node u), the list must be scanned again from the top and shift all the keys to next node till u, counting at each step one *swap* operation. For the Transpose heuristic instead just one swap operation is needed. Note this swapping operations (and so the second list scan for MTF) could be avoided by appropriately changing the node headers after finding the key in the list.

The *comparison* operations are those required to find the key in the list, thus if the key is at position i in the list, it requires i comparisons, for both heuristics.

Table 1 provides the corresponding result.

This document is for the exclusive use of y47bai.

Table 1: Proposed solution to Problem P4 c).

MTF	**Cost**	**Transpose**	**Cost**	**Key**
$DABCEFGH$	5 (3 Comp. + 2 swap)	$ADBCEFGH$	4 (3 Comp. + 1 swap)	D
$HDABCEFG$	15 (8 + 7)	$ADBCEFHG$	9 (8 + 1)	H
$HDABCEFG$	1 (1 + 0)	$ADBCEHFG$	8 (7 + 1)	H
$GHDABCEF$	15 (8 + 7)	$ADBCEHGF$	9 (8 + 1)	G
$HGDABCEF$	3 (2 + 1)	$ADBCHEGF$	7 (6 + 1)	H
$EHGDABCF$	13 (7 + 6)	$ADBCEHGF$	7 (6 + 1)	E
$GEHDABCF$	5 (3 + 2)	$ADBCFGHF$	8 (7 + 1)	G
$HGEDABCF$	5 (3 + 2)	$ADBCFHGF$	8 (7 + 1)	H
Total	62	**Total**	60	

Problem 5 [2+3+5+10=20 marks]

Consider the following ordered array of size $n = 10$,

$$1\ 5\ 9\ 15\ 21\ 30\ 35\ 46\ 47\ 50$$

and the pseudocode of the Interpolation Search algorithm given below.

```
InterpolationSearch(A[l, r], k)
A: an array
l: index of the left boundary
r: index of the right boundary
k: key to search for
 1.  if A[l] > k  ||  A[r] < k then
 2.      return false
 3.  if l = r then
 4.      if A[l] = k then
 5.          return true
 6.      else
 7.          return false
 8.  i ← l + ⌊(r − l) (k−A[l]) / (A[r]−A[l])⌋
 9.  if A[i] = k then
10.      return true
11.  else if A[i] < k then
12.      return InterpolationSearch(A[i + 1, r], k)
13.  else
14.      return InterpolationSearch(A[l, i − 1], k)
```


This document is for the exclusive use of y47bai.

a) Apply the `InterpolationSearch` algorithm to search for the keys 5 and 46, showing the behavior of the algorithm until the key is found.

Solution: Searching for key 5:

- `InterpolationSearch`$(A[0,9],5)$, $i=0$, $A[0]=1\neq 5$,
- `InterpolationSearch`$(A[1,9],5)$, $i=1$, $A[1]=5\neq 5$, stop.

Searching for key 46:

- `InterpolationSearch`$(A[0,9],46)$, $i=8$, $A[8]=47\neq 46$,
- `InterpolationSearch`$(A[0,7],46)$, $i=7$, $A[7]=46$, stop.

b) Determine the number of key equality comparisons (lines 4 and 9) needed in both cases and discuss whether it is consistent with the expected complexity.

Solution: In both cases the number of key equality comparisons is 2 (see the solution of point **a)**), and specifically one comparison at line 9 in both the first and second calls of the procedure. The array A contains 10 keys in the range $[1,50]$, which are roughly equispaced in the interval. In such case, that is when keys are distributed in the interval, the expected asymptotic complexity of the procedure is $\Theta(\log\log n)$, and 2 calls are in line with it.

c) Provide an example of an ordered array that contains $n=10$ distinct keys, including keys 5 and 46, such that the `InterpolationSearch` algorithm performs at least $n/2$ key equality comparisons for searching both keys 5 and 46. For full credit, exactly $n/2$ key equality comparisons in both searches are required. Show the behavior of the pseudocode on your solution (for each recursive call to `InterpolationSearch` show the values of $l, r, i, A[i]$ after the calculation at line 8).

Solution. Let us consider the following ordered array B

1 2 3 4 5 46 47 48 49 350

Searching for key 5:

- `InterpolationSearch`$(B[0,9],5)$, $i=0$, $B[0]=1<5$,
- `InterpolationSearch`$(B[1,9],5)$, $i=1$, $B[1]=2<5$,
- `InterpolationSearch`$(B[2,9],5)$, $i=2$, $B[2]=3<5$,
- `InterpolationSearch`$(B[3,9],5)$, $i=3$, $B[3]=4<5$,
- `InterpolationSearch`$(B[4,9],5)$, $i=4$, $B[4]=5$, stop.

At each call of the procedure `InterpolationSearch` one key equality comparison is carried out (line 9), thus obtaining $5=\frac{n}{2}$ comparisons.

Searching for key 46:

This document is for the exclusive use of y47bai.

- `InterpolationSearch`$(B[0,9],46)$, $i=1$, $B[1]=2<46$,
- `InterpolationSearch`$(B[2,9],46)$, $i=2$, $B[2]=3<46$,
- `InterpolationSearch`$(B[3,9],46)$, $i=3$, $B[3]=4<46$,
- `InterpolationSearch`$(B[4,9],46)$, $i=4$, $B[4]=5<46$,
- `InterpolationSearch`$(B[5,9],5)$, $i=5$, $B[5]=46$, stop.

Again the total number of key equality comparisons is $5=\frac{n}{2}$. We point out that 350 is not the only value suitable to have both key searches taking $\frac{n}{2}$ comparisons. In principle, any value such that the first computed index i at line 8 is 1, and all the indexes computed at line 8 in the next recursive calls are 0 can be adopted. Moreover, as it is obvious, values 47, 48, 49 in the provided solution are not relevant: any triple of ordered values in the interval $[47,349]$ is suitable.

d) Generalize part c) to an arbitrary n. That is, given n, explain how to construct an ordered array with n distinct keys, where there exists at least one key whose search through the `InterpolationSearch` algorithm has time complexity $\Omega(n)$.

Solution: Given an asymptotically large n, let us consider the following ordered vector:

$$A=(1,2,\ldots,n-1,2^n)$$

Suppose the key to search is $k=n-1$.

- At step 1 the call is `InterpolationSearch`$(A[l,r],k)=$ `InterpolationSearch`$(A[0,n-1],n-1)$. Then the algorithm at line 8 computes the index

$$\begin{aligned} i &= 1+\left\lfloor \frac{k-A[1]}{A[n]-A[1]}(n-1)\right\rfloor = 0+\left\lfloor \frac{n-2}{2^n-1}(n-1)\right\rfloor \\ &= 0+\left\lfloor \frac{n^2-3n+2}{2^n-1}\right\rfloor = 0 \end{aligned}$$

  The last equality holds since for $n\geq 1$ we have $0\leq \frac{n^2-3n+2}{2^n-1}<1$. $A[0]\neq k$ and the recursive call is `InterpolationSearch`$(A[1,n-1],n-1)$

- Step 2, is `InterpolationSearch`$(A[1,n-1],n-1)$. Again, the algorithm computes the index

$$i = 1+\left\lfloor \frac{n-3}{2^n-2}(n-2)\right\rfloor = 1+\left\lfloor \frac{n^2-5n+6}{2^n-2}\right\rfloor \overset{n\geq 2}{=} 1$$

  $A[1]\neq k$ and the recursive call is `InterpolationSearch`$(A[2,n-1],n-1)$

This document is for the exclusive use of y47bai.

- By iterating, at step j, with $1 \leq j < n-1$, the recursive call is `InterpolationSearch`$(A[j-1, n-1], n-1)$, leading to the selection of following index

$$\begin{aligned} i &= j - 1 + \left\lfloor \frac{n-j-1}{2^n - j}(n-j) \right\rfloor \\ &= j - 1 + \left\lfloor \frac{n^2 - (2j+1)n + j(j+1)}{2^n - j} \right\rfloor \overset{n \geq j}{=} j - 1 \end{aligned}$$

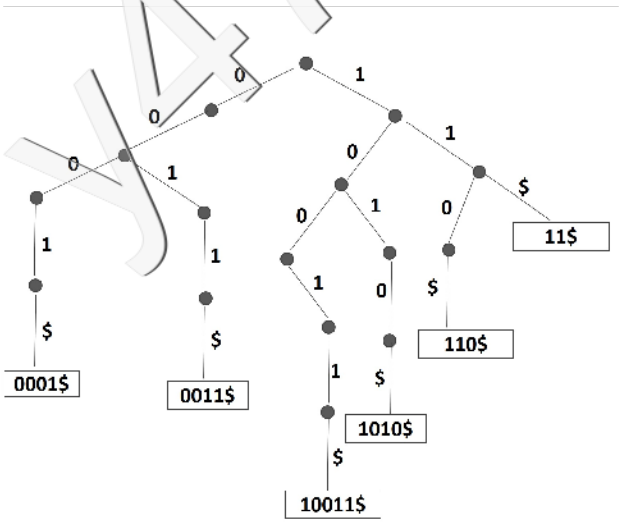
$A[j-1] \neq k$ and the recursive call is `InterpolationSearch`$(A[j, n-1], n-1)$.

Thus, at step $n-1$ the recursive call is `InterpolationSearch`$(A[n-2, n-1], n-1)$, which leads to the index $i = n-2$, and at line 9 the procedure ends since $A[n-2] = k$. The total number of steps is thereby $n-1$, thus the running time of `InterpolationSearch`$(A[0, n-1], n-1)$ is $\Omega(n)$.

Problem 6 [3+3+8=14 marks]

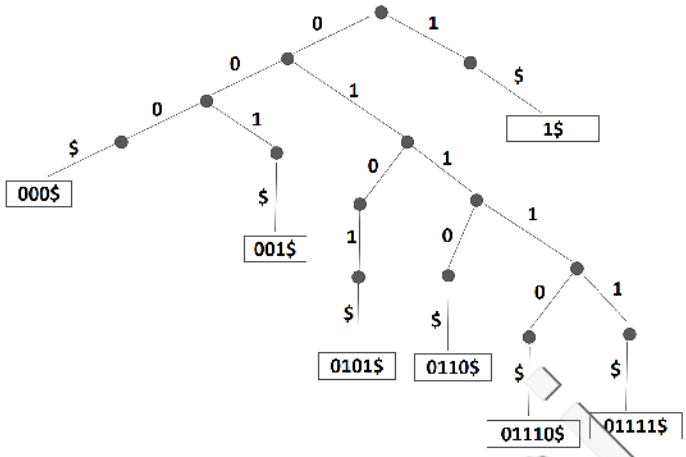
a) Draw your final trie after inserting binary keys $S = \{0001\$, 0011\$, 1010\$, 11\$, 110\$, 10011\$\}$ into an initially empty (uncompressed) binary trie.

Solution: Trie after inserting binary keys $S = \{0001\$, 0011\$, 1010\$, 11\$, 110\$, 10011\$\}$:

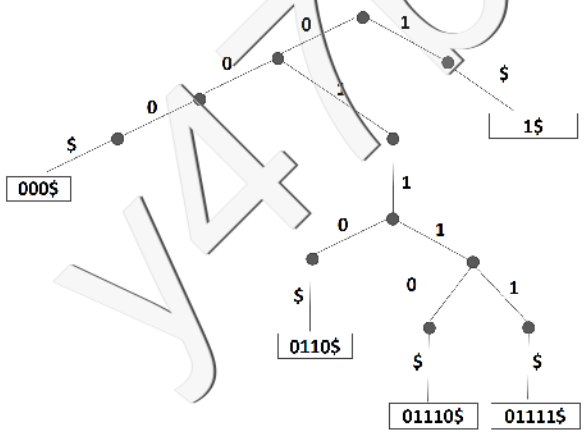


This document is for the exclusive use of y47bai.

b) Draw your final trie after removing the keys 0101$, 001$, in the order given, from the following trie.



Solution: Trie after removing the keys 0101$ and 001$:



y47bai

This document is for the exclusive use of y47bai.

c) Consider a trie T that contains n strings. Design an algorithm called $Look(T, x, l)$, which returns the list of all strings with length l (without \$) that begin with a given string x of length m, where $m \leq l$. Assume x does not end with \$. The worst-case running time of your algorithm should be $O(m + 2^{l-m+1})$.

Solution:

Idea.

The algorithm before searches for x, then explores the corresponding subtrie until the depth $l - m$, adding to the list L all strings having exactly length $l - m$ in the subtrie.

Code.

The proposed pseudocode for $Look(T, x, l)$ is as below.

Algorithm `Look`(T, x, l)

```
1: L ← empty list
2: m ← x.length
3: if m > 0
4:       for i ← 0 to m − 1 do
5:          if x[i] = 0
6:             if T.left ≠ null
7:                T ← T.left
8:             else
9:                return(L)
10:         else
11:            if T.right ≠ null
12:               T ← T.right
13:            else
14:               return(L)
15: return Trie-visit (T, l, L, m)
```

`Trie-visit` $(T,\ d,\ L,\ depth)$ visits the (sub)trie T and returns the list L filled by all the strings of depth d (excluding \$). The argument *depth* is the depth of the root node in T. In general, given a trie T, the depth of the root node is 0, and the depth of a node is the depth of its parent +1. To simplify the code, we assume that the binary string corresponding to a node u in the trie is contained in the field $u.value$.

Algorithm `Trie-visit`$(T,\ d,\ L,\ depth)$

```
1: if T ≠ NULL
2:       if T.left ≠ NULL or T.right ≠ NULL
3:             if depth = d // the string is longer than d
4:                return(L)
5:             else
```


y47bai

This document is for the exclusive use of y47bai.

```
6:              L ← Trie-visit (T.left, d, L, depth + 1)
7:              L ← Trie-visit (T.right, d, L, depth + 1)
8:      else
9:          if depth = d // the string has length d
10:             L.Add (T.value)
11: return L
```

Correctness. The algorithm before has to check if x is the empty string (line 3), and in such case it goes to line 15 (trie visit); otherwise, in lines 4-14 it searches for x in the trie T. If x is not present in T, lines 9 and 14 ensure an empty list is returned; otherwise, at the end of **for** loop, T refers to the node in the trie corresponding to x, and line 15 visits it. If x was the empty string, since in this case x is prefix of all strings, the algorithm will visit the whole initial trie T. The visit (procedure `Trie-visit`) keeps the depth of the current node, thus avoiding exploring paths longer than $l - m$ (d in the pseudocode of `Trie-visit`). Indeed, if a string corresponds to a path longer than d, the visit of that node stops, and no string is added to L (lines 3-4). If the string is long exactly d, then line 9 and 10 add it to L. Finally, if the current node corresponds to a string shorter than d and the next character is not \$, a recursive call explores the remaining subtrie by increasing the depth on 1 (lines 6 and 7). At the end of this visit, all the subtrie until depth d has been visited, but just strings with length d have been added to L.

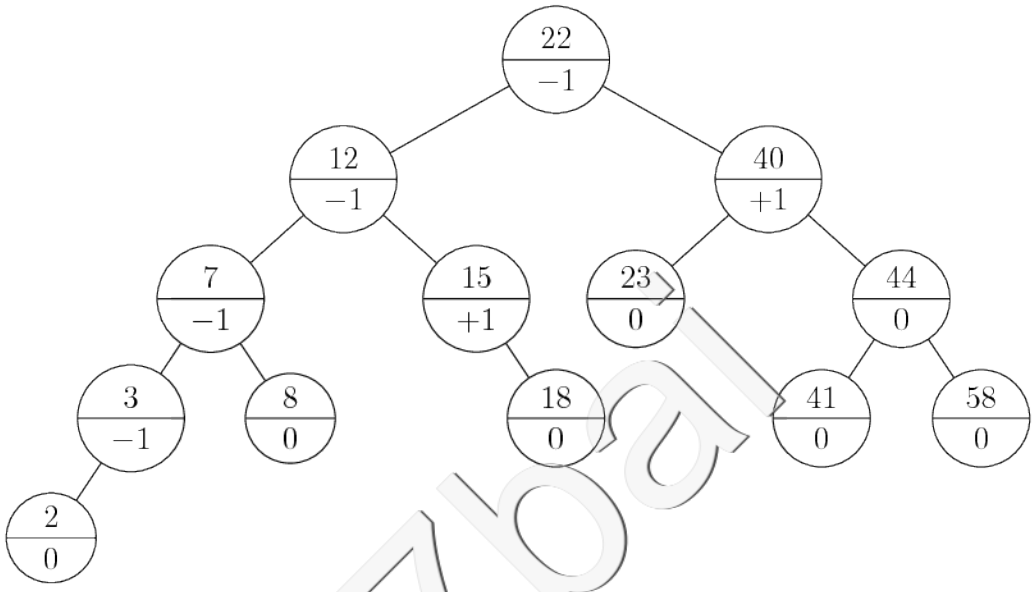
Analysis. The procedure `Trie-visit`(T, d, L, *depth*) visits the subtrie rooted at T until depth d, starting from depth *depth*. A binary trie with h levels contains at most $2^{h+1} - 1$ nodes, when the trie is full. Accordingly, since each node is visited just once, and nodes with depth $> d$ are not visited, the overall complexity is $O(2^{d-depth+1})$. The procedure `Look(T,x,l)` searches for x in the trie, which takes $O(m)$, and calls the procedure `Trie-visit` with depth l, thus its time complexity is $O(m + 2^{l-m+1})$.

This document is for the exclusive use of y47bai.

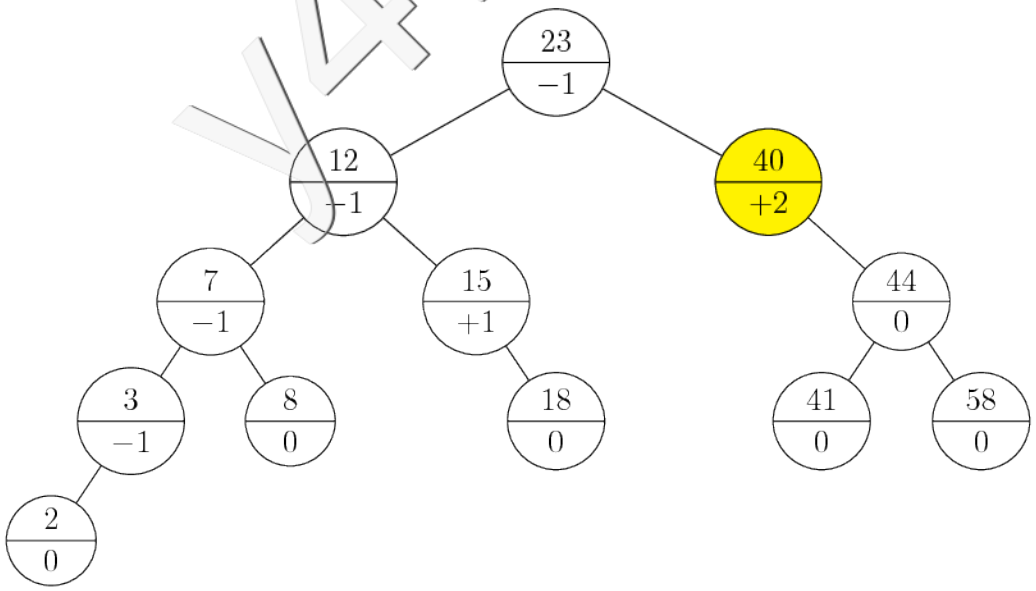
Problem 2b: First solution: Replace the deleted node with its successor.

Steps:

- T_2 with balancing factor



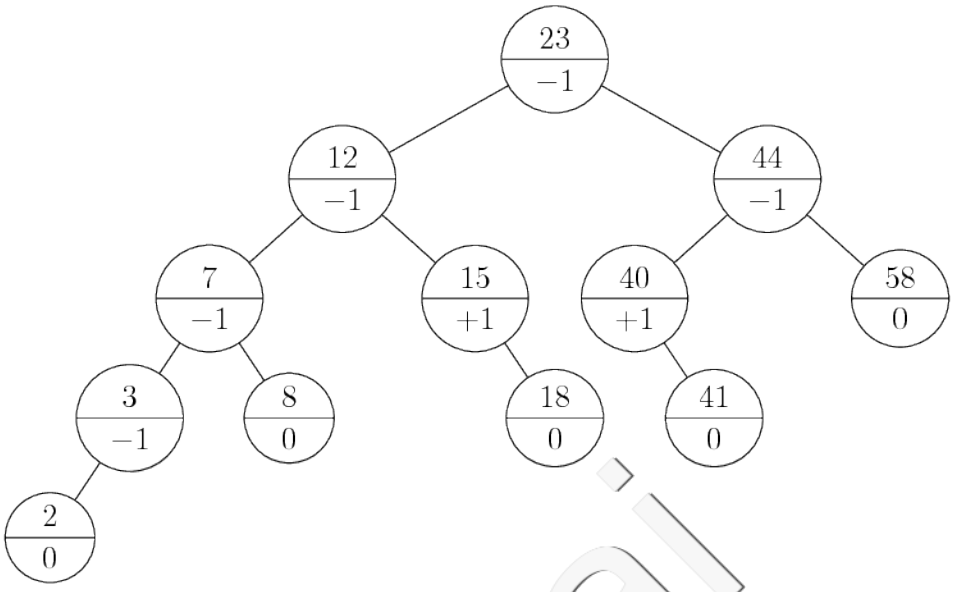
- After deleting 22



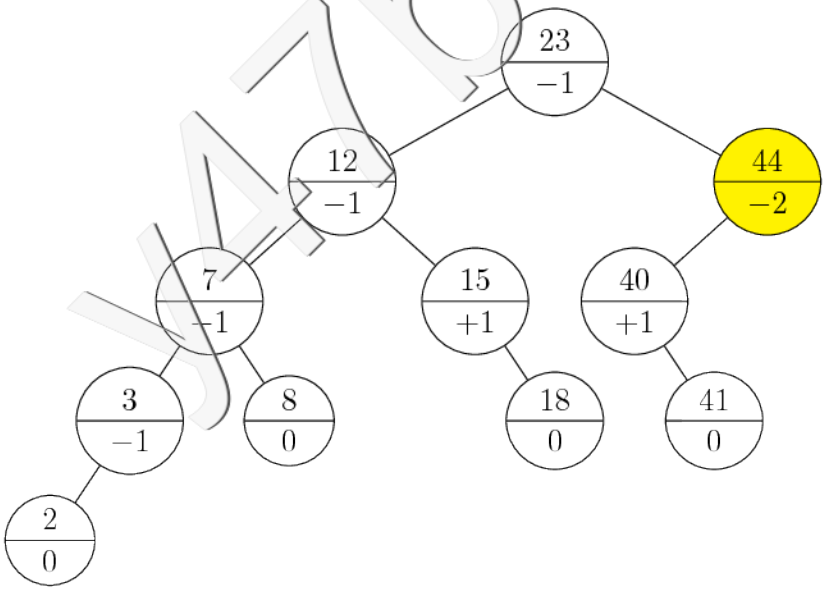
y47bai

This document is for the exclusive use of y47bai.

- Rotation: left rotation in 40



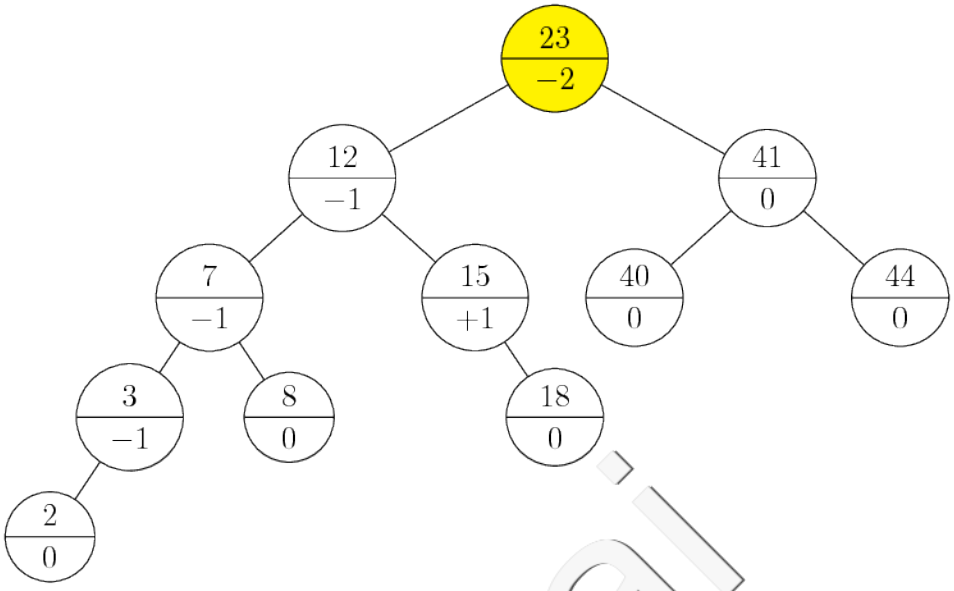
- After deleting 58



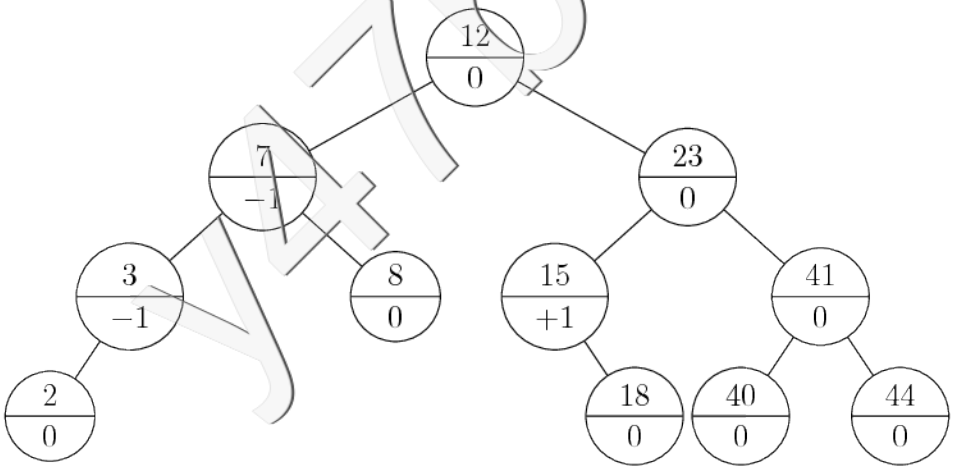
y47bai

This document is for the exclusive use of y47bai.

- Double rotation- left rotation in 40, right rotation in 44



- Rotation: right rotation in 23



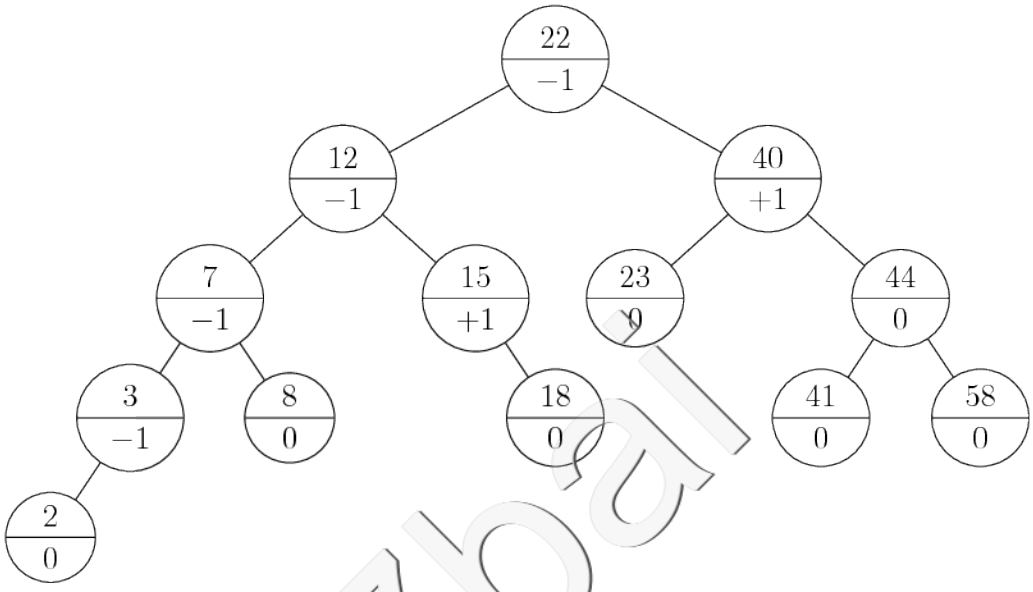
y47bai

This document is for the exclusive use of y47bai.

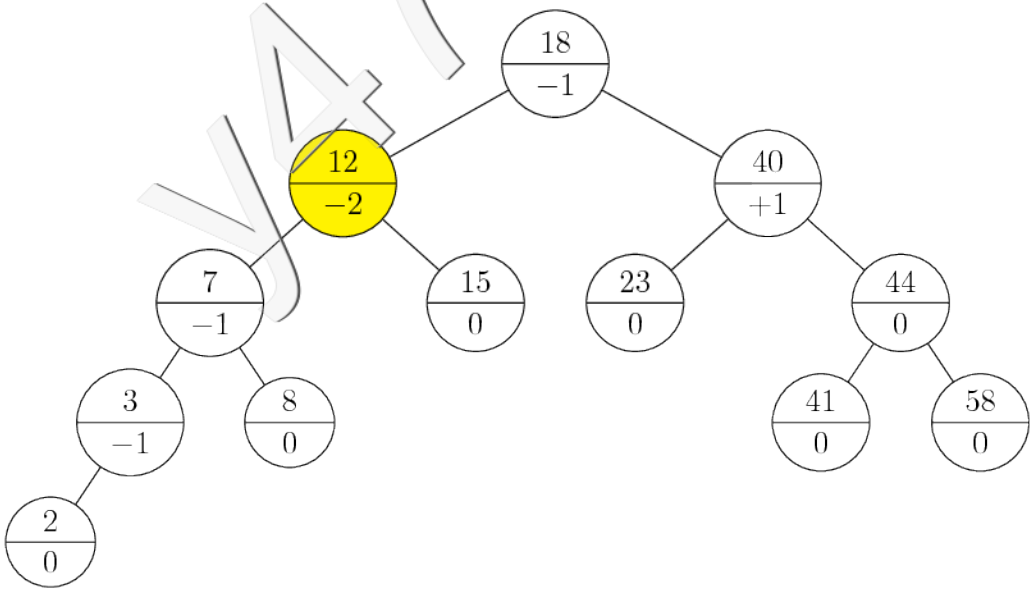
Problem 2b: Second solution: Replace the deleted node with its predecessor.

Steps:

- T_2 with balancing factor

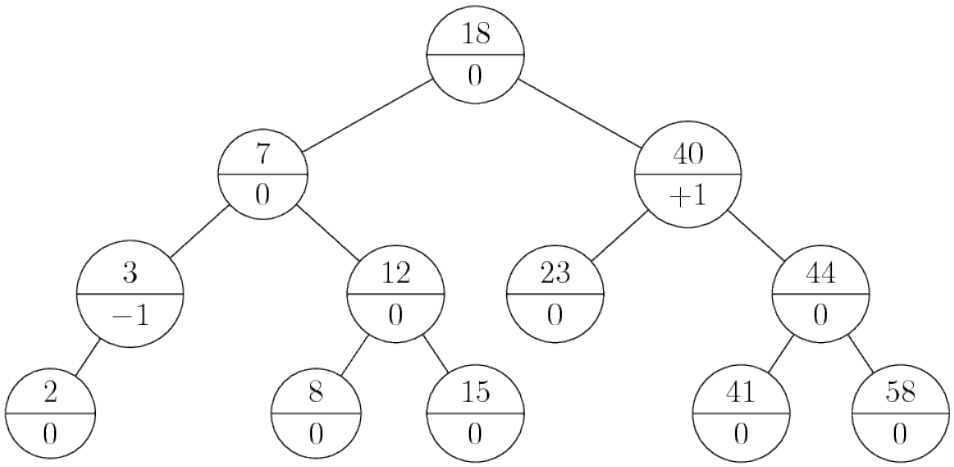


- After deleting 22: (subsituationfg with predecessor)

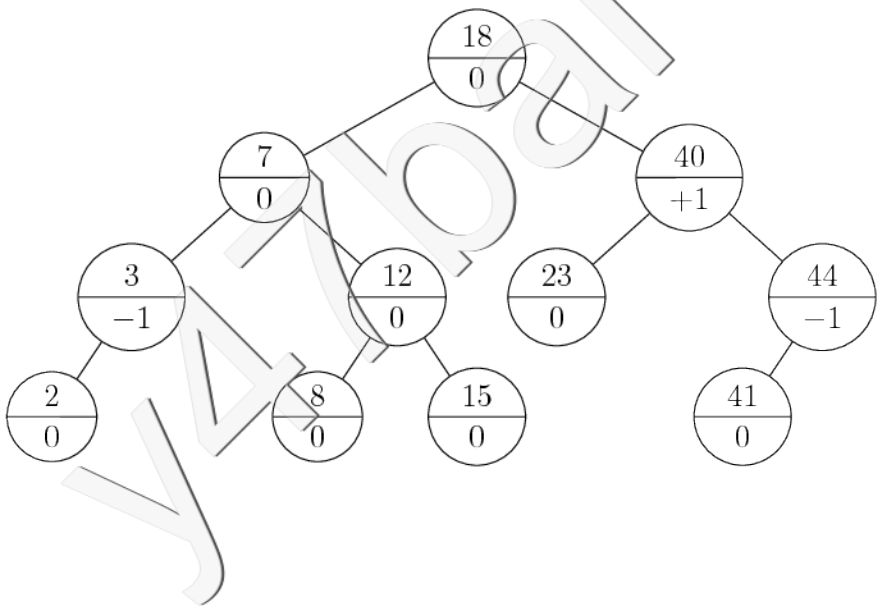


This document is for the exclusive use of y47bai.

- Rotation: right rotation in 12



- After deleting 58



y47bai

This document is for the exclusive use of y47bai.