DSA Assignment №2 Klyushev Vsevolod BS20-02

1 task

1)

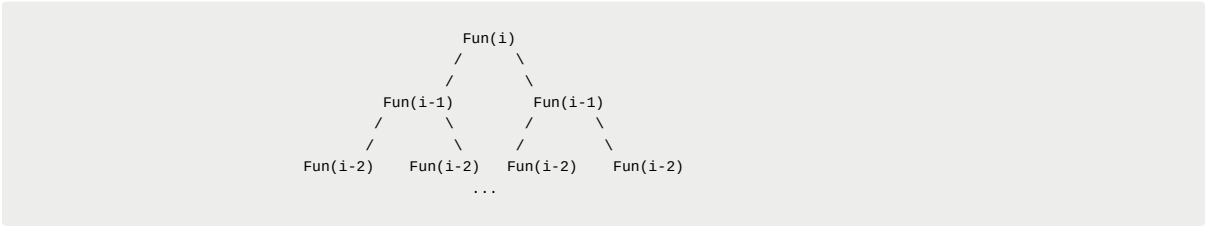
```
//in all my pseudocodes array[n] starts from 1 to n, but we can also store something in array[0]
//this initial task is of a type of scheduling problem, so applying ideas are the same.
1)==========================================================
input n, x[n], h, r[n], d; //in all arrays values starts from 1
//n - number of shoping centers
//x[i] - starting coordinate of i'th shoping center
//r[i] - estimated revenue of i'th city
//h - the highway length
//d - restriction

//represent shoping centers as range for using same ideas as in scheduling problem
xRange[n][2];

for (int j=1; j<=n; j++){ //asymptotic worst case time complexity O(n)
  xRange[j][1]=x[j];
  xRange[j][2]=x[j]+d;
}

p[n]  //there is an index of the closest possible neighbour center that satisfying the condition
      //that there is a restriction that no two shopping centres can be placed
      //within d kilometres of each other

//binary search
last(int i){ //asymptotic worst case time complexity O(log(n))
    int left = 1, right = i;
    while (left <= right){
      int mid = (left + right) / 2;
      if (xRange[mid][2] <= xRange[i][1]) {
        if (xRange[mid + 1][2] <= xRange[i][1])
          left = mid + 1;
        else
          return mid;
      }
      else
        right = mid - 1;
    }
    return 0; //No compatible neighbour was found. Return 0. (no closest previous neighbout)
}

//p array
for (int j=1; j<=n; j++){ //asymptotic worst case time complexity O(n*log(n))
  p[j]=last(j);
}

//count maximum estimated revenue for i`th city
Fun(j) {  //Worst case happens when we can put all the shopping centers one after the other,
          //therefore for all i | p[i]=i-1
          //Asymptotic worst case time complexity O(2^(n)) since we can represent it as a tree with 2 branches.
          //We will have j levels starting from 0 level =>2^(j+1)-1 elements, where element = call
  if (j==0)
    return 0;
  else
    return max(r[j]+Fun(p[j]),Fun(j-1));
}

Run Fun(n);
```

2)Asymptotic worst case time complexity is $O(2^n)$ - because it's the worst time complexity from the previous functions from subtask 1.

- loop for xRange array has asymptotic worst case time complexity $O(n)$
- last() function has asymptotic worst case time complexity $O(log(n))$
- loop for p array has asymptotic worst case time complexity $O(n * log(n))$ because of n calls of last() function
- Fun() function :Asymptotic worst case time complexity $O(2^n)$ since we can represent it as a tree with 2 branches. We will have j levels starting from 0 level ⇒ $2^{j+1} - 1$ elements, where element = call of function
  Worst case happens when we can put all the shopping centers one after the other, therefore for all i | p[i]=i-1

3)In tree representation there are overlapping problems because we call function Fun() for same arguments several times.

For example for Fun(i) in worst case we will call Fun(i-1) - 2 times, Fun(i-2) - 4 times, Fun(i-3) - 8 times, etc.



4)

```
//in all my pseudocodes array[n] starts from 1 to n, but we can also store something in array[0]
//this initial task is of a type of scheduling problem, so applying ideas are the same.
input n, x[n], h, r[n], d //in all arrays values starts from 1
//n - number of shoping centers
//x[i] - starting coordinate of i'th shoping center
//r[i] - estimated revenue of i'th city
//h - the highway length
//d - restriction

//represent shoping centers as range for using same ideas as in scheduling problem
xRange[n][2];
for (int j=1; j<=n; j++){ //asymptotic worst case time complexity O(n)
  xRange[j][1]=x[j];
  xRange[j][2]=x[j]+d;
}

p[n]  //there is an index of the closest possible neighbour that satisfying the condition
      //that there is a restriction that no two shopping centres can be placed
      //within d kilometres of each other

//binary search
last(int i){ //asymptotic worst case time complexity O(log(n))
    int left = 1, right = i;
    while (left <= right){
      int mid = (left + right) / 2;
      if (xRange[mid][2] <= xRange[i][1]) {
        if (xRange[mid + 1][2] <= xRange[i][1])
          left = mid + 1;
        else
          return mid;
      }
      else
        right = mid - 1;
    }
    return 0; //No compatible neighbour was found. Return 0. (no closest previous neighbout)
}

//p array
for (int j=1; j<=n; j++){ //asymptotic worst case time complexity O(n*log(n)) because of n calls of last()
  p[j]=last(j)
}

//count maximum estimated revenue for i first cities
int Fun[n] // array for elements from 0 to n
Fun[0]=0;
for (int j=1; j<=n; j++){ //asymptotic worst case time complexity O(n)
  Fun[i]=max(r[j]+Fun[p[j]],Fun[j-1]);
}

findSolution(j) { //asymptotic worst case time complexity O(n) because
                  //number of recursive calls in our function ≤ n ⇒ O(n).
  if (j = 0) {
    output nothing;
  } else if (r[j] + Fun[p[j]] > Fun[j-1]) {
    print j;
    findSolution(p[j]);
  } else {
    findSolution(j-1);
  }
}

Run findSolution(n);
```

5) Asymptotic worst case time complexity is $O(n * log(n))$ - because it's the worst time complexity from the previous functions from subtask 4.

- loop for xRange array has asymptotic worst case time complexity $O(n)$
- last() function has asymptotic worst case time complexity $O(log(n))$
- loop for p array has asymptotic worst case time complexity $O(n * log(n))$ because of n calls of last() function
- loop for Fun array has asymptotic worst case time complexity $O(n)$
- findSolution() function has asymptotic worst case time complexity $O(n)$
  because number of recursive calls in our function ≤ n ⇒ O(n).

2 task

1)Starting from the 2nd element we pick an element, compare it with all elements in the sorted subarray locates before the considered element. Shift all the elements in the sorted subarray that is greater/lower (depending on the task) than the value we want to sort. Then insert the value. Go to the next element and repeat until the array is sorted.

2)

```
//It is pseudocode implementation of insertion sort for an array A with n elemnts starting from index 1
//insertion sort(A)                                          cost                    time                        time in best case
for j=2 to A.length                                          c1                      n                                 n
  key=A[j]                                                   c2                      n-1                               n-1
  //insert A[j] into sorted sequence A[1..j-1]               0                       n-1                               n-1
  i=j-1                                                      c3                      n-1                               n-1
  while A[i]>key and i>0                                     c4                      n(n+1)/2-1                        n-1
    A[i+1]=A[i]                                              c5                      n(n-1)/2                           0
    i=i-1                                                    c6                      n(n-1)/2                           0
  A[i+1]=key                                                 c7                       n-1                               n-1
```

$T(n) = c1 * n + c2 * (n-1) + c3 * (n-1) + c4 * (n * (n+1)/2 - 1) + c5 * (n * (n-1)/2) + c6 * (n * (n-1)/2) + c7(n-1) = a * n^2 + b * n + c$

$a * n^2/2 \leq a * n^2 + b * n + c \leq (a+b+c) * n^2 \implies \Theta(n^2)$ for any n≥1

Therefore, worst case time complexity would be $\Theta(n^2)$

Best case time complexity would be for an already sorted array (in order that we need) since while loop in sorting won't work because of condition A[i]>key (always false in that case). Therefore best time complexity would be $\Theta(n)$

3)K-sorted array - an array of n elements, where each element is at most k away from its target position.

Therefore each while loop would work k time at worst case. That mean, that $T(n) = c1 * n + c2 * (n-1) + c3 * (n-1) + c4 * (n-1) * (k+1) + c5(n-1) * k + c6 * (n-1) * k + c7(n-1) = a * n * k + b * n + c$

$a * n * k/2 \leq a * n * k + b * n + c \leq (a+b+c) * n * k \implies T(n) = \Theta(n * k)$ for any n≥1

Therefore, worst case time complexity would be $\Theta(n * k)$

4)

```
//We can represent table as array of objects Line.
//We also consider that we have such type as data as default type (as int, char, double, etc.), so comparisons for types are given
class Line(string code, date date_start, date date_end, string sponsor, string descriptoin)

//So firstly we compare 2 elements by 1st char symbol from codes, if they are equal we compare by their integer parts of code,
//if they are equal we compare elements by start date, if they are equal we compare by the end date, and if they are equal again,
//we compare by sponsor name using a hash function that converts a string into integer hash code.
bool moreThan(Line l1, Line l2) {
  if (l1.code[0]>l2.code[0]) {
  | return true;
  } else if (l1.code[0]==l2.code[0]) {
  | if (toInteger(l1.code,l1.code.begin+1,l1.code.end) > toInteger(l2.code,l2.code.begin+1,l2.code.end)) {
  | | return true;
  | } else if (toInteger(l1.code,l1.code.begin+1,l1.code.end) == toInteger(l2.code,l2.code.begin+1,l2.code.end)) {
  | | if (l1.date_start > l2.date_start) {
  | | | return true;
  | | } else if (l1.date_start == l2.date_start) {
  | | | if (l1.date_end > l2.date_end) {
  | | | | return true;
  | | | } else if (l1.date_end == l2.date_end) {
  | | | | int i=0;
  | | | | while (i<l1.sponsor.size() && i<l2.sponsor.size()) {
```

```
  | | | | | if (l1.sponsor[i]>l2.sponsor[i])
  | | | | | | return true
  | | | | | i++;
  | | | | }
  | | | | if (l1.sponsor.size>l2.sponsor.size) {
  | | | | | return true;
  | | | | }
  | | | }
  | | }
  | }
  }
  return false;
}

//using insertion sort we sort our table, that represented as an array of objects Line
main() {
  input Line A[n]
  for j=2 to A.length
  Line key = A[j]
  i=j-1
  while i>0 and moreThan(A[i],key)
    A[i+1]=A[i]
    i=i-1
  A[i+1]=key
  output A[n]
}
```

5)

recursive version

```
//arrays starts from 1
bool belongs(from, to, x, A[]) {
  if (from>to) {
    return false;
  }
  if (from==to) {
    if (x==A[from]) {
      return true;
    } else {
      return false;
    }
  }
  int mid=(from+to)/2;
  if (x<A[mid]) {
    return belongs(from, mid, x, A[]);
  } else if (x>A[mid] {
    return belongs(mid, to, x, A[]);
  } else {
    return true;
  }
}
```

iterative version

```
//arrays starts from 1
bool belongs(from, to, x, A[]) {
  while (from<=to) {
    int mid=(from+to)/2;
    if (A[mid]==x)
      return true;
    if (A[mid]<x) {
      from=mid+1;
    } else {
      to=mid-1;
    }
  }
  return false;
}
```

6)

recursive version

```
//arrays starts from 1
bool search(from, to, x, A[]) {
  if (from>to) {
    return null;
```

```
  }
  if (from==to) {
    if (x==A[from]) {
      return from;
    } else {
      return null;
    }
  }
  int mid=(from+to)/2;
  if (x<A[mid]) {
    return search(from, mid, x, A[]);
  } else if (x>A[mid] {
    return search(mid, to, x, A[]);
  } else {
    return mid;
  }
}
```

iterative version

```
//arrays starts from 1
int search(from, to, x, A[]) {
  while (from<to) {
    int mid=(from+to)/2;
    if (A[mid]==x)
      return mid;
    if (A[mid]<x) {
      from=mid+1;
    } else {
      to=mid-1;
    }
  }
  return null;
}
```

7)

$T(n) = a * T(n/b) + f(n)$

$T(n) = T(n/2) + const$

a=1 b=2 f(n)=const

$n^{log_b(a)} = n^{log_2(1)} = n^0 = 1$

$n^o = f(n) = const \implies$ 2nd case of Master theorem

Therefore $T(n) = \Theta(n^{log_b a} * log(n)) = \Theta(log(n))$

8)

If it's a search function from the 6th task, then there is no sense to use it since it returns the index of an existing element in the sorted part of the array, but in our case, we'll receive null all the time because our element x, for which we are looking for a place, isn't in the sorted subarray. If we change search that it would return index where we should put our element, then there would be some sense to do it.

```
//arrays starts from 1
bool new_search(from, to, x, A[]) {
  if (from>to) {
    return null;//wrong input
  }
  if (from==to) {
    if (x>=A[from]) {
      return from+1;
    } else {
      return from;
    }
  }
  int mid=(from+to)/2;
  if (x<A[mid]) {
    return new_search(from, mid, x, A[]);
  } else if (x>A[mid] {
    return new_search(mid, to, x, A[]);
  } else {
    return mid+1;
  }
}

//insertion sort(A) with search
```

```
main() {
  input Line A[n]
//--------------------------start of sorting part-----------cost--------------------time---------------------time in best case
  for j=2 to A.length                                        c1                    n                             n
    Line key = A[j]                                          c2                    n-1                           n-1
    index=new_search(1,j-1, key, A[])                        cAdd*log(n)           n-1                           n-1
    i=j-1                                                    c3                    n-1                           n-1
    while i>=index                                           c4new                 n(n+1)/2-1                    n-1
      A[i+1]=A[i]                                            c5                    n(n-1)/2                      0
      i=i-1                                                  c6                    n(n-1)/2                      0
    A[index]=key                                             c7                    n-1                           n-1
//--------------------------end of sorting part-------------------------------------------------
  output A[n]
}

=============================================================================================================================================
//just for reference
//insertion sort(A)                                       cost                    time                    time in best case
for j=2 to A.length                                       c1                      n                             n
  key=A[j]                                                c2                      n-1                           n-1
  //insert A[j] into sorted sequence A[1..j-1]            0                       n-1                           n-1
  i=j-1                                                   c3                      n-1                           n-1
  while A[i]>key and i>0                                  c4                      n(n+1)/2-1                    n-1
    A[i+1]=A[i]                                           c5                      n(n-1)/2                      0
    i=i-1                                                 c6                      n(n-1)/2                      0
  A[i+1]=key                                              c7                      n-1                           n-1
```

Asymptotic worst case time complexity will not change, however, it may improve the real time of working of an algorithm for an average and worst cases a little. So total time complexity of checking while condition was approximately
const* 2*n^2. Now total time complexity of checking while condition is approximately const*n^2 but we also have some additional operations from search() function const* n * log(n). For big n's, our new function is better. However, the best case would become worse, since while loop condition in best case time complexity approximately just const* (n), now in the best case while loop condition time complexity still const*(n) but our algorithm has additional const*log(n)*n (from binary search)

In other words, before we had

$$T(n) = c1 * n + c2 * (n-1) + c3 * (n-1) + c4 * (n * (n+1)/2 - 1) + c5 * (n * (n-1)/2) + c6 * (n * (n-1)/2) + c7(n-1) = a1 * n^2 + b * n + const1$$

and now we have

$$T(n) = c1 * n + c2 * (n-1) + cAdd * (n-1) * log(n) + c3 * (n-1) + c4new * (n * (n+1)/2 - 1) + c5 * (n * (n-1)/2) + c6 * (n * (n-1)/2) + c7(n-1) = a2 * n^2 + d * n * log(n) + b * n + e * log(n) + const2$$

So, since c4New is < c4, because of only one compare operation instead of 2, for big n's we will have that c4*(n*(n+1)/2-1)>c4new*(n*(n+1)/2-1)+cAdd*(n-1)*log(n) , so in general
$a1 * n^2 + b * n + const1 > a2 * n^2 + d * n * log(n) + b * n + e * log(n) + const2$, which means that it is more profitable to use new sorting

However, best case of our previous implementation of insertion sort would be better, because

$$c4new * (n-1) + cAdd * (n-1) * log(n) > c4(n-1)$$

Moreover, asymptotically best case of usual insertion sort was $\Theta(n)$, and now best case of new sort would have $\Theta(n * log(n))$

9)

Bubble sort is difficult to parallelize because we need to come up with an idea of how to separate process on parts and be sure that they'll work correctly on our array.

```
//Applying the well-known idea of odd-even parallel sorting we use 2 types of threads.
//First type sorts odd pairs and second one sorts even.
//That parallelization making the performance of our program faster.
//we'll use n/2 threads of first type and n/2 of second type

void parallelSort(int array[], int n)
{
    bool sorted = false;
    int n = array.size(); //length of the array to sort
    Thread[] thread1 = new Thread[n/2];
    Thread[] thread2 = new Thread[n/2];
```

```
    int index=0;
    while (!isSorted)
    {
        sorted = true;
        //start bubble sort on odd pairs
        thread1[index]:
        for (int i=1; i<=n-2; i=i+2)
        {
            if (array[i] > array[i+1])
            {
                swap(array[i], array[i+1]);
                sorted = false;
            }
        }
        //start bubble sort on even pairs
        thread2[index]:
        for (int i=0; i<=n-2; i=i+2)
        {
            if (array[i] > array[i+1])
            {
                swap(array[i], array[i+1]);
                sorted = false;
            }
        }
        index++
    }
}
```

3 task

1)

AVL trees have the following properties:

- Balance condition | Balance Factor (value) = height(left-subtree) – height(right-subtree) | if Balance Factor ∈ {-1,0,1} , then tree is balanced.
- AVL tree use rotations to its elements in order to balance itself.

```
//T1, T2, T3 and T4 are subtrees.

       z                                      y
      / \                                   /   \
     y   T4     Right Rotate (z)          x      z
    / \         ----------------->      /  \    /  \
   x   T3                             T1  T2  T3  T4
  / \
T1   T2

  z                                       y
 /  \                                   /   \
T1   y          Left Rotate(z)         z      x
    /  \        ----------------->     / \    / \
   T2   x                             T1  T2 T3  T4
       / \
     T3  T4
```

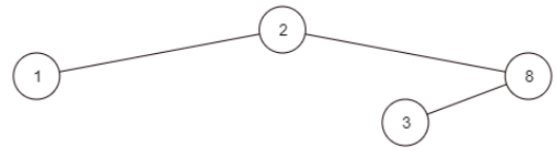
2)

- Step 1: Insert an element into the tree using Binary Search Tree insertion logic
- Step 2: Check the Balance Factor from bottom to root for all ancestors of the added node
- Step 3: If the Balance Factor of every node of ancestors of the added node is -1, 0 or 1 that means that our tree is balanced and we can start the next operation of the program.
- Step 4: Otherwise, it means that there is a balance factor that is not equal to -1, 0, or 1, so starting from the bottom to up for all ancestors of the added node until root we need to check the Balance Factor and if it's not satisfied the Balance Factor's condition, perform the following operations to make the tree balanced. After that, if the condition from step 3 holds, we can proceed to the next operation of the program, otherwise, repeat step 4.

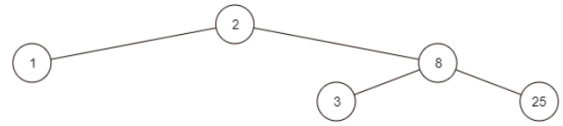
```
//Following operations
//Balance Factor (BF)
//let z be a current node
if BF(z) > 1
  if BF(y) >= 0 //y is a left child of z
    right-rotate(z)
```

```
  else
    left-rotate(y)
    right-rotate(z)
else if BF(z) <-1
  if BF(y) <= 0 //y is a right child of z
    left-rotate(z)
  else
    right-rotate(y)
    left-rotate(z)
```

3)25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55, 30, 15

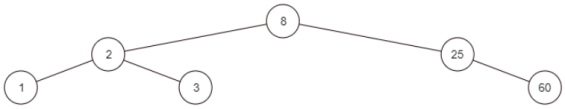


initial tree



inserted 25

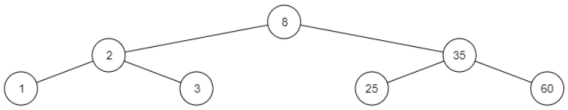
tree is balanced ⇒ no rotate



inserted 60

node with value 2 was unbalanced, so we have done:

right rotate for 2

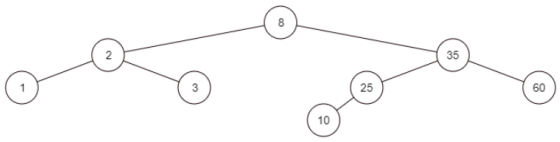


inserted 35

node with value 25 was unbalanced, so we have done:

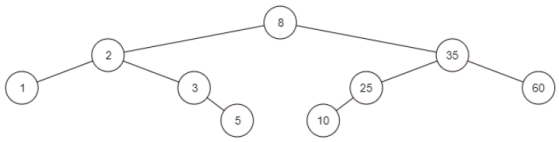
right rotate for 60

left rotate for 25



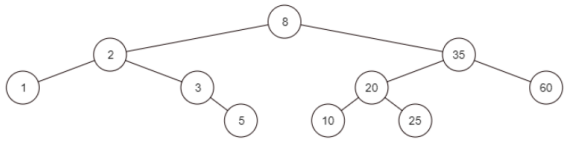
inserted 10

tree is balanced ⇒ no rotate



inserted 5

tree is balanced ⇒ no rotate

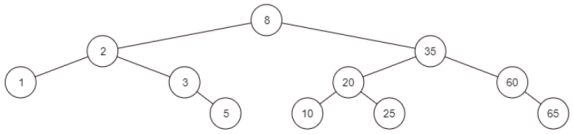


inserted 20

node with value 25 was unbalanced, so we have done:

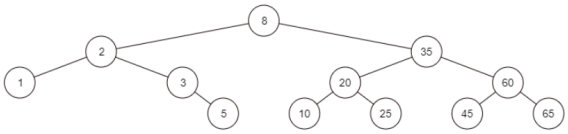
left rotate 10

right rotate 25



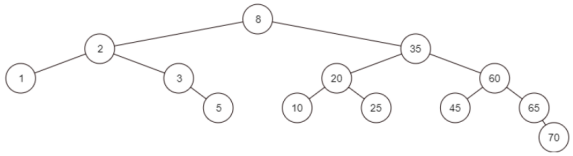
inserted 65

tree is balanced ⇒ no rotate



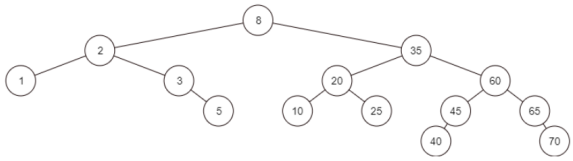
inserted 45

tree is balanced ⇒ no rotate



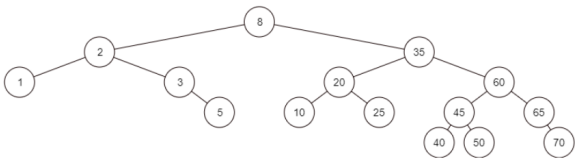
inserted 70

tree is balanced ⇒ no rotate



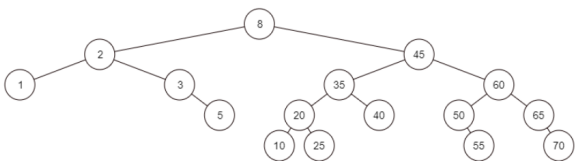
inserted 40

tree is balanced ⇒ no rotate



inserted 50

tree is balanced ⇒ no rotate

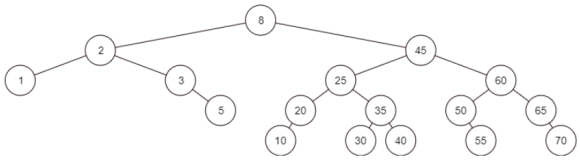


inserted 55

node with value 35 was unbalanced, so we have done:

right rotate 60

left rotate 35

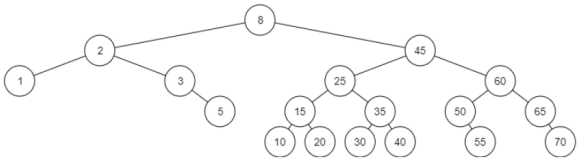


inserted 30

node with value 35 was unbalanced, so we have done::

left rotate 20

right rotate 35



inserted 15

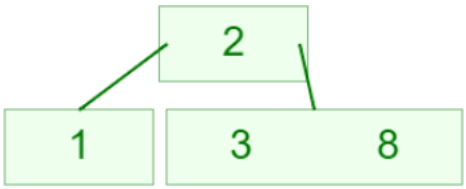
node with value 20 was unbalanced, so we have done:

left rotate 10

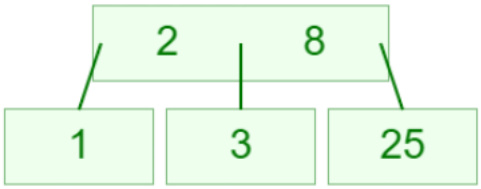
right rotate 20

4) 1,2,3,5,8,10,15,20,25,30,35,40,45,50,55,60,65,70 - sorted sequence in increasing order. And it's always work.

5)



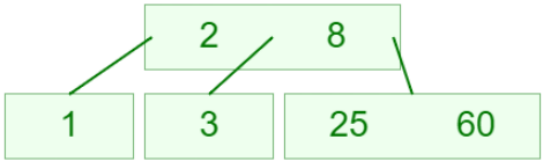
initial balanced 2-3 tree



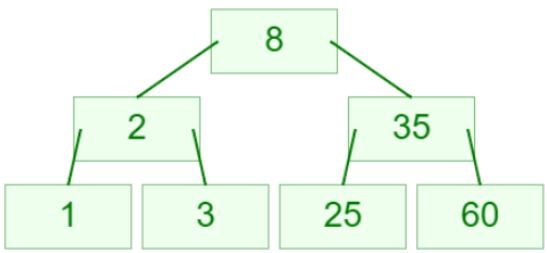
inserted 25 at bottom level

done:

8 pushed up from <3,8,25>



inserted 60 at bottom level

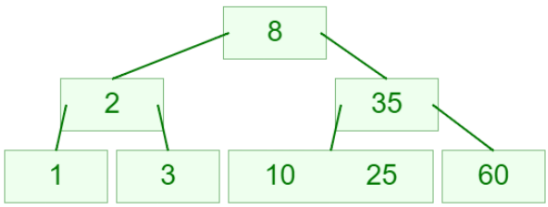


inserted 35 at bottom level

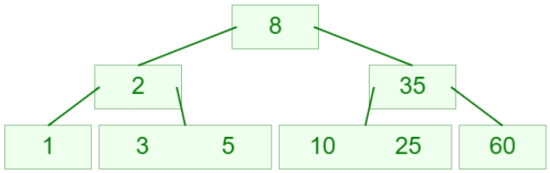
done:

35 pushed up from <25,35,60>

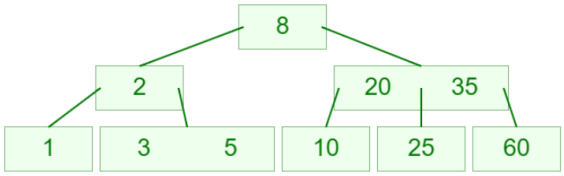
8 pushed up from <2,8,35>



inserted 10 at bottom level



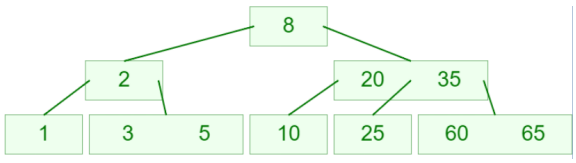
inserted 5 at bottom level



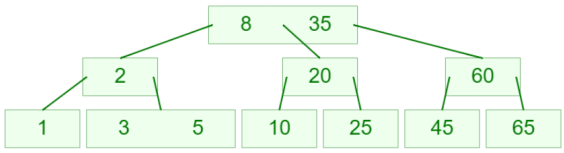
inserted 20 at bottom level

done:

20 pushed up from <10,20,25>



inserted 65 at bottom level

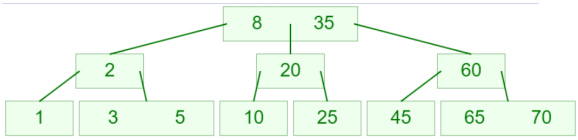


inserted 45 at bottom level

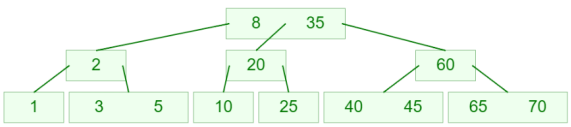
done:

60 pushed up from <45,60,65>

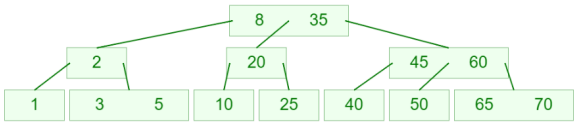
35 pushed up from <20,35,60>



inserted 70 at bottom level



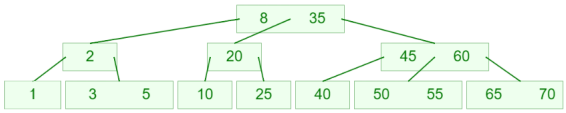
inserted 40 at bottom level



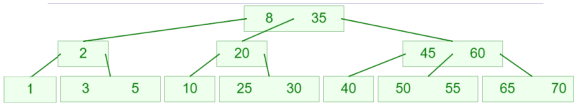
inserted 50 at bottom level

done:

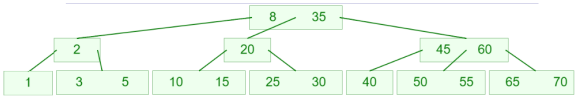
45 pushed up from <40,45,50>



inserted 55 at bottom level



inserted 30 at bottom level



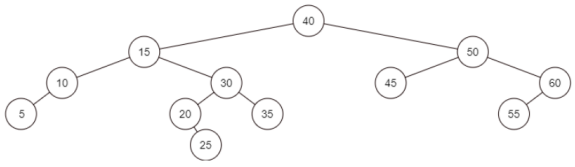
inserted 15 at bottom level

6)

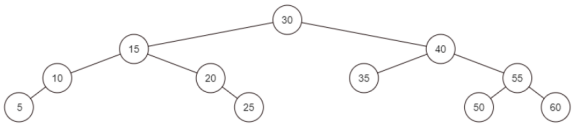
- Step 1: Delete an element from the tree using Binary Search Tree deletion logic
- Step 2: Check the Balance Factor from bottom to root for all ancestors of the deleted node
- Step 3: If the Balance Factor of every node of ancestors of the deleted node is -1, 0 or 1 that means that our tree is balanced and we can start the next operation of the program.
- Step 4: Otherwise, it means that there is a balance factor that is not equal to -1, 0, or 1, so starting from the bottom to up for all ancestors of the added node until root we need to check the Balance Factor and if it's not satisfied the Balance Factor's condition, perform the following operations to make the tree balanced. After that, if the condition from step 3 holds, we can proceed to the next operation of the program, otherwise, repeat step 4.

```
//Following operations
//Balance Factor (BF)
//let z be a current node and y to be it's child in which new node were added
if BF(z) > 1
  if BF(y) >= 0 //y is a left child of z
    right-rotate(z)
  else
    left-rotate(y)
    right-rotate(z)
else if BF(z) <-1
  if BF(y) <= 0 //y is a right child of z
    left-rotate(z)
  else
    right-rotate(y)
    left-rotate(z)
```

7)



initial tree



removed 45

node with value 50 was unbalanced, so we have done:

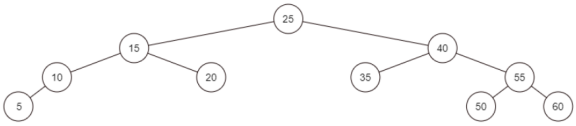
right rotate 60

left rotate 50

node with value 40 was unbalanced, so we have done

left rotate 15

right rotate 40



removed 30

tree is balanced ⇒ no rotate

4 task

```
class MedianHeap<T> {
  minH //min heap which elements are of type T, in that heap we will store elements >= median
  maxH //max heap which elements are of type T, in that heap we will store elements <= median
//Size of minH should differ on -1, 0 or 1 from maxH

//This function inserts value in our MedianHeap
//in min or in max heaps with using of
//insert(), size(), removeMin(), removeMax()
//functions from binary heap interface
//median value would be top of min heap,
//if min heap size >= max heap size.
//median value would be top of max heap,
//if min heap size < max heap size.
//So, we compare our input element with median
//and decide in which heap we need to insert it.
//if input element >=median, insert in min heap
//otherwise, insert in max heap.
//After insertion we need to check that
//size of minH is differ on -1, 0 or 1 from maxH
//and if it is not holds, rebalance heaps
//by moving top element from the bigger heap in lower one.
  void insert(T t) { // O(log(N))
    minH.insert(t);
    maxH.insert(t);
    if (this.isEmpty()) {
      minH.insert(t);
    } else {
      T median;
      if (minH.size() >= maxH.size()) {
        median = minH.min();
      } else {
        median = maxH.max();
      }
      if (t >= median) {
        minH.insert(t);
      } else {
        maxH.insert(t);
      }
      if (minH.size()+1 < maxH.size()) {
        minH.insert(maxH.removeMax());
      }
      if (maxH.size()+1 < minH.size()){
        maxH.insert(minH.removeMin());
      }
  }

//This function removes median value from
//our MedianHeap in both mini and max heaps
//with using removeMax(), removeMin() and size()
//functions from binary heap interface.
//The logic is the following:
```

```
//If our MedianHeap is empty, we print a message and return -1
//Otherwise we do the following:
//We compare sizes from min and max heaps
//and return top element from the biggest of them.
  T remove_median() { // O(log(N))
    if (isEmpty()) {
      print "Median Heap is empty"
      return -1
    }
    if (minH.size() >= maxH.size()) {
      return minH.removeMin();
    } else {
      return maxH.removeMax();
    }
  }

//Since size of our MedianHeap is the same as
//sum of sizes of the min and max heaps, we just use
//size() function from binary heap interface for both heaps
//and summarize the results
  int size() { //O(1)
    return minH.size()+maxH.size();
  }

//Since size of our MedianHeap is the same as
//sum of sizes of the min and max heaps, we just use
//size() function from binary heap interface for both heaps,
//summarize the results and compare it with 0.
  bool isEmpty() { //O(1)
    return (minH.size()+maxH.size())== 0;
  }
}
```

Asymptotic worst case time complexity of methods are the following:

- insert(k) function has asymptotic worst case time complexity $O(log(n))$ because it uses functions for heaps, each of which works with asymptotic worst case time complexity $O(log(n))$ or lower.
  (we don't have any loops or recursive calls, so total time complexity would be just a particular sum of time complexities of all functions)
- remove_median() function has asymptotic worst case time complexity $O(log(n))$ because it uses functions for heaps, each of which works with asymptotic worst case time complexity $O(log(n))$ or lower.
  (we don't have any loops or recursive calls, so total time complexity would be just a particular sum of time complexities of all functions)
- size() has asymptotic worst case time complexity $O(1)$ because it uses size function for heaps, which works with asymptotic worst case time complexity $O(1)$
- isEmpty() has asymptotic worst case time complexity $O(1)$ because it uses size function for heaps, which works with asymptotic worst case time complexity $O(1)$