Gina Cody School of Computer Science and Software Engineering
Concordia University
COMP 352: Data Structure and Algorithms

Student: Duc Nguyen

Contents

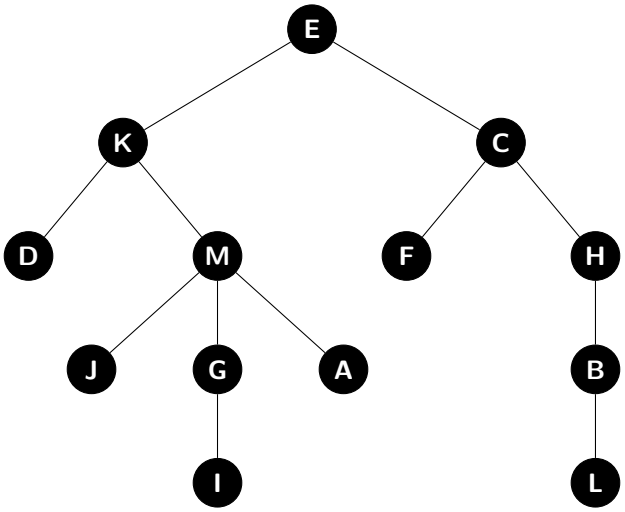
1 Draw a Tree statisfies the conditions

1.1 Instruction

Draw a (single) tree T, such that

- Each internal node of T stores a single character;
- A preorder traversal of T yields: E K D M J G I A C F H B L;
- A postorder traversal of T yields: D J I G A M K F L B H C E.

1.2 Tree satisfies the requirements



2 Smallest x numbers in the array

2.1 Instruction

Given an array A of n integers, write an algorithm that finds the smallest x values in the array, where x $\leq$n. However, the first constraints are given: You are not allowed to use heaps; the array A should not be modified, and you can only use a maximum auxiliary space of O(n).

2.2 Find x smallest elements in the array

Algorithm 1 Find, and return the x smallest elements in a given array

Input: the array of integer A, and an integer x represents the requested number of return elements.

Output: x smallest elements from the array A

```
1: procedure SMALLESTELEMENTS(integer array A, integer x)
2:     P ← priority queue with comparator for integer
3:     for each value v ∈ A do
4:         P.insert(v)
5:     result ← an integer array to store the result
6:     i ← 0
7:     while i < x do
8:         result[i] ← P.removeMin()
9:     return result
```

Since the algorithm only relies on one more priority queue which has the same number of element as the given array's, the space complexity would be of ***O(n)****.*

3 Construct a min-heap by bottom-up heap algorithm

3.1 Instruction

Draw the min-heap that results from running the bottom-up heap construction algorithm on the following list of values:

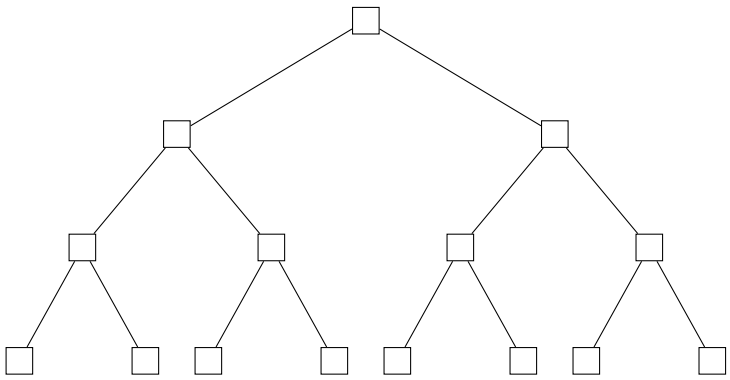
42 25 49 18 57 105 112 35 38 64 53 20 25 19 16

Starting from the bottom layer, use the values from left to right as specified above. Show intermediate steps and the final tree representing the min-heap. Afterwards perform the operation removeMin() 3 times and show the resulting min-heap after each step.

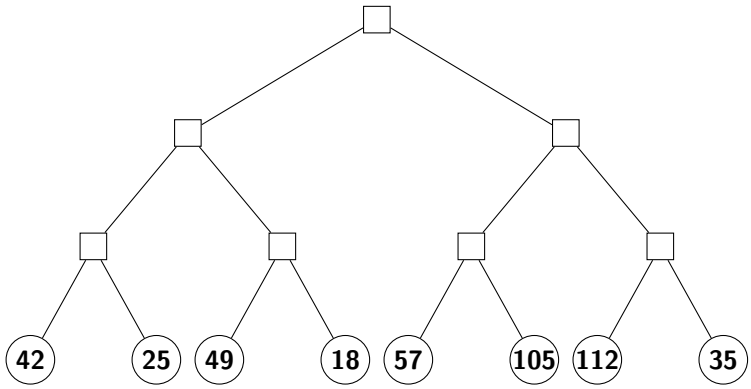
3.2 Steps to use bottom-up heap algorithm to construct a min-heap

- There are 15 entries need to be inserted
- Therefore, need to construct a heap with height

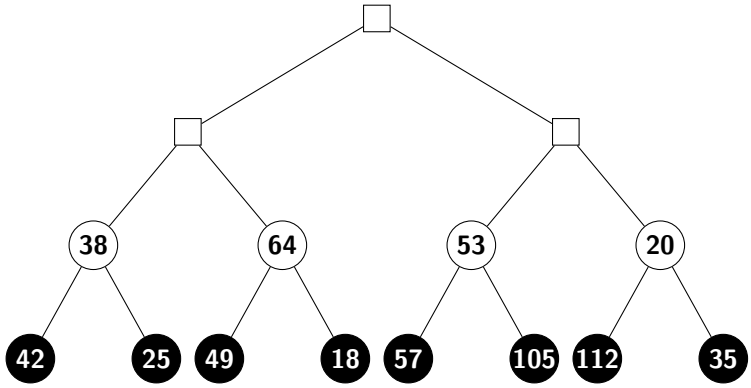
$$\left\lceil \log\left(N+1\right)-1 \right\rceil = \left\lceil \log\left(15+1\right)-1 \right\rceil = 3 \tag{1}$$



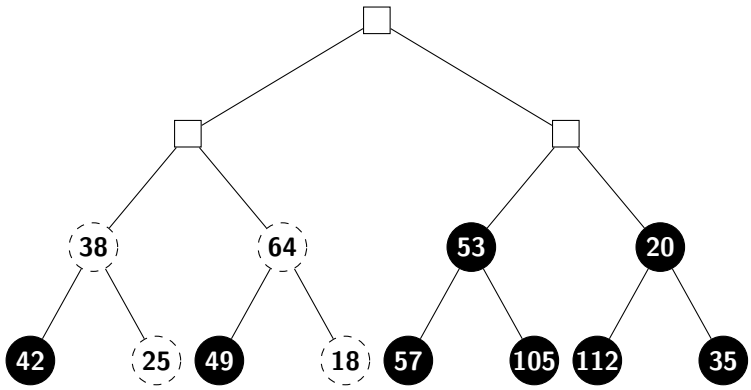
Initialize Structure (height = 3) with empty entries



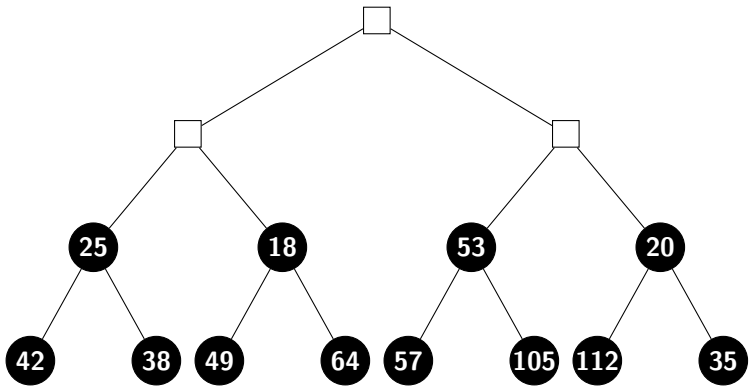
Construct $\frac{(n+1)}{2} = 8$ first elements to store in each entries at height 3. No bubbling needed



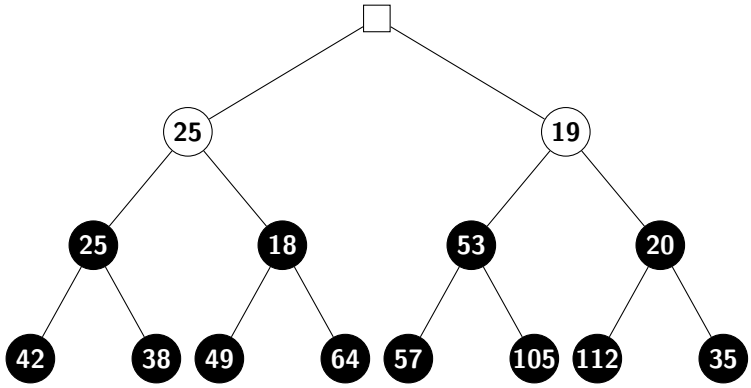
Construct $\frac{(n+1)}{4} = 4$ next elements to store in each entries at height 2. Bubbling needed.



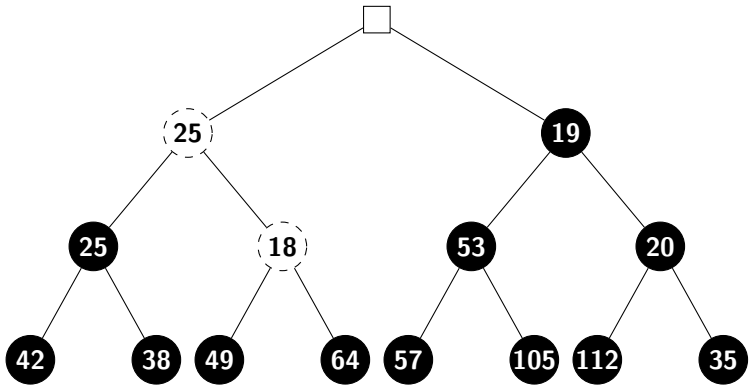
Bubbling needed for 38-25 and 64-18.



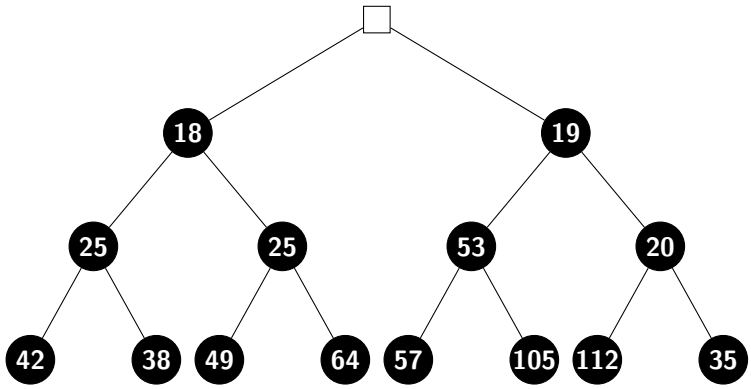
Finished bubbling.



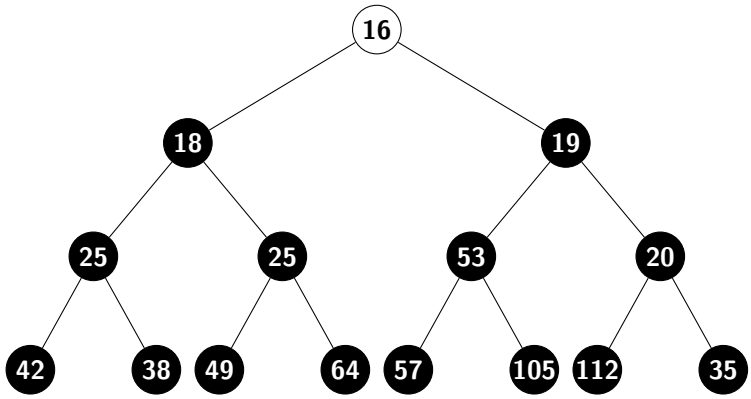
Construct $\frac{(n+1)}{8} = 2$ next elements to store in each entries at height 1. Bubbling needed.



Bubbling needed for 25-18

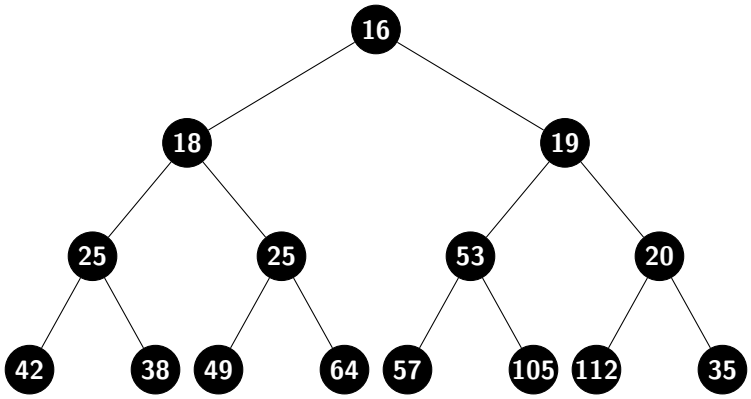


Finished bubbling.



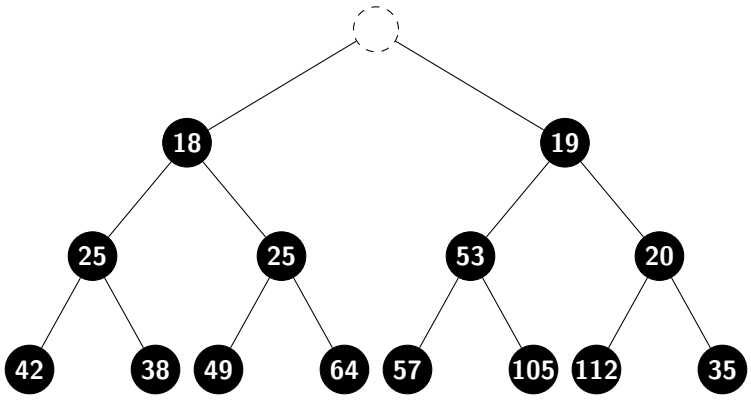
Construct $\frac{(n+1)}{16} = 1$ next element (last element) to store in entry at height 0. No conflict found.

3.3 Final min-heap tree after bottom-up constructing

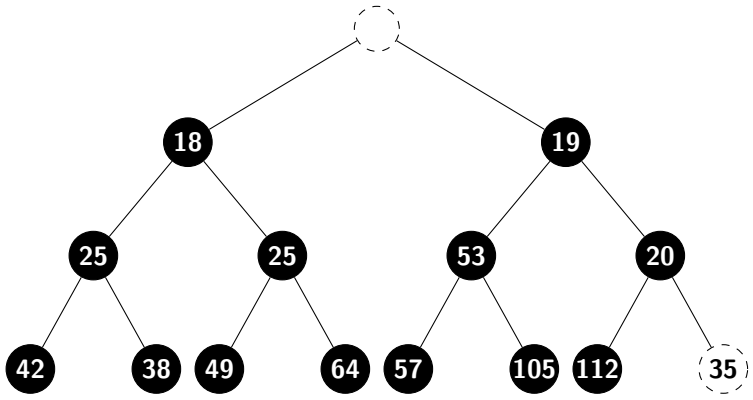


3.4 Perform removeMin() 3 times

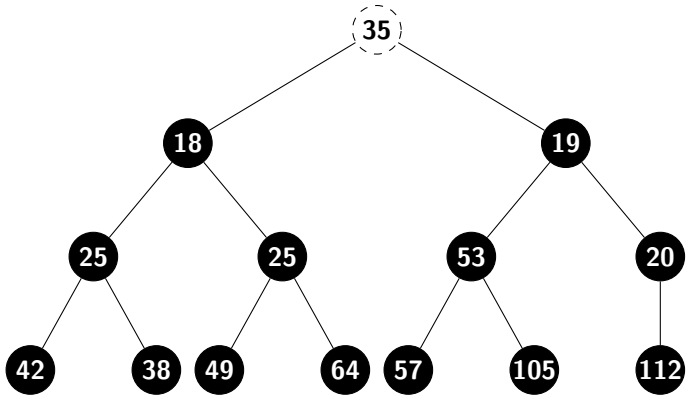
3.4.1 First removeMin()



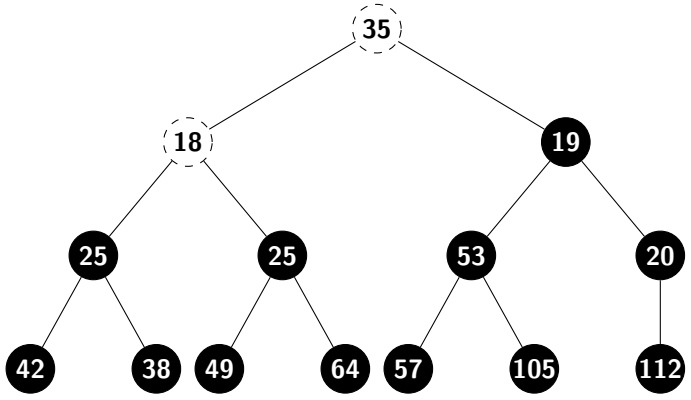
Remove the root's key. Return the key 16



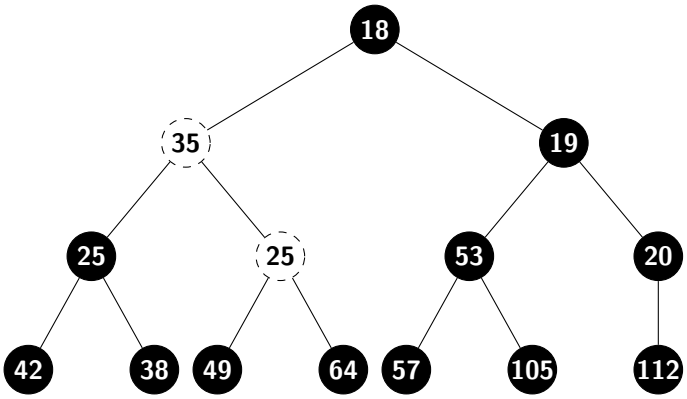
The last node is 35



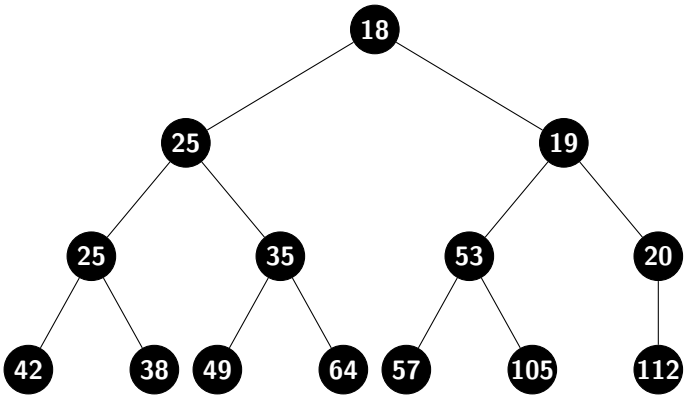
The entry of the last node will replace the root key. Then, remove the last node out of the heap



Start down-heap bubbling for 35-18

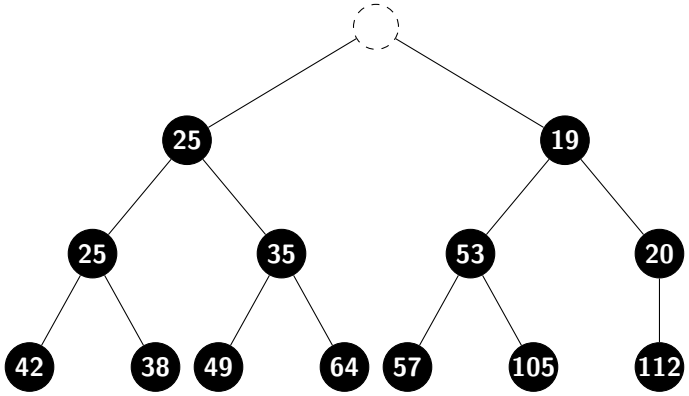


Continue down-heap bubbling for 35-25

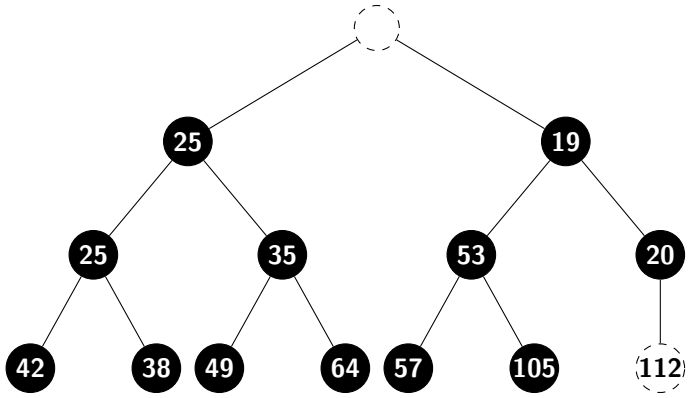


Finished bubbling

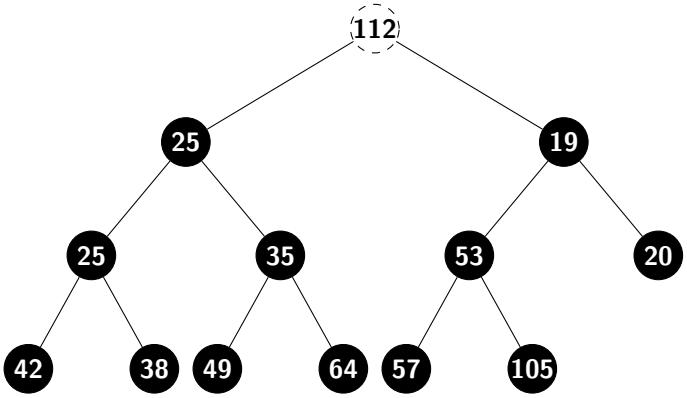
3.4.2 Second removeMin()



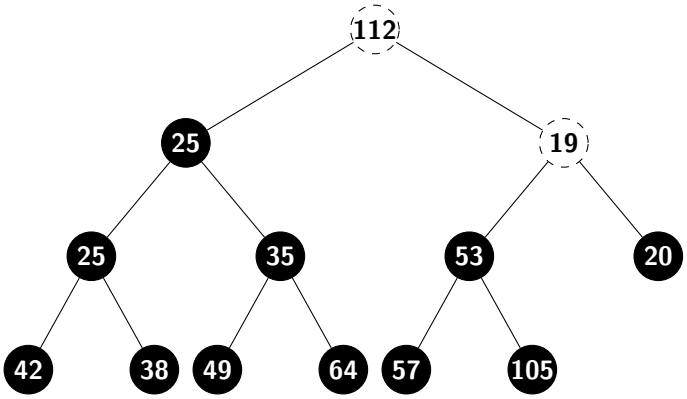
Remove the root's key. Return the key 18



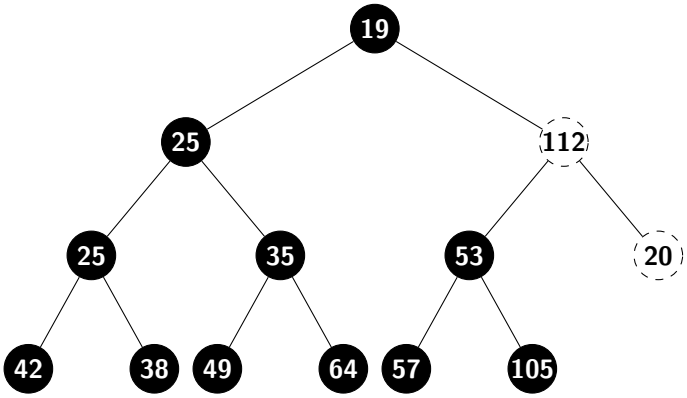
The last node is 112



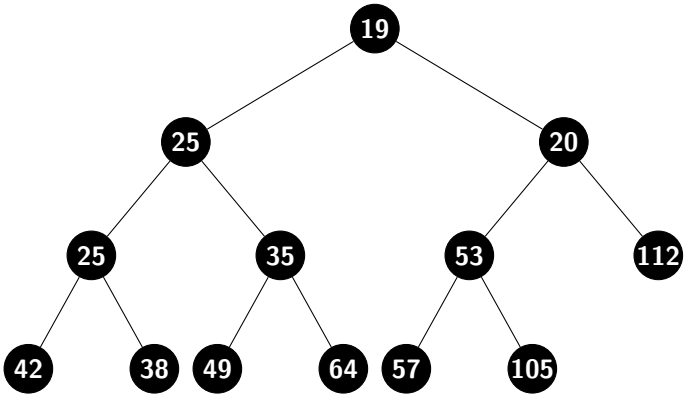
The entry of the last node will replace the root key. Then, remove the last node out of the heap



Start down-heap bubbling for 112-19

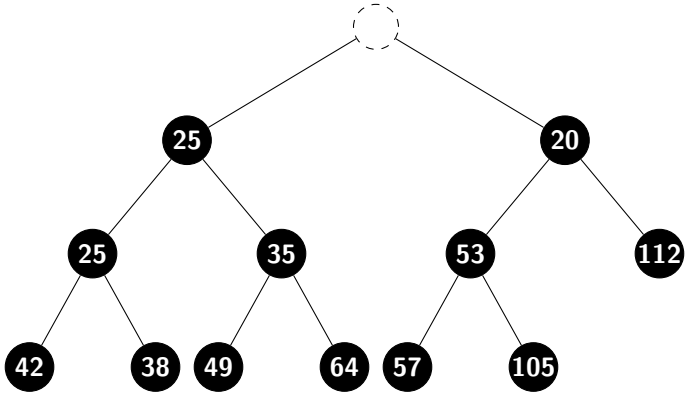


Continue down-heap bubbling for 112-20

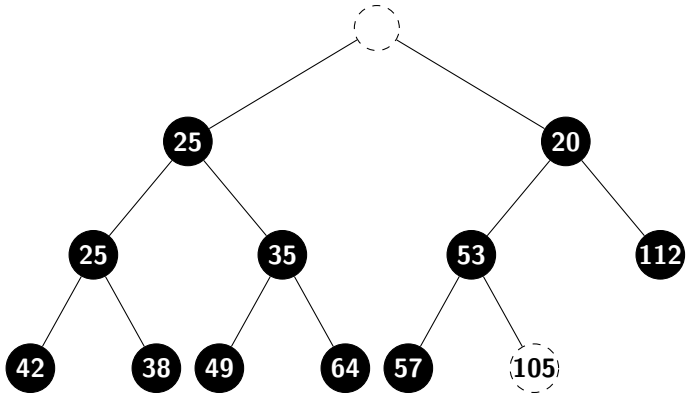


Finished bubbling

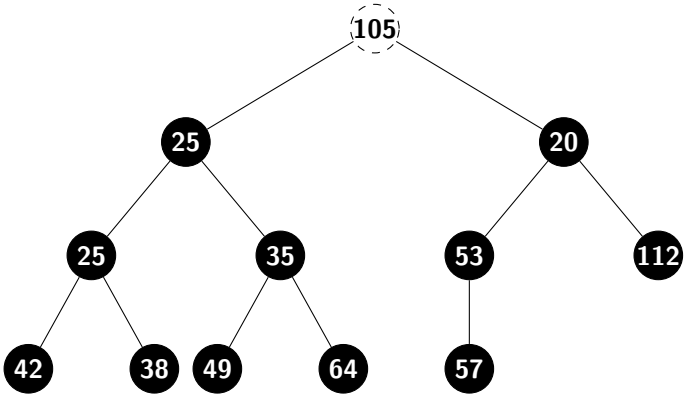
3.4.3 Third removeMin()



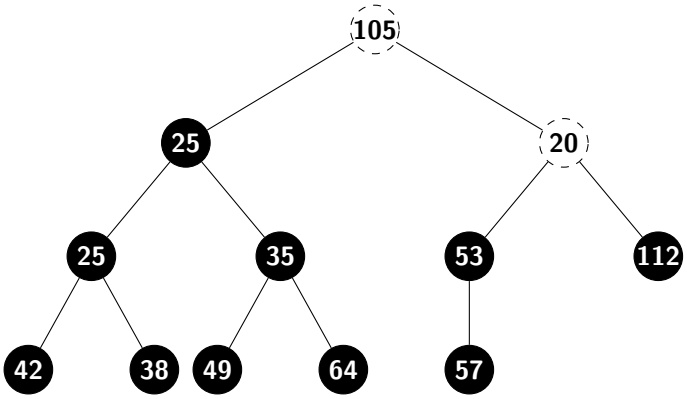
Remove the root's key. Return the key 19



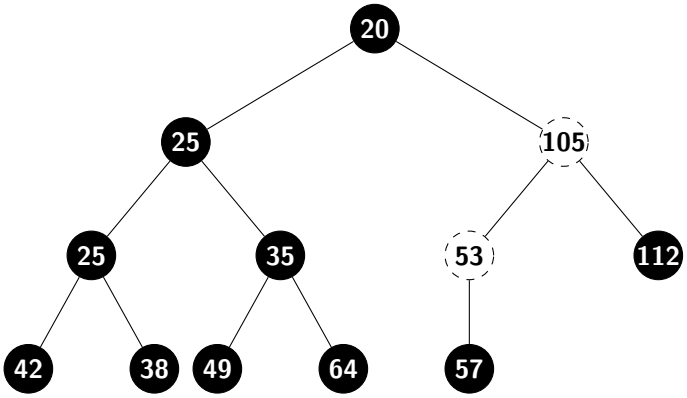
The last node is 105



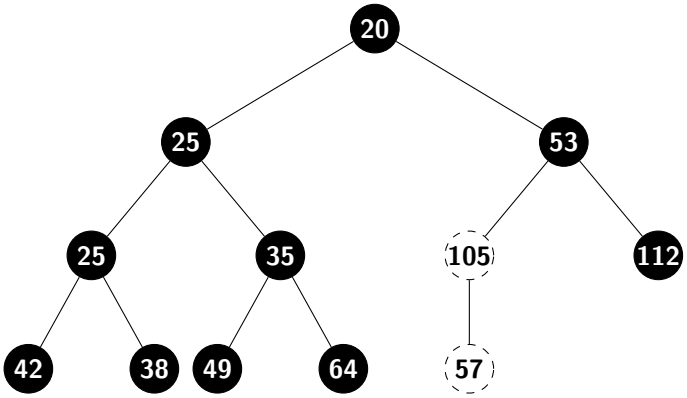
The entry of the last node will replace the root key. Then, remove the last node out of the heap



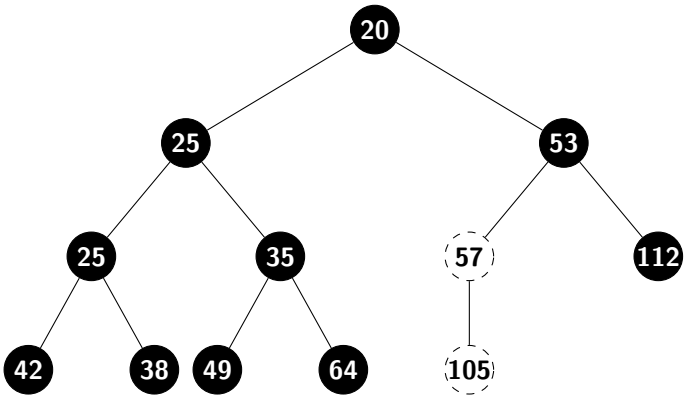
Start down-heap bubbling for 105-20



Continue down-heap bubbling for 105-53

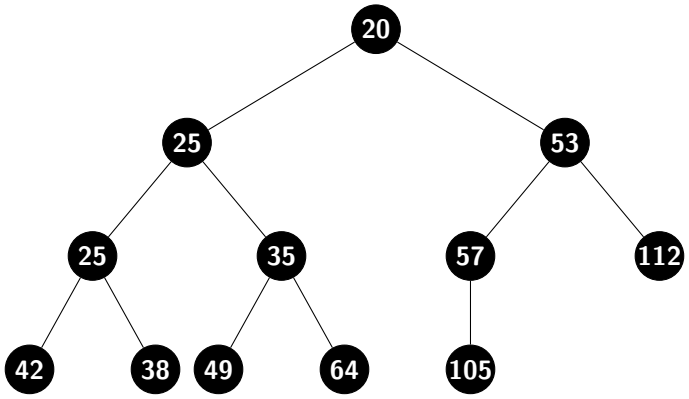


Continue down-heap bubbling for 105-57



Finished bubbling

3.5 Final Min-heap tree



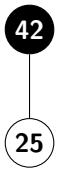
4 Create a max-heap

4.1 Instruction

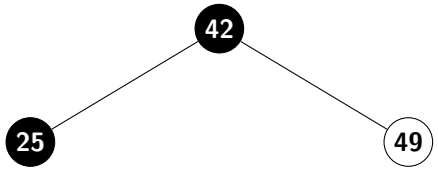
Create a max-heap using the list of values from Question 3 above but this time you have to insert these values one by one using the order from left to right (i.e insert 42, then 25, then 49, etc.). Show the tree after each step and the final tree representing the max-heap.

4.2 Step-by-step process to construct the Max-heap tree

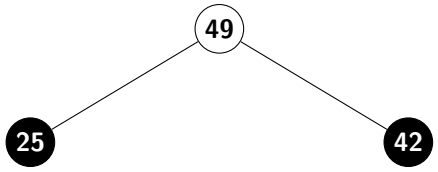
Adding 42 to the heap



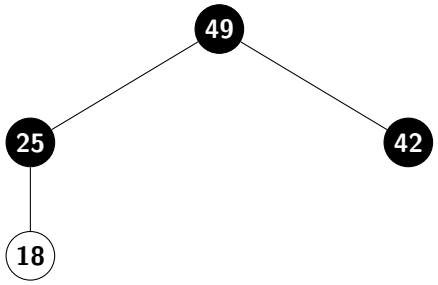
Adding 25 to the heap. No conflict found



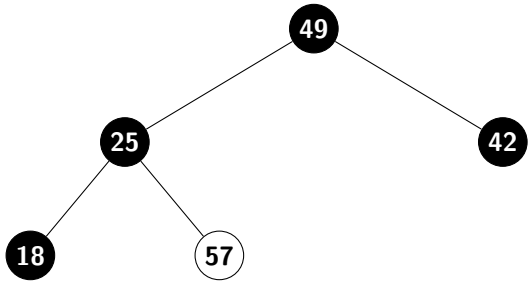
Adding 49 to the heap. Conflict found. Bubbling up



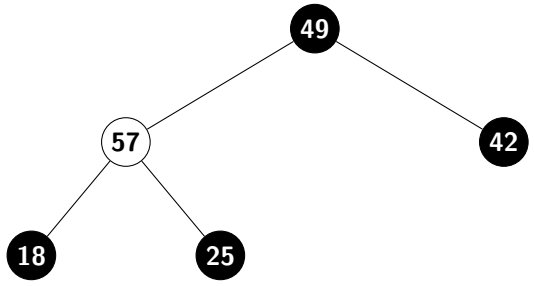
Swapping with its parents. $49 > 42$**. No remaining conflict**



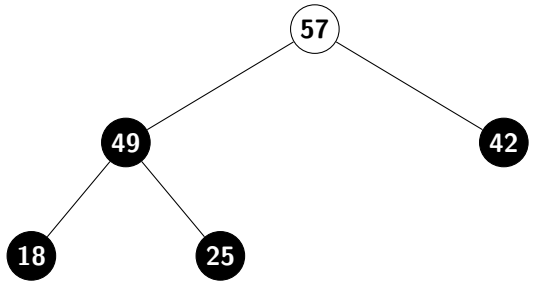
Adding 18 to the heap. No conflict found



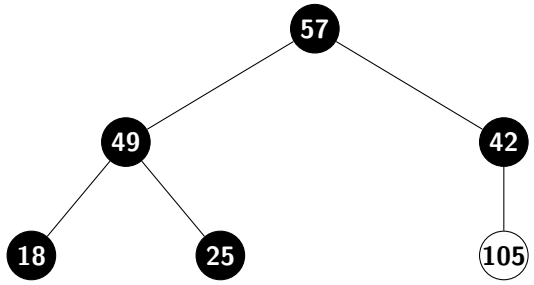
Adding 57 to the heap. Conflict found. Bubbling up



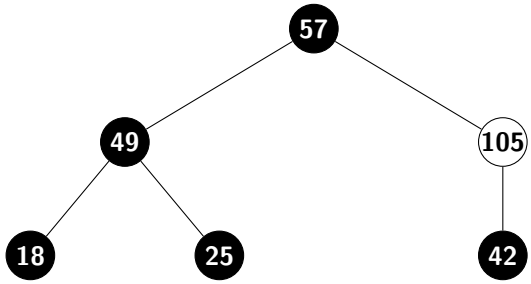
Swapping with its parents. $57 > 25$**. Still having conflict. Continue bubbling up**



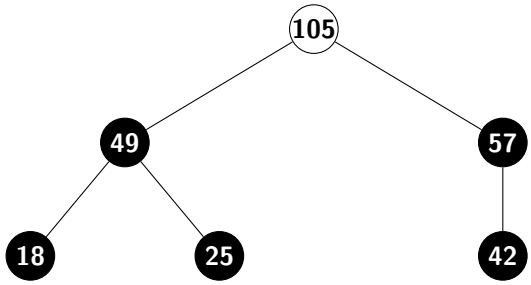
Swapping with its parents. $57 > 49$**. No remaining conflict**



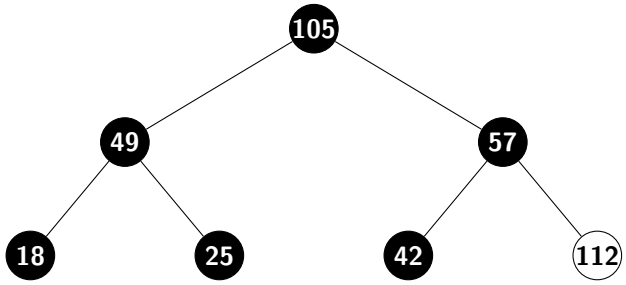
Adding 105 to the heap. Conflict found. Bubbling up



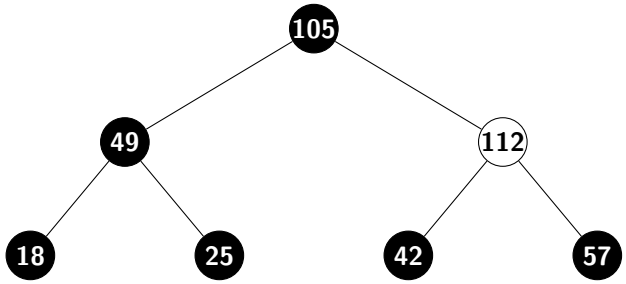
Swapping with its parents. $105 > 42$**. Still having conflict. Continue bubbling up**



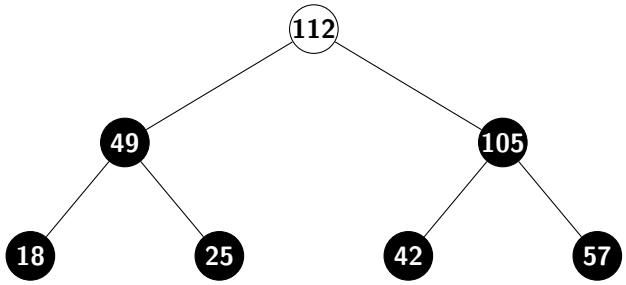
Swapping with its parents. $105 > 57$**. No remaining conflict**



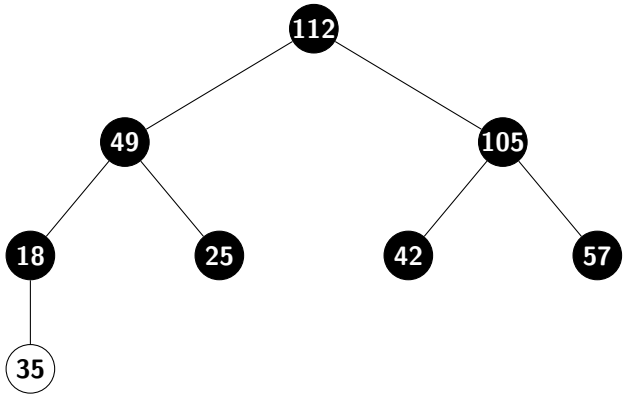
Adding 112 to the heap. Conflict found. Bubbling up



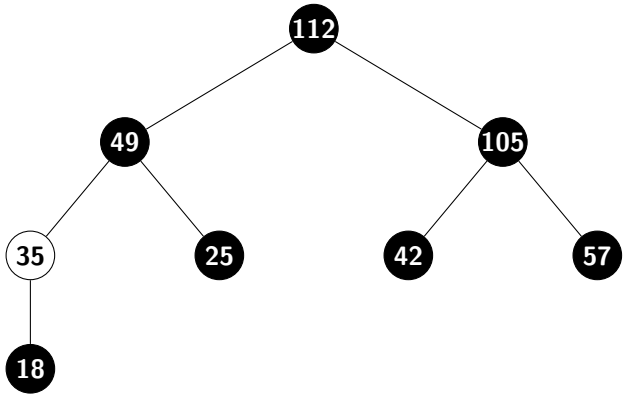
Swapping with its parents. $112 > 57$**. Still having conflict. Continue bubbling up**



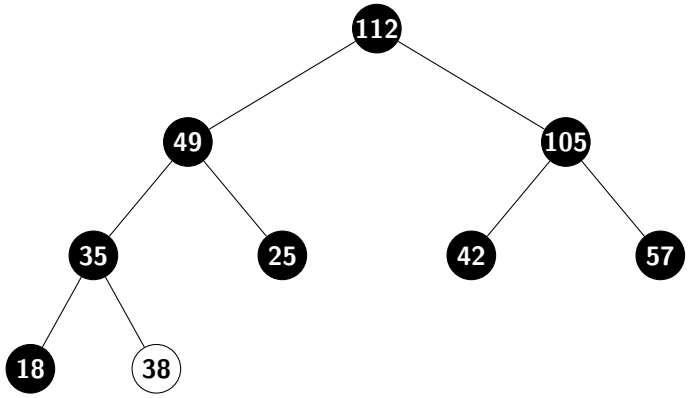
Swapping with its parents. $112 > 105$**. No remaining conflict**



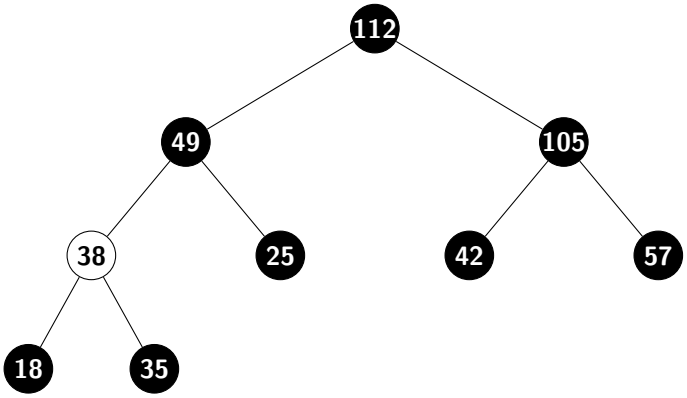
Adding 35 to the heap. Conflict found. Bubbling up



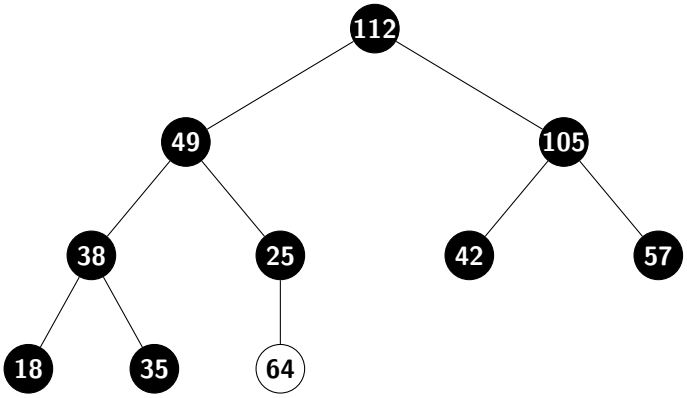
Swapping with its parents. $35 > 18$**. No remaining conflict**



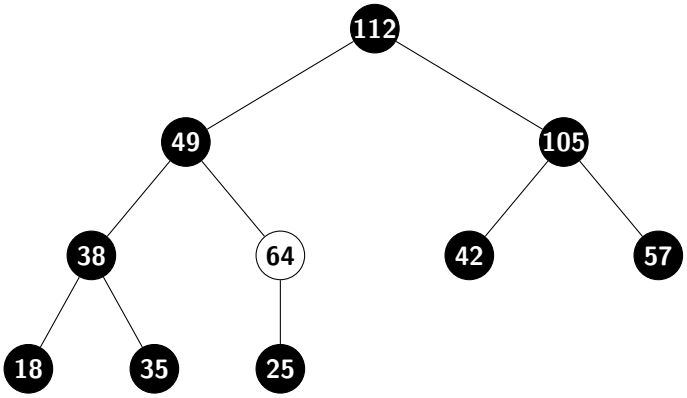
Adding 38 to the heap. Conflict found. Bubbling up



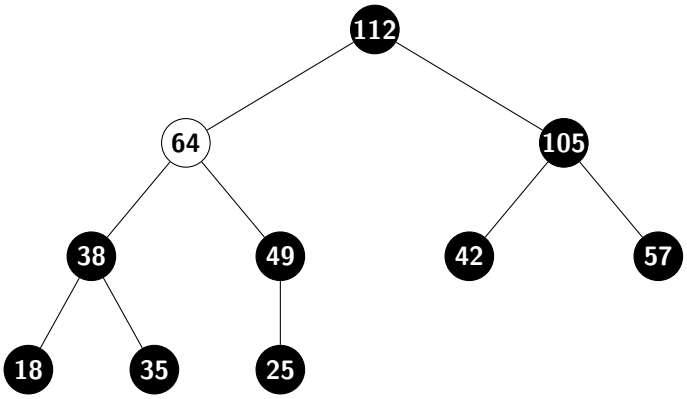
Swapping with its parents. $38 > 35$**. No remaining conflict**



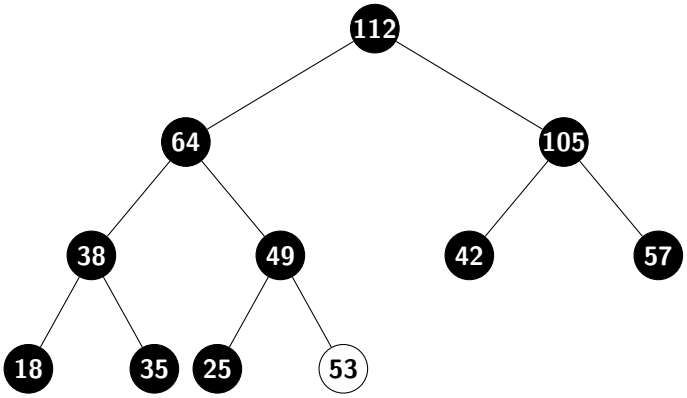
Adding 64 to the heap. Conflict found. Bubbling up



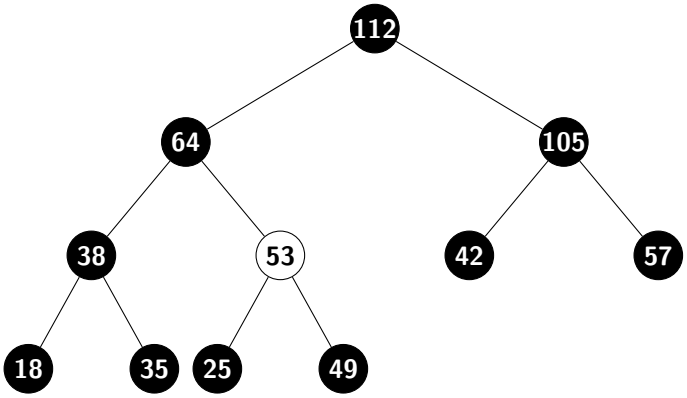
Swapping with its parents. $64 > 25$**. Still having conflict. Continue bubbling up**



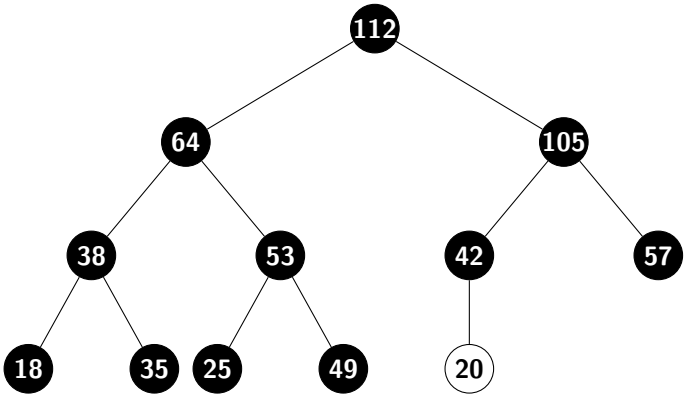
Swapping with its parents. $64 > 49$**. No remaining conflict**



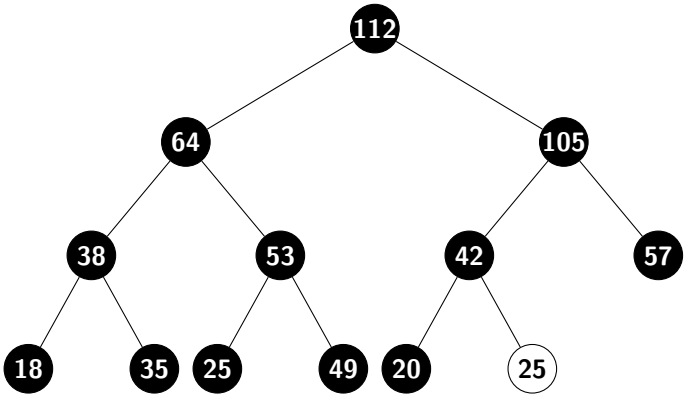
Adding 53 to the heap. Conflict found. Bubbling up



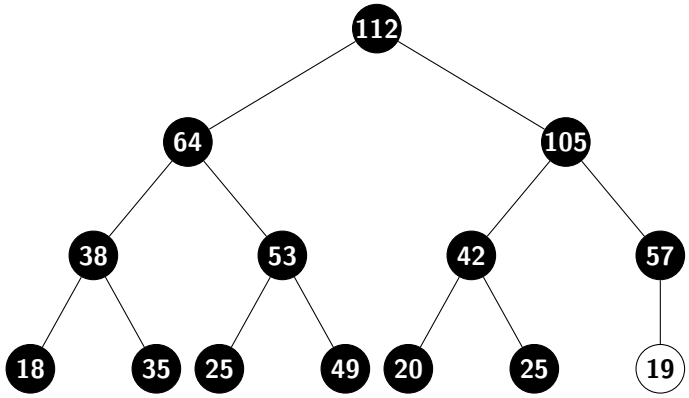
Swapping with its parents. $53 > 49$**. No remaining conflict**



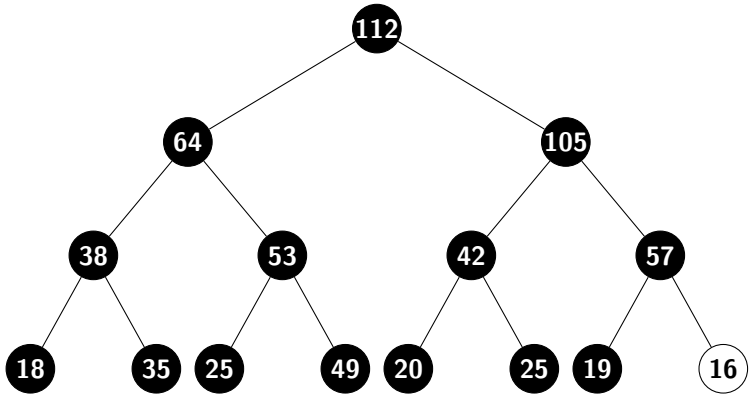
Adding 20 to the heap. No conflict found



Adding 25 to the heap. No conflict found

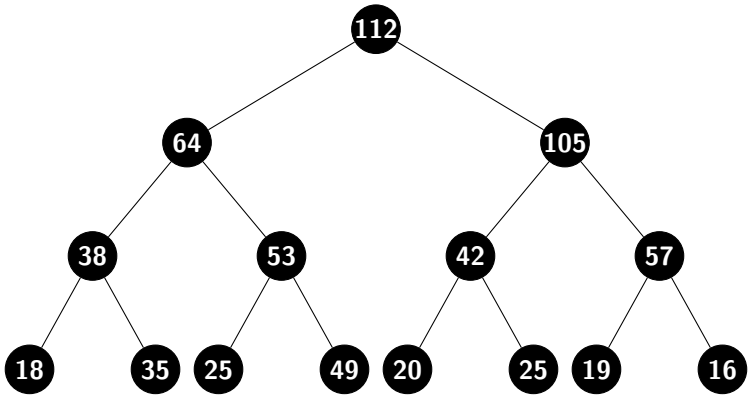


Adding 19 to the heap. No conflict found



Adding 16 to the heap. No conflict found

4.3 Final Max-heap tree



5 Trees

5.1 Instruction

Given 2 trees T1 and T2, where T1 has more nodes than T2, and where each of the nodes in both trees holds an integer value. Further, the values in any of the trees are assumed to be unique (no duplicate values within the same tree).

i) Give an algorithm that determines whether or not T2 represents a subtree of T1.

ii) What is the time and space complexity of your algorithm in i)?

iii) Will your algorithm still work if duplicate values are permitted in these trees?

a. If your algorithm still works when duplicate values are allowed, explain clearly why this change could not have affected your solution.

b. If your algorithm will fail if duplicate values are allowed, provide an example of the failure and propose some modification to fix this failure.

5.2 Check sub-tree

Algorithm 2 Check if a tree is the subtree of another tree or not

Input: Root node of trees t1, t2, in which t1 is the tree with the number of nodes is equal or greater than that of t2.

Output: a boolean value indicates if t2 is a subtree of t1.

```
1: procedure isSubTree(TreeNode rootT1, TreeNode rootT2)
2:     ArrayList t1 ← null
3:     ArrayList t2 ← null
4:     preOrder(rootT1, t1)
5:     preOrder(rootT2, t2)
6:     return isSubList(t1, t2)
```

Two dependent methods to support the main isSubTree method:

Algorithm 3

Input: A root node of a tree. An array list acts as a container for all the nodes inside the given tree.

Output: Assign the given arrayList to have all the nodes in the given tree with pre order traversal.

```
1: procedure preOrder(TreeNode node, ArrayList t)
2:     t.add(node)
3:     for each child w ∈ node do
4:         preOrder(w)
```

Algorithm 4

Input: An array list of the tree which has the number of nodes is more or equal than that of the second array list.

Output: A boolean value indicates if the array list t2 is a sub array of t1

```
1:  procedure isSubList(ArrayList t1, ArrayList t2)
2:      i ← 0
3:      j ← 0
4:      while i < t1.size and j < t2.size do
5:          if t1[i] = t2[j] then
6:              i++
7:              j++
8:              if j = t2.size then return true          ▷ iterated over t2.
9:          else
10:             i++
11:             j ← 0
        return false
```

5.3 Time and Space complexity

The problem concerns of 2 different input: tree 1 and tree 2. Let denote the number of nodes inside the first tree t1 to be n, and that of the second tree t2 to be m.

- **Time Complexity:** The algorithm takes n, m time for t1, t2 respectively to iterate and put every node into the 2 created array lists. Therefore, it takes $\mathbf{n + m}$ time for this operation. Next, the algorithm calls the function **isSubList** to check if one of the array list is the sub array of the other. By doing this, the algorithm has to iterate over every element in the bigger array which has n elements. Thus, the time it takes here is **n**. In conclusion, the algorithm takes up to $\mathbf{O(n + m)}$ of time complexity.
- **Space Complexity:** The algorithm creates 2 new array lists to hold the elements from both trees. Thus, the space needs to be allocated is the total number of elements in both tree, which is $\mathbf{n + m}$. Space complexity is of $\mathbf{O(n+m)}$.

5.4 Algorithm still works if there are duplicate values?

The proposed algorithm above will work even when there are some duplicated elements inside one of the two given trees. Since the approach was to take all elements and put into one array list, and then check if one of them is the sub array of the other, it would guarantee that there is no possible misbehaviour with duplicated items.

6 The important of AVAILABLE object in linear probing for Hash-tables

6.1 Instruction

Assume the utilization of linear probing for hash-tables. To enhance the complexity of the operations performed on the table, a special AVAILABLE object is used. Assuming that all keys are positive integers, the following two techniques were suggested in order to enhance complexity:

- i) In case an entry is removed, instead of marking its location as AVAILABLE, indicate the key as the negative value of the removed key (i.e. if the removed key was 16, indicate the key as -16). Searching for an entry with the removed key would then terminate once a negative value of the key is found (instead of continuing to search if AVAILABLE is used).

  ii) Instead of using AVAILABLE, find a key in the table that should have been placed in the location of the removed entry, then place that key (place the entire entry of course) in that location (instead of setting the location as AVAILABLE). The motive is to find the key faster since it is now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these proposals enhance the search time complexity? Additionally, will any of these approaches result in misbehaviour (in terms of functionalities, or even performance)? If so, explain clearly through illustrative examples.

6.2 Claim:

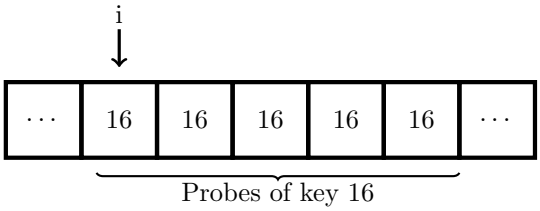
In this particular problem, since the statement indicates the sign of interchange in keywords "key" and "hash-value", the answer for this also assumes the interchanging of these two keywords. However, the distinction in the keywords is still important to be recognized in order to avoid any further confusion. "Key" is usually referred to an entry in a pair of element (key, value), while "hash-value" is usually the integer value of a key after being hashed by a function. In this problem, to whom read this, please perceive the term "key" as in fact, "hash value".

6.3 Consider alternative method I

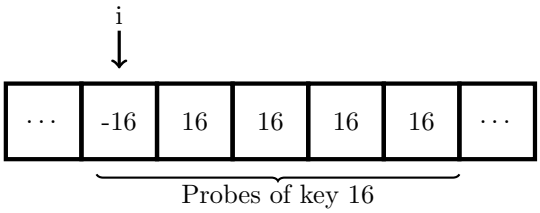
The first proposal will not change the *search* time complexity at all since the algorithm still has to iterate through all the probes, and eventually reaches the **recent removed entry** with negative value. Thus, the time complexity for *search* operation is still $\mathbf{O(n)}$, which means it does not improve anything.

Moreover, there is a high chance the algorithm will **misbehaviour** in many situations.

Consider the following situation:



Remove A[i]. Then as proposed, replace the key with negative value.

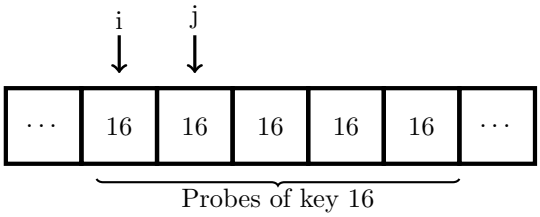


In this situation, when using *search* algorithm for key 16, the algorithm will stop at A[i], and return null indicating there is no such key in the array. While in reality, there are plenty of entries with key 16 in the probes after -16. Therefore, the algorithm misses the actual results.

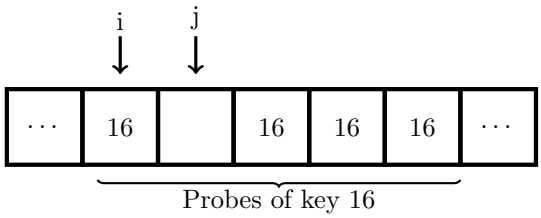
In fact, the algorithm only behaves as expected when there is **only** one single probe of a key. If that is the case, then it completely defeats the purpose of open-addressing.

6.4 Consider alternative method II

This proposed method will make a slightly improvement in *search* operation in the hash-table. However, it may lead to a more serious problem in term of algorithm behaviour. For example:



Removing the first element in the probe. Then found another key to replace the removed position



The *search* operation will terminate at A[j] since it is an empty position (indicating the end of probe). But in fact, there are more elements with the same key after that empty position. Thus, the algorithm misses checking those elements, resulting in a possible lost of searching results.
However, this problem could be avoided if the algorithm was implemented by checking every position after the first-found possible hashed location. But this results in a time complexity of *O(n)*, which means it does not improve anything. As a matter of fact, the algorithm is even slower in some cases than that using AVAILABLE object. Implementing the mechanism of AVAILABLE object will provide a chance to terminate the algorithm by reaching empty position without having to iterate until the last element in the table.

7 Collision handling with separate chaining

7.1 Instruction

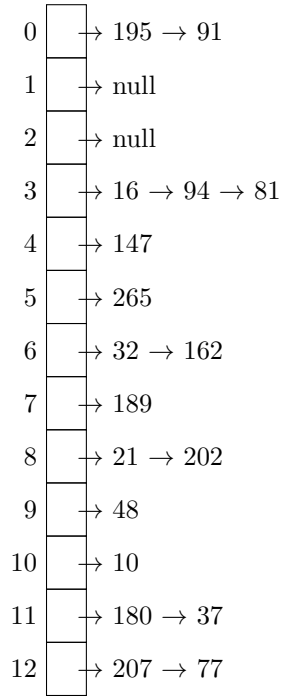
Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: h(k)=k mod 13.

- i) Draw the contents of the table after inserting elements with the following keys:

  32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48

- ii) What is the maximum number of collisions caused by the above insertions?

7.2 Contents of the Hash-table



Contents of the Hash-table after inserting

7.3 Maximum number of collisions

The maximum number of collisions is 3 collisions, happened at index 3 of the table with 3 values 16, 94, 81. The maxim

8 Attempt to reduce collisions in separate chaining

8.1 Instruction

To reduce the maximum number of collisions in the hash table described in Question 7 above, someone proposed the use of a larger array of 15 elements (that is roughly 15% bigger) and of course modifying the hash function to: h(k)=k mod 15. The idea is to reduce the load factor and hence the number of collisions.

Does this proposal hold any validity to it? If yes, indicate why such modifications would actually reduce the number of collisions. If no, indicate clearly the reasons you believe/think that such proposal is senseless.

8.2 Use a larger array to achieve a better load factor

The idea of reducing load factor by increasing the size of an array is actually a good solution to lower the number of collisions. When the size of the table is expanded, the distribution of hash value could be more uniformly, and well distributed. In another word, the probability of getting an uniformed distribution increases. Thus, a better load factor could be achieved.
However, with such a small increase with a small set of number as in the current case (expand from 13 elements to 15 elements), the improvement will just be a slight change. The number of collisions even could stay the same despite the decrease of the load factor.

9 Double-Hashing for collision handling

9.1 Instruction

Assume an open addressing hash table implementation, where the size of the array is N = 19, and that *double hashing* is performed for collision handling. The second hash function is defined as:

$$\mathbf{d(k) = q - k \bmod q}$$

where k is the key being inserted in the table and the prime number q is = 7. Use simple modular operation (k mod N) for the first hash function.

- i) Show the content of the table after performing the following operations, in order: **put(25), put(12), put(42), put(31), put(35), put(39), remove(31), put(48), remove(25), put(18), put(29), put(29), put(35).**
- ii) What is the size of the longest cluster caused by the above insertions?
- iii) What is the number of occurred collisions as a result of the above operations?
- iv) What is the current value of the table's *load factor*?

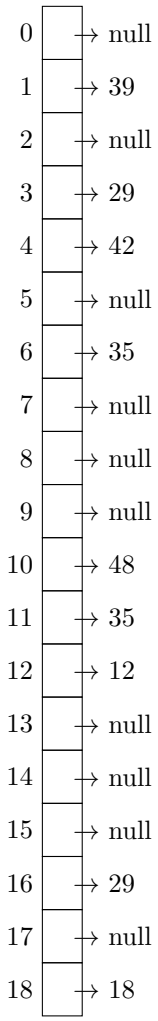
Summary

- q = 7
- N = 19
- 1st hash function: **k mod N**
- 2nd hash function: **d(k) = q - k mod q**

9.2 Contents of the hash-table

Operation	k	h(k)	d(k)	Probes
put	25	6	3	6
put	12	12	2	12
put	42	4	7	4
put	31	12	4	$12 \rightarrow 16$
put	35	16	7	$16 \rightarrow 4 \rightarrow 11$
put	39	1	3	1
remove	31	$\emptyset$	$\emptyset$	AVAILABLE object at 16
put	48	10	1	10
remove	25	$\emptyset$	$\emptyset$	AVAILABLE object at 6
put	18	18	3	18
put	29	10	6	$10 \rightarrow 16$
put	29	10	6	$10 \rightarrow 16 \rightarrow 3$
put	35	16	7	$16 \rightarrow 4 \rightarrow 11 \rightarrow 18 \rightarrow 6$

Table to construct the hash table



Hash Table

9.3 Size of the longest cluster

The longest cluster has size of 3 at index 10, 11, and 12 consecutively.

9.4 Number of occured collisions

There were 10 collisions happened.

9.5 Current load factor

- The table is an array of size 19 → N = 19

- Currently, there are 9 elements inside the hash table → n = 9

→ *The load factor* = $\frac{n}{N} = \frac{9}{19} \approx 0.47$

10 Radix-sort

10.1 Instruction

Show the steps that a radix sort takes when sorting the following array of integer keys:

832 91 411 172 243 573 326 292 682 489 96

10.2 Run Radix-Sort through the array

The original array

832	91	411	172	243	573	326	292	682	489	96

Run the first round of radix-sort with the most right bit of each key

832	91	411	172	243	573	326	292	682	489	96

Result as:

91	411	832	172	292	682	243	573	326	96	489

Run the second round with the second bit

91	411	832	172	292	682	243	573	326	96	489

Result as:

411	326	832	243	172	573	682	489	91	292	96

Run the third round with the third bit

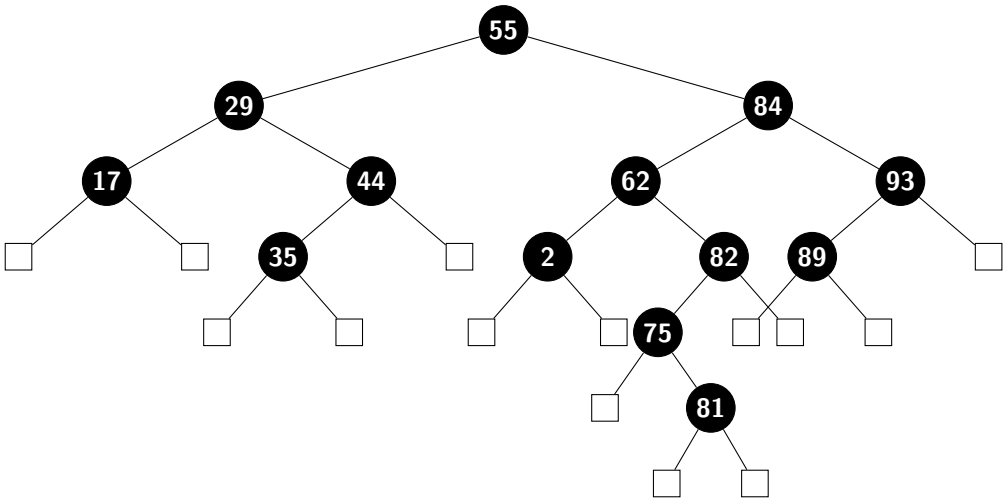
411	326	832	243	172	573	682	489	091	292	096

Result as:

91	96	172	243	292	326	411	489	573	682	832

The maximum length of a key is 3. The algorithm has gone through 3 rounds → Terminate radix-sort. The array of integer keys was sorted.

11 AVL tree



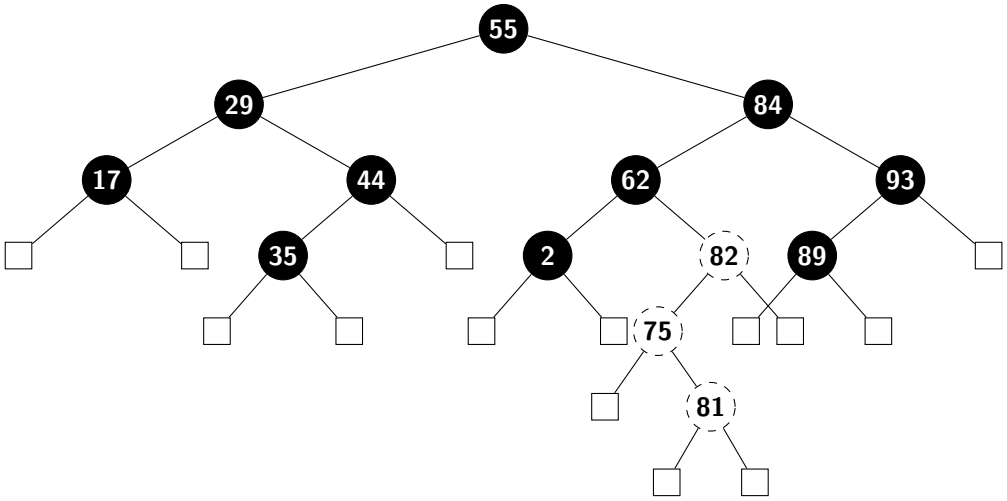
11.1 Instruction

- Are there any errors with the tree as shown? If so, indicate what the error(s) are, correct these error(s) [you should attempt applying the smallest possible number of changes to correct the tree], show the corrected AVL tree, then proceed to the following questions (Questions ii to iv) and start with the tree that you have just corrected. If no errors are there in the above tree, indicate why the tree is correctly an AVL tree, then proceed to the following questions (Questions ii to iv) and continue working on the tree as shown above.
- ii) Show the AVL tree after put(74) operation is performed. Give the complexity of this operation in terms of Big-O notation.

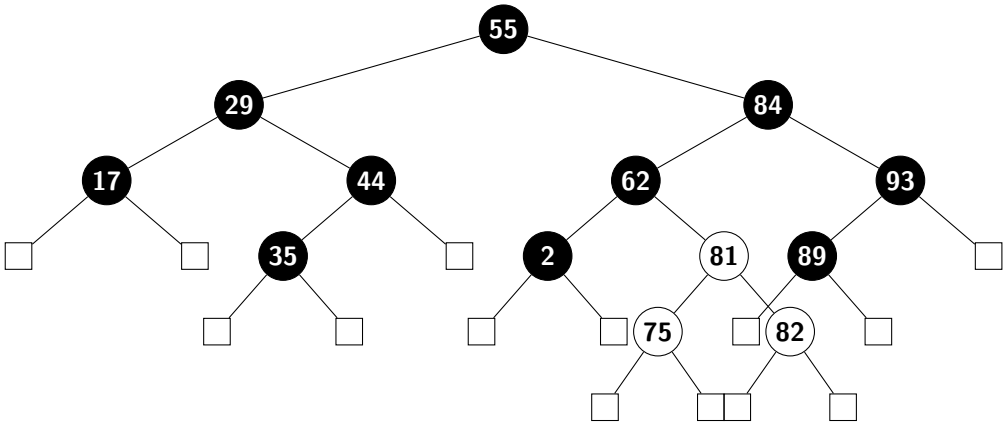
- iii) Show the AVL tree after remove(62) is performed. Give the complexity of this operation in terms of Big-O notation.
- iv) Show the AVL tree after remove(93) is performed. Show the progress of your work step-by-step. Give the complexity of this operation in terms of Big-O notation.

11.2 Check errors with the given AVL tree

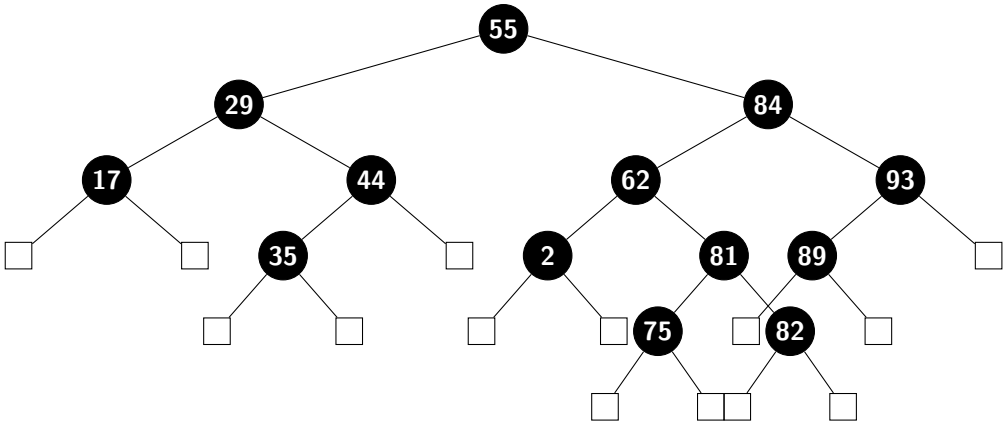
The error lying at the subtree of 82. The right sub tree has height 0 while the left subtree has heigh of 2. Thus, it is a violation of AVL tree's property.



After restructuring the sub tree of 82

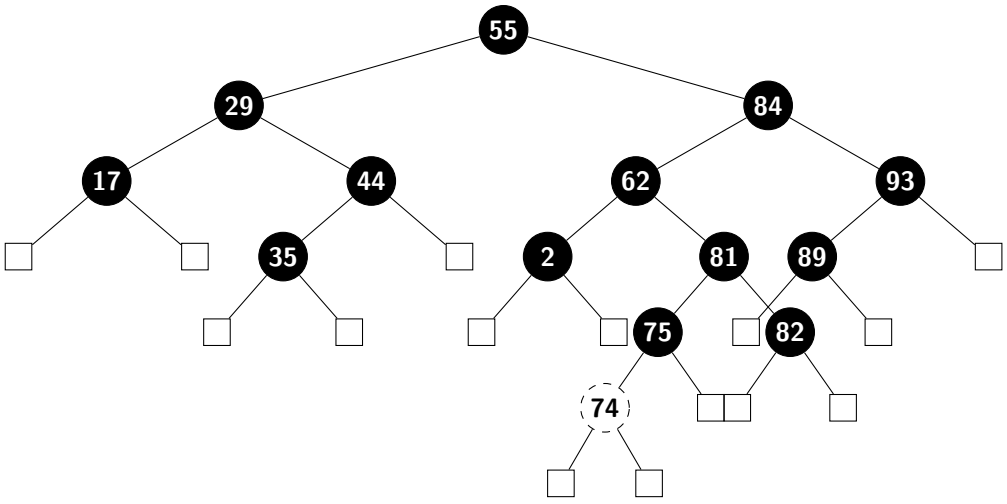


Final result:

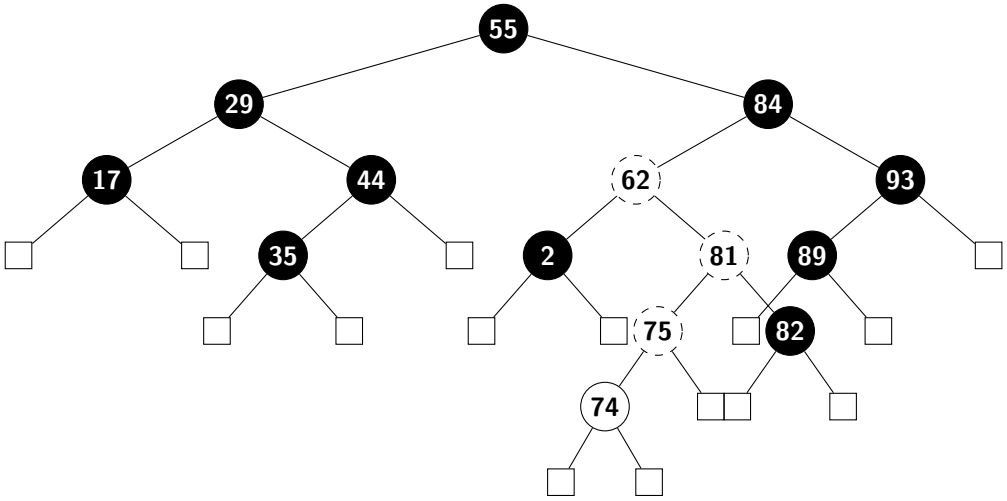


11.3 Put(74)

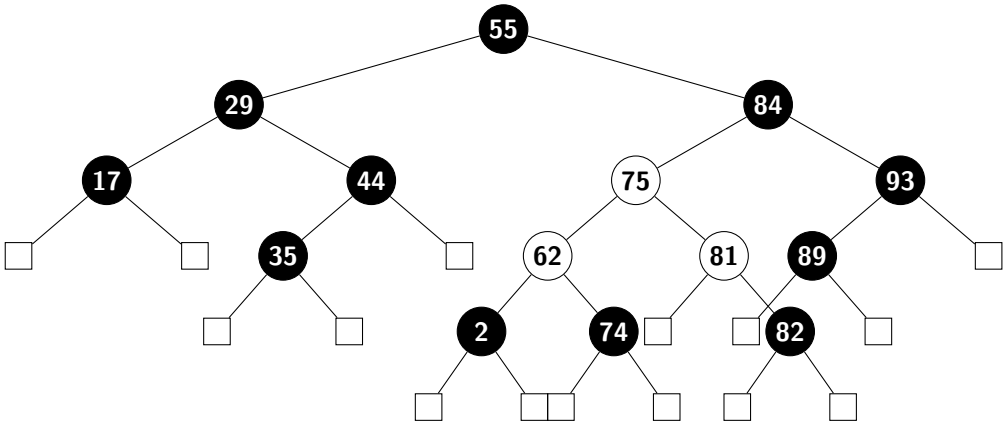
11.3.1 Perform the operation



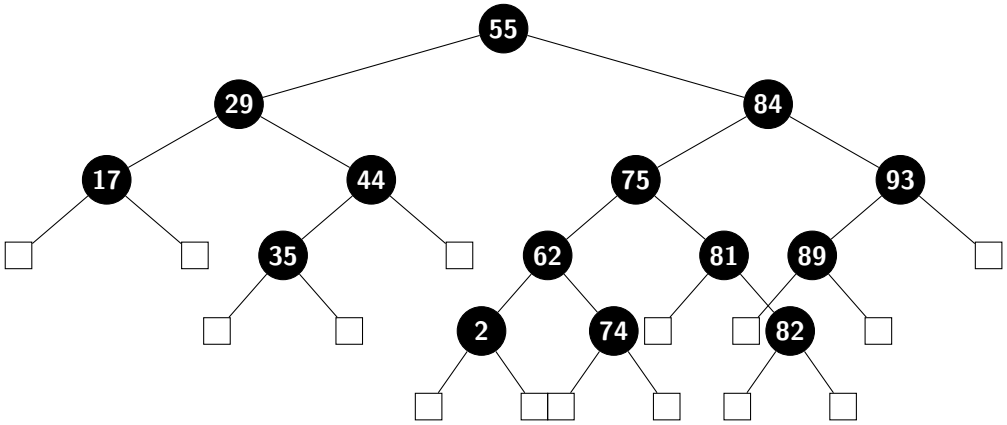
Need restructuring at the subtree of 62.



After restructuring:



Final result:



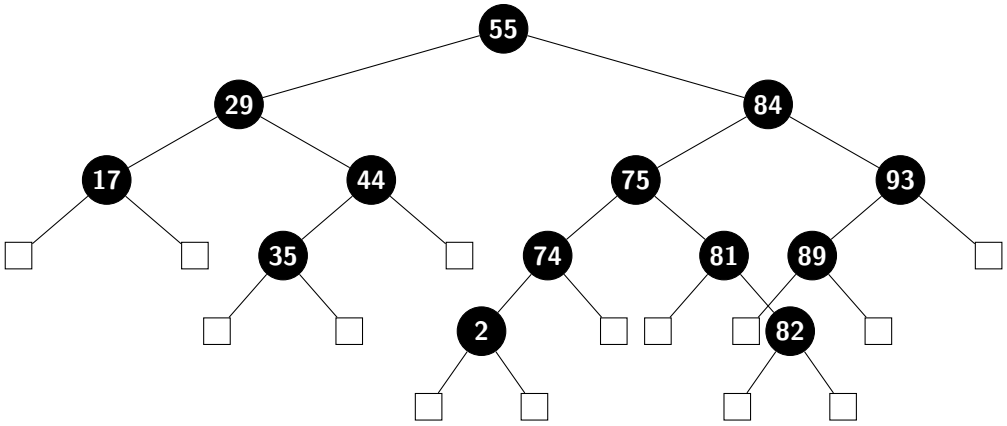
11.3.2 Time complexity

To find the place to put a new node at the beginning, it takes n time. In addition, to find out if there is a need for restructuring, the algorithm has to run log n time. To actually restructure the tree to suit the AVL tree's property, it takes constant time, meaning 1 in Big-O. In conclusion, the whole operation has the time complexity of **O(logn).**

11.4 remove(62)

11.4.1 Perform the operation

Final result:



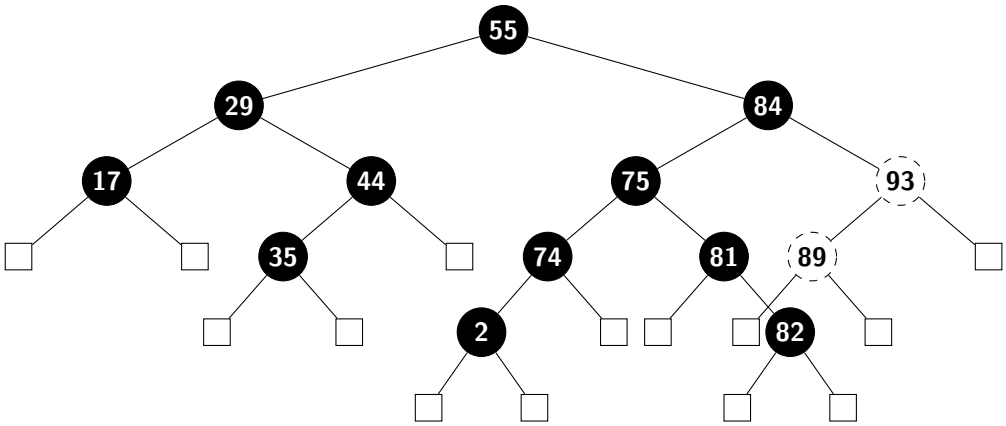
11.4.2 Time complexity

The time complexity is **O(log n)** since the algorithm has to find where to remove (log n), and then check if it needs restructuring or not (takes log n).

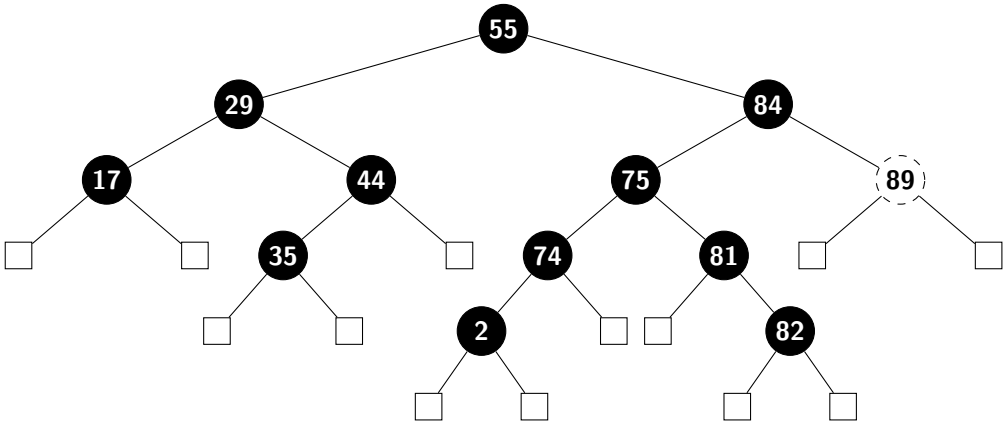
11.5 remove(93)

11.5.1 Step-by-step progress

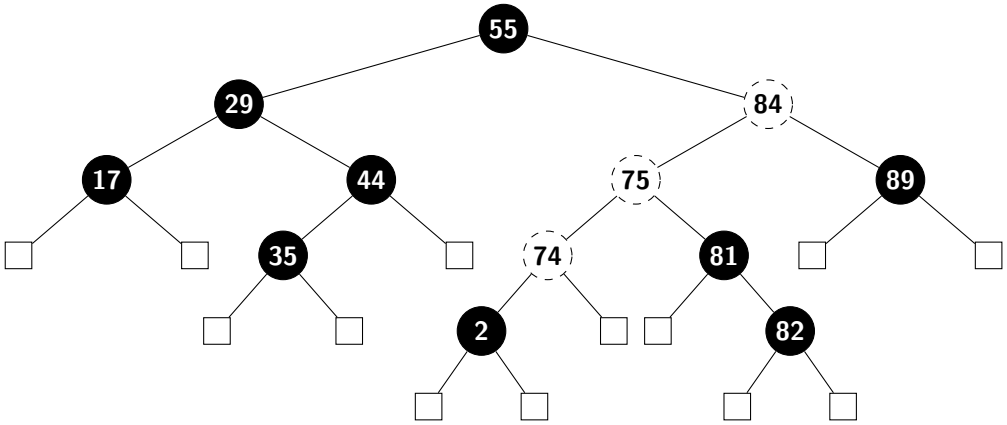
Remove(93). Traverse to find the replace node for 93. Found 89



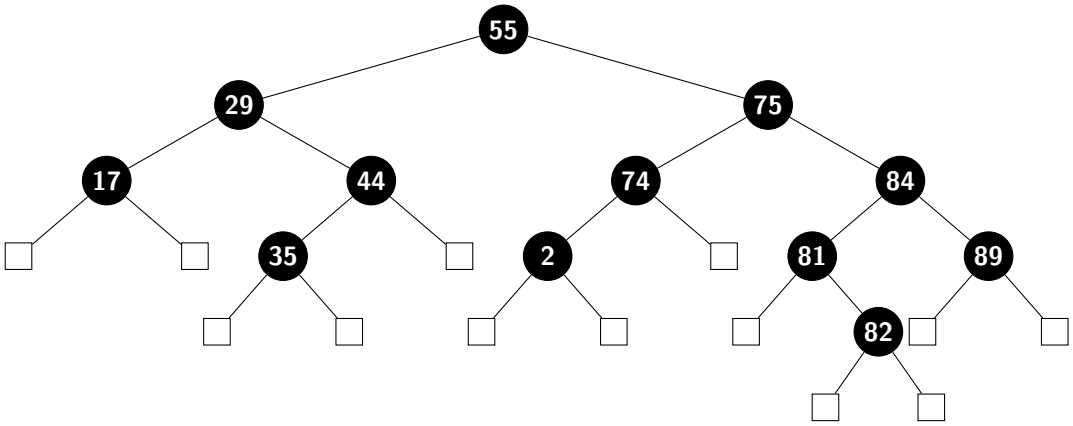
Remove 93. Replace 93 by 89



Need to restructure. The subtree 84 is violating AVL tree's property.



Use 3 main nodes: 84, 75, 74 to restructure the sub tree. Resulting:



11.5.2 Time complexity

In order to find the node to replace 93, the algorithm has to traverse log n time to find 93 and log n time to find the element that can replace the position of 93. Then, the algorithm has to run log n time to restructuring the tree. In conclusion, the total operation has time complexity of **O(log n).**