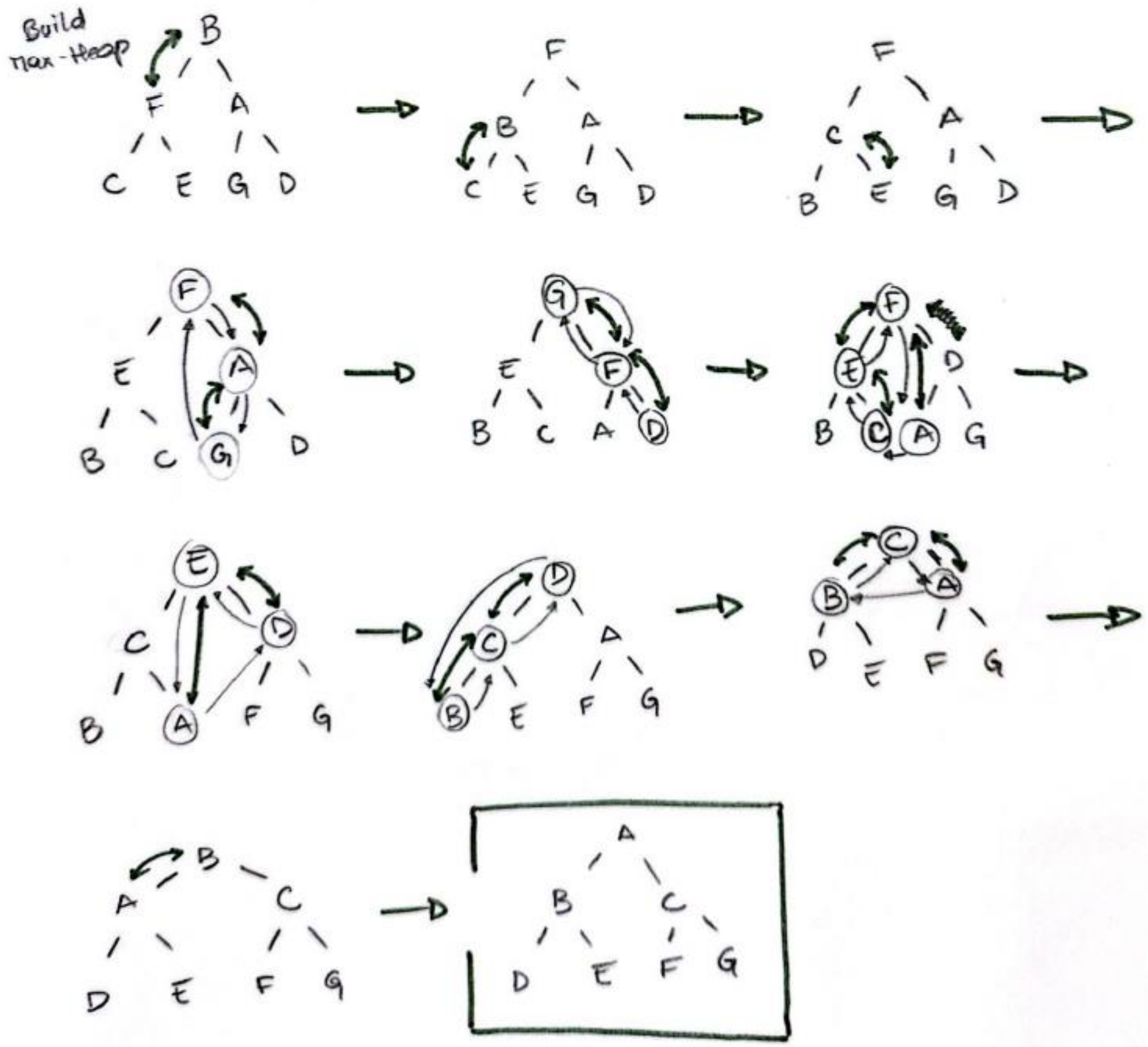
Valentina Silveira
Eagle ID: 901365377

Valentina Silveira
Eagle ID : 901365377

[Algorithm Assignment 2]

① HeapSort on A[1..7] = BFACEGD



Valentina Silveira
Eagle ID: 901365377

② Max-Heap Delete (A,i)

```
if $i > A.heapsize$
    error "index out of range"
Max-Heap-Increase-key $(A, i, \infty)$
Max-Heap-Extract-max(A)
```

a. Perform Max-Heap-Delete (A,2) on the given Max-Heap represented by array $A[1..7] = \langle 9,6,8,3,5,4,2 \rangle$

Step 1 - if $i > A.heapsize$ → check if $2 > 7$ × it's false continue to next line

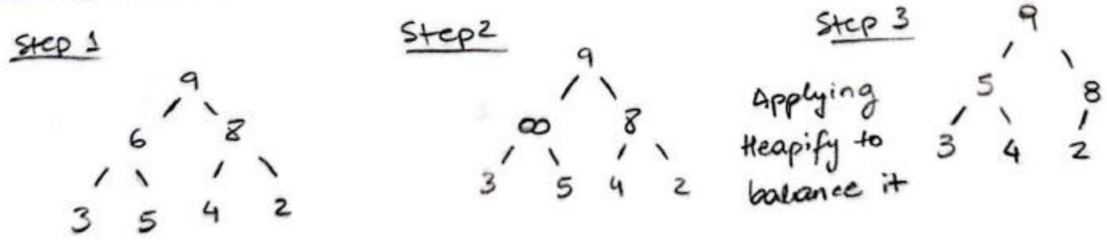
Step 2 - Max-Heap-Increase-key $(A, 2, \infty)$

- set $A[2] = \infty$ → update key at index 2('6') to ∞

Step 3 - Max-Heap-Extract-max (A)

- swap A[1] with A[7]
- decrease heapsite $A.heapsize = 6$
- max heapify from index $1 (A[1] = 9)$
- update Array $A[1..6] = \langle 9, 5, 8, 3, 2, 4 \rangle$

Binary tree representation



b. Best case scenario $\Theta(\log n)$

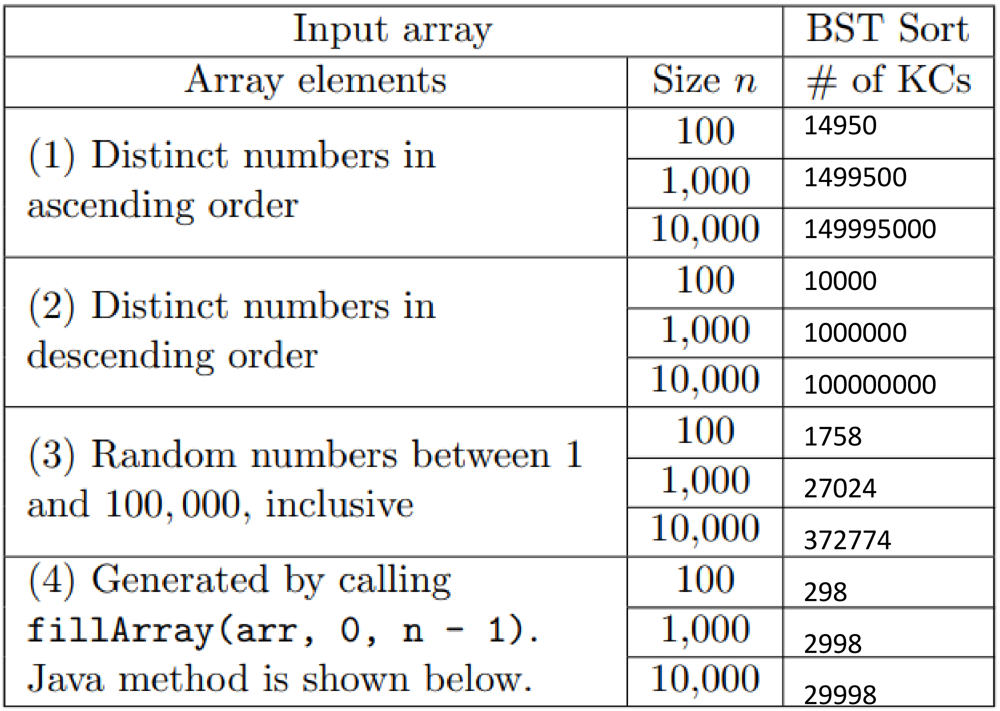
The best case occurs when the element to be deleted is the last element in the heap. the Max-Heap-Increase-key will take constant time as the key is increased to ∞, and then Max-Heap-Extract-max operation takes $\Theta(\log n)$ time as it restores the heap property

worst case scenario $\Theta(\log n)$

The worst case occurs when the element to be deleted is the max element, and it is not the last element, Max-Heap-Increase-key operation take $\Theta(\log n)$ time, and Max-Heap-Extract-max operation also takes $\Theta(\log n)$ time. Therefore, the overall worst case running time is dominated by the Max-Heap-Extract-max operation.

Valentina Silveira
Eagle ID: 901365377

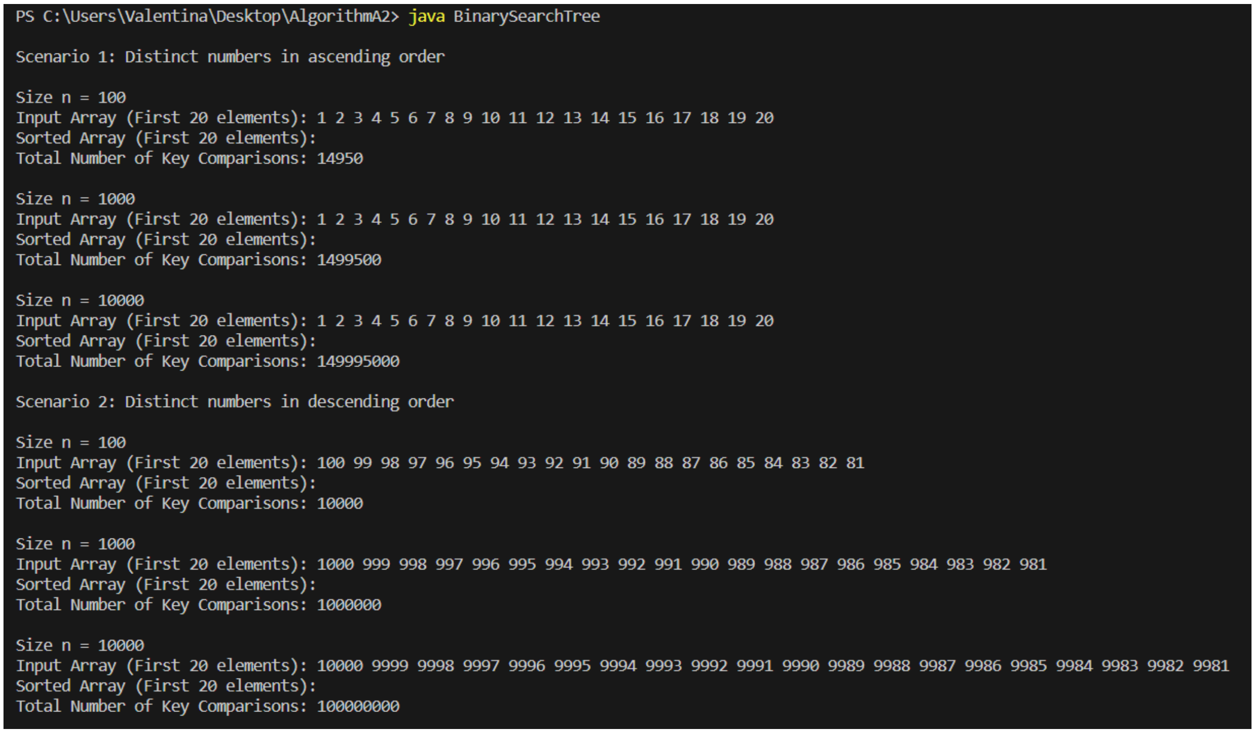
3.



Input array		BST Sort
Array elements	Size n	# of KCs
(1) Distinct numbers in ascending order	100	14950
	1,000	1499500
	10,000	149995000
(2) Distinct numbers in descending order	100	10000
	1,000	1000000
	10,000	100000000
(3) Random numbers between 1 and $100,000$, inclusive	100	1758
	1,000	27024
	10,000	372774
(4) Generated by calling `fillArray(arr, 0, n - 1)`. Java method is shown below.	100	298
	1,000	2998
	10,000	29998

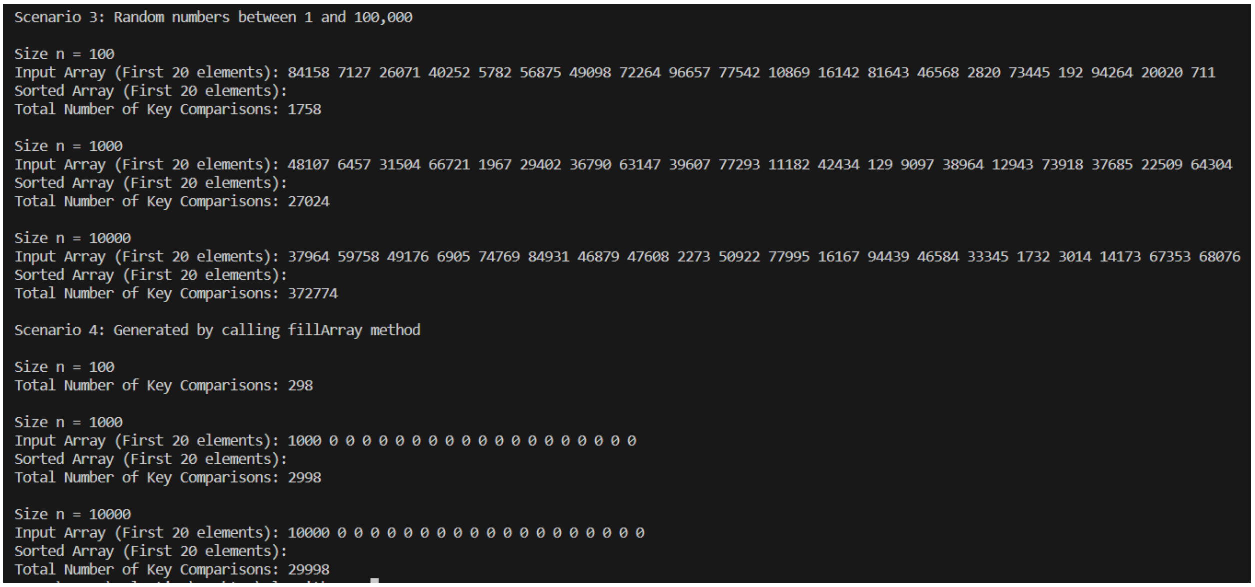
b.

Output:



```
PS C:\Users\Valentina\Desktop\AlgorithmA2> java BinarySearchTree

Scenario 1: Distinct numbers in ascending order

Size n = 100
Input Array (First 20 elements): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 14950

Size n = 1000
Input Array (First 20 elements): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 1499500

Size n = 10000
Input Array (First 20 elements): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 149995000

Scenario 2: Distinct numbers in descending order

Size n = 100
Input Array (First 20 elements): 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 10000

Size n = 1000
Input Array (First 20 elements): 1000 999 998 997 996 995 994 993 992 991 990 989 988 987 986 985 984 983 982 981
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 1000000

Size n = 10000
Input Array (First 20 elements): 10000 9999 9998 9997 9996 9995 9994 9993 9992 9991 9990 9989 9988 9987 9986 9985 9984 9983 9982 9981
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 100000000
```

Valentina Silveira
Eagle ID: 901365377



```
Scenario 3: Random numbers between 1 and 100,000

Size n = 100
Input Array (First 20 elements): 84158 7127 26071 40252 5782 56875 49098 72264 96657 77542 10869 16142 81643 46568 2820 73445 192 94264 20020 711
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 1758

Size n = 1000
Input Array (First 20 elements): 48107 6457 31504 66721 1967 29402 36790 63147 39607 77293 11182 42434 129 9097 38964 12943 73918 37685 22509 64304
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 27024

Size n = 10000
Input Array (First 20 elements): 37964 59758 49176 6905 74769 84931 46879 47608 2273 50922 77995 16167 94439 46584 33345 1732 3014 14173 67353 68076
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 372774

Scenario 4: Generated by calling fillArray method

Size n = 100
Total Number of Key Comparisons: 298

Size n = 1000
Input Array (First 20 elements): 1000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 2998

Size n = 10000
Input Array (First 20 elements): 10000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Sorted Array (First 20 elements):
Total Number of Key Comparisons: 29998
```

c.

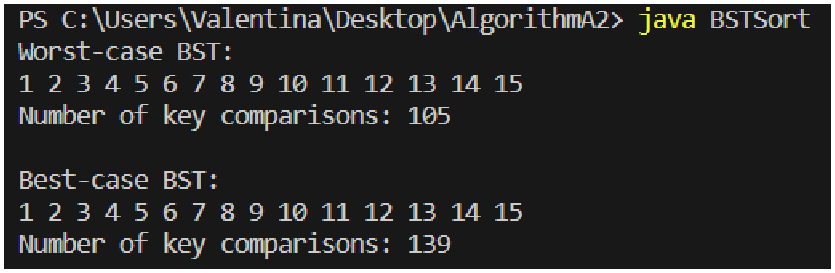
1. Creating Worst-case and Best-case Input Arrays for n = 15

- Worst-case input array (ascending order): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

    - Number of key comparisons: (14 + 13 + 12 + … + 2 + 1 = 105)

- Best-case input array (well-balanced): [8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15]

    - Number of key comparisons: 139



```
PS C:\Users\Valentina\Desktop\AlgorithmA2> java BSTSort
Worst-case BST:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Number of key comparisons: 105

Best-case BST:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Number of key comparisons: 139
```

2. Identifying the Worst-case Scenario Array

From the tests conducted in part b, it's evident that the array sorted in ascending order results in the worst-case scenario. This is because inserting elements in ascending order leads to the creation of a completely unbalanced tree, where each node only has a right child, resulting in a skewed structure.

3. Identifying the Best-case Scenario Array

Among the tested arrays, the well-balanced array (such as [8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15]) results in the best-case scenario. This array leads to the creation of a balanced binary search tree, where the height of the tree is minimized, resulting in fewer key comparisons during insertion.