Detailed system requirements:

BinaryHeap can hold generic types as heap. All the elements in a left subtree and right subtree are smaller than their parents. The tree is a complete tree. Generic types can be added and removed from the BinaryHeap and 2 heap can be merged. Tree structure can be printed to a console in a readible format.

ArrayBST is a binary search tree which holds the nodes of the tree in an ArrayList. In binary search trees there is no duplicate elements, all the elements in a left subtree are smaller than their parents and all the elements in a right subtree are greater than their parents. ArrayBST implements SearchTree interface. Generic types can be added and removed from this tree. An element can be searched to see if it is in the tree. Tree structure can be printed to a console in a readible format.

Theese classes written in Java 11 so it can be run in anyplace that java can run. Program starts and prints the outputs of the methods that shows their functionality and capability of the tree structures then ends.

Test Case:

Element adding and printing the tree after every addition.
Removing an element exist in the tree then printing the tree to show the result.
Removing an element which doesn't exist in the tree then printing the tree to show the result.
Searching for an element exist in the tree
Searching for an element which doesn't exist in the tree

Question 1:

a)

```
   *      1
 *   *    2
* * * *   3
   .
   .
   .
****...   h
```

The total depth of the of the nodes in a complete binary tree of height h is -> 1* 2^0 + 2*2^1 + 3* 2^2 + ... (h-k)*(2^(h-1)) k>=0 && k<h there is -k because a complete binary tree is a perfect binary tree thorugh level h - 1 with some extra leaf nodes at level h all toward the left. These nodes represented with k.

So the answer is: 2+(h-1) *2^(h+1) - h*(h+1) -k*h

b) In complete trees the height(h) of the tree is logn (n equals the number of the noods in the tree). And in binary search trees all nodes are sorted (all nodes in the left branch is smaller than root and all the nodes in the right

branch is larger than root).
The search will be 0(logn) in worst case and 0(1) in best case.

So the answer is O(logn).

c) In full binary tree all nodes have two or no children. This means every leaf has no children and every internal node has two children. It doesn't matter if the tree is perfect tree or not when calculating the number of leaves and internal nodes. There is no restriction on the number of the nodes. The number of leaves = (n + 1)/2 and the number of internal nodes = (n - 1)/2 where n is the number of nodes in the tree.

Question 3:

In binary heaps all the elements in a left subtree and right subtree are smaller than their parents. Binary heap must be complete tree. This BinaryHeap class also extends BinaryTree class. It is exptected to implement this class using node-link structure. To implement it I added Node variable 'up' to the Node inner class so I can traverse the tree more efficiently.
To implement methods I used some helper private methods

Analysis of private helper methods:

private void mergeHelper(Node<E> tree) -> Does pre order traverse recursively and takes every node then inserts them to the binary heap. Calls insert method which calls setLNH method in every recursive call.
worst case: T(n) = 2 * T(n/2) + n -> 0(nlogn) according to master theorem
best case: T(n) = 2 * T(n/2) + logn -> 0(ln)
average case 0(n), Ω(n)

private Node<E> setLIH(Node<E> tree) -> Finds last added item in the binary heap. Calls setLNH method and using the return value of it accesses last added item of the binary heap. If the null node is right branch returns left branch directly. If its left branch then goes to up nodes until it becomes the right branch of an upper node. Then passes the left branch of that upper node and goes to the rightmost branch. The rightmost branch is the last added node to the binary heap. Best case 0(logn), worst case 0(n), average case O(n), Ω(logn) -> comes from setLNH

private Node<E> setLNH(Node<E> tree) -> Returns last node has null branch in the binary heap which is going to be used when adding an item. To access it calculates the depths of the leftmost branch, the rightmost branch and the middle branch. If the depth of the leftmost branch is greater than the others then passes to left branch. If the depths of the leftmost and the rightmost branch is the same then passes the left branch because it means that the binary heap is perfect tree at that moment. If the depths of the leftmost and middle branch is the same then passes the right branch. Does these recursively until it reaches the null node which is going to be used when adding a new element. Best case 0(logn), worst case 0(n), average case O(n), Ω(logn)
Best case 0(logn), worst case 0(n), average case O(n), Ω(logn)

private void swap(Node<E> x, Node<E> y) -> Swaps the datas of x and y. Best case 0(1), worst case 0(1), average case 0(1)

private Node<E> getSmallBranch(Node<E> x) -> Takes one Node as a parameter and returns small branch of it. Best case 0(1), worst case 0(1), average case 0(1)

private void preOrderTraverse (Node<E> node, int depth, StringBuilder sb) -> Pre order traverse function from the book
worst case: T(n) = 2 * T(n/2) + logn -> 0(n) according to master theorem
best case: T(n) = 2 * T(n/2) + n -> 0(nlogn)
average case 0(n)

Analysis of BinaryHeap methods:

public boolean insert(E data) -> Adds one item to the binary heap and returns true. Calls setLNH method and inserts the node that has data value as data parameter to the null branch of the returned node. After that compares the data value with the data values of upper nodes. If the upper node has bigger value then swaps their data values and does this until the data value will be the bigger one.
Best case 0(logn), worst case 0(n), average case O(n), Ω(logn) -> comes from setLNH

public void merge(BinaryHeap<E> tree)- > Merges the binary heap with given parameter tree. Calls mergeHelper method with the root of the tree parameter. mergeHelper does the rest of the work.
worst case: T(n) = 2 * T(n/2) + n -> 0(nlogn) according to master theorem
best case: T(n) = 2 * T(n/2) + logn -> 0(ln)
average case 0(n), Ω(n) -> comes from mergeHelper

public E remove() -> Removes the last item from the binary heap and returns it. To remove it first calls setLIH method and gets last added item to the tree then calls swap method for swapping the datas of the root and last added item. After that sets the last added item (swapped with the root) to null and compares the root node (which has last added item data) with smallest subtrees. Swaps the datas with smaller subtrees until it can't find any smaller subtree. Returns the data which is removed if it is in the tree otherwise false.
Best case 0(logn), worst case 0(n), average case O(n), Ω(logn) -> comes from setLNH

public String toString() -> Calls preOrderTraverse and return the StringBuilder variable which has readible string format of the tree.
worst case: T(n) = 2 * T(n/2) + n -> 0(nlogn) according to master theorem
best case: T(n) = 2 * T(n/2) + logn -> 0(n) according to master theorem
average case O(nlogn), Ω(n) -> same with preOrderTraverse function

Question 4:

In binary search trees there is no duplicate elements, all the elements in a left subtree are smaller than their parents and all the elements in a right subtree are greater than their parents. It is exptected to hold theese elements in an array so I used an ArrayList to hold the tree. Name of the class is ArrayBST and it implements SearchTree interface.
If the index of the root is x then index of the root of the left subtree is 2x+1 and index of the root of the right subtree is 2x+2.
I used some private methods to implement the functions came from SearchTree interface.

Analysis of private helper methods:

private int searchedIndex(E target) -> this functions takes a target value and searches it in the tree. If it finds return its index otherwise returns -1. Best case 0(1), worst case 0(logn), average case O(logn)

private int max(int index) -> finds min value after index (a branch of the tree) and returns its index Best case 0(1), worst case 0(logn), average case O(logn)

private int min(int index) -> finds min value after index (a branch of the tree) and returns its index Best case 0(1), worst case 0(logn), average case O(logn)

private void preOrderTraverse(E node, int depth, StringBuilder sb, int index)-> Pre order traverse function from the book but I adapt it to array implementation.
worst case: T(n) = 2 * T(n/2) + logn -> 0(n) according to master theorem
best case: T(n) = 2 * T(n/2) + logn -> 0(n)
average case 0(n)

Analysis of SearchTree methods implemented in ArrayBST class:

public boolean add(E item) -> Compares item with the data values (if it is small than data looks for left branch which means 2*index +1 in the array and if it is larger than data looks for left branch which means 2*index +2)
of the node of the tree until it finds an appropriate place for it. If item is already in the tree returns false otherwise add is and returns true; Best case 0(1), worst case 0(logn), average case O(logn)

public boolean contains(E target) -> Calls searchedIndex. If it returns -1 returns false otherwise returns true. Best case 0(1), worst case 0(logn), average case O(logn) -> comes from searchedIndex.

public E find(E target) -> Calls searchedIndex. If it returns -1 returns null otherwise index value will be the returned value from searchedIndex and returns the index'th element of the tree array.
Best case 0(1), worst case 0(logn), average case O(logn) -> comes from searchedIndex.

public E delete(E target) -> Calls remove function. If it returns true returns target otherwise returns null. Best case 0(1), worst case 0(logn), average case O(logn) -> same with the remove function.

public boolean remove(E target); Calls searchedIndex. If it returns -1 returns false otherwise index value will be the returned value from searchedIndex. If element at the 2*index + 1 position is not null finds max value of the left subtree with using max function. If it is null finds min value of the right subtree with using min function. Swaps the values of the nodes and sets the min or max node to null then returns true. If both left and right subtree is null its means that target is already a leaf node so there is no need to swapping operation. Sets it to null and returns true;
Best case 0(1), worst case 0(logn), average case O(logn) -> searchedIndex, min and max functions have the same complexities. remove function will have the same complexity with them.

public String toString() -> Calls preOrderTraverse and return the StringBuilder variable which has readible string format of the tree.

worst case: T(n) = 2 * T(n/2) + n -> θ(nlogn) according to master theorem
best case: T(n) = 2 * T(n/2) + logn -> θ(n) according to master theorem
average case O(nlogn), Ω(n) -> same with preOrderTraverse function