CISC2006: Algorithm Design and Analysis
Homework 1 **Due Date:** 23:59, 17 Feb 2023

Rules: Discussion of the problems is permitted, but writing the assignment together is not (i.e. you are not allowed to see the actual pages of another student). You can get at most 100 points if attempting all problems. Please make your answers precise and concise.

1. (10 pts) Give "True" or "False" for each of the following statements.

   No proof is needed.

   - $T(n) = 3n^2 + 5n \cdot \log_2 n = O(n)$.
   - $T(n) = 4^{\log_2 n} + \sqrt{n} = \Omega(n^2)$.
   - $T(n) = 3n^2 + 9n = O(n^3)$.
   - $T(n) = 4 \cdot (\log_2 n)^5 + 5\sqrt{n} + 10 = \Theta(\sqrt{n})$.
   - $T(n) = (\log_2 n)^{\log_2 n} + n^4 = \Theta(n^4)$.

   Solution:

   - **False**
   - **True**
   - **True**
   - **True**
   - **False**

2. (10 pts) Given a sequence of $A = \{a_1, a_2, \ldots, a_n\}$ of n integers, where $a_1 \le a_2 \le \ldots \le a_n$ and another integer K, give an algorithm that outputs

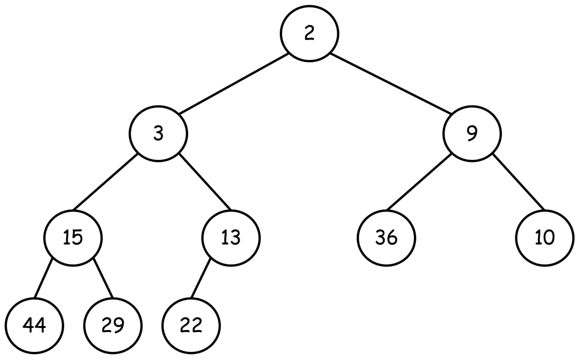
   - three different $i, j, k \in \{1, 2, \ldots, n\}$ such that $a_i + a_j + a_k = K$, or
   - "do not exist" if they don't exist.

   Complete the missing steps in the pseudo-code of the algorithm.

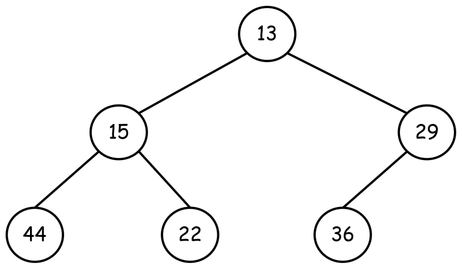
```
Algorithm 1: Sum_of_three($A$, $K$)
1  let $n \leftarrow |A|$ and assume $A = \{a_1, a_2, \ldots, a_n\}$.
2  for $i = 1, 2, \ldots, n-2$ do
3      $j \leftarrow i + 1$ and $k \leftarrow n$.
4      while $j < k$ do
5          if $a_i + a_j + a_k = K$ then
6              Output: $(i, j, k)$.
7          else if $a_i + a_j + a_k < K$ then
8              $j \leftarrow j + 1$.
9          else if $a_i + a_j + a_k > K$ then
10             $k \leftarrow k - 1$.
11 Output: "do not exist".
```

3. (20 pts) Draw the heap (as a binary tree) after executing the following operations on an initially empty heap (you do not need to show the intermediate steps):

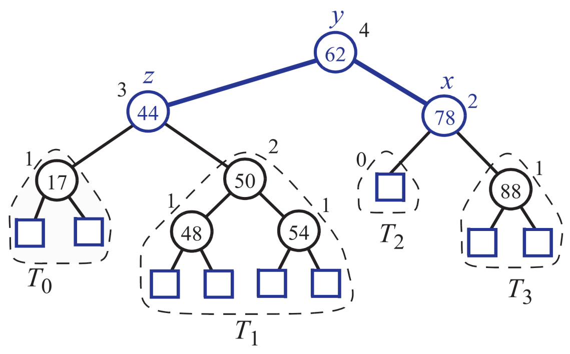
- insertions of elements $22, 15, 36, 44, 10, 3, 9, 13, 29, 2$ (inserted one-by-one);
  Solution:



- removeMin() four times.
  Solution:

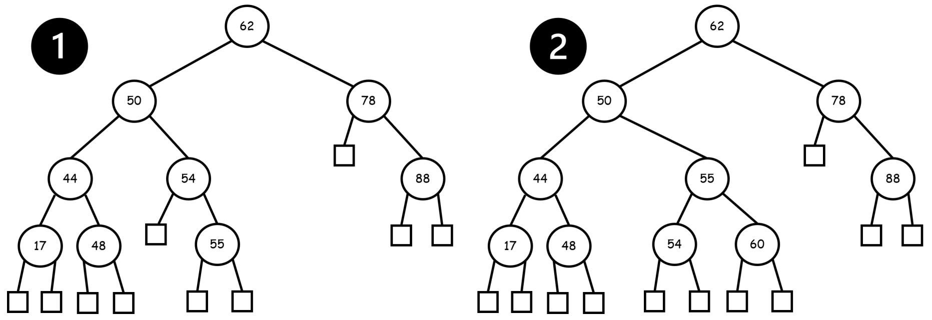


4. (20 pts) Draw the AVL tree after executing each of the following operations on:



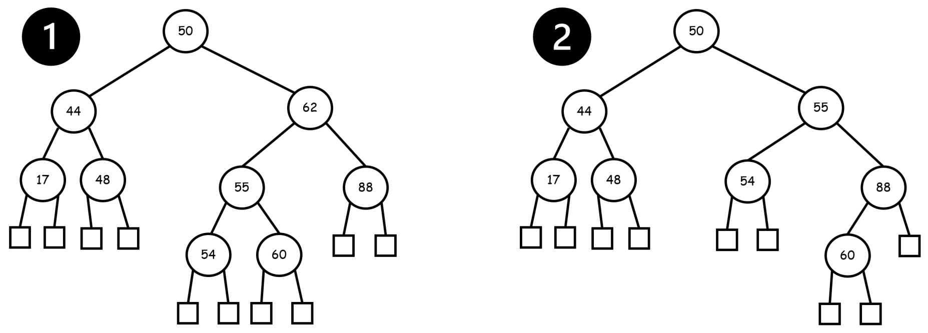
- insertion of element 55, then insertion of element 60;

Solution:



- removal of element 78, then removal of element 62.

Solution:



5. (10 pts) You are given an implementation of binary search tree (e.g., AVL tree) that supports find(k) (finding the element with key $= k$) in $O(\log n)$ time, where n is the total number of elements. Design a function findAll(k_1, k_2) to find all elements with keys in $[k_1, k_2]$ in $O(\log n + s)$ time, where s is the output size.

   Present your algorithm in pseudocode, prove its correctness and analyze its complexity. You can assume that all elements have different integer key values.

   Solution: The following function outputs all elements with key in $[k_1, k_2]$ located in the sub-tree rooted at v. To answer the query of findAll(k_1, k_2), we run findAll(k_1, k_2, r), where r is the root of the tree.

   Algorithm 2: findAll(k_1, k_2, v)

```
1  if $v = null$ then
2      terminate
3  else if $v.key \in [k_1, k_2]$ then
4      print: $v$.
5      findAll($k_1$, $v.key$, $v.left$)
6      findAll($v.key$, $k_2$, $v.right$)
7  else if $v.key > k_2$ then
8      findAll($k_1$, $k_2$, $v.left$)
9  else if $v.key < k_1$ then
10     findAll($k_1$, $k_2$, $v.right$)
```

   Correctness. The correctness of the algorithm follows from the fact that the left sub-tree of $T(v)$ is not searched only if $v.key < k_1$; the right sub-tree of $T(v)$ is not searched only if $v.key > k_2$. In other words, as long as there exists an element u with key value in $[k_1, k_2]$, the subtree rooted at u will be searched.

   Complexity. Observe that excluding the recursive calls, the running time for each call to function findAll() is $O(1)$. Hence to show that the query time is $O(\log n + s)$, it suffices to argue that there are at most $O(\log n + s)$ calls to the function. We classify the calls into two types: Type-A are the calls in line 5-6; Type-B are those in line 8 and line 10. Since each Type-A call happens only if some key value in $[k_1, k_2]$ is found and output, there are at most $2s$ Type-A calls (recall that s is the output size). Next, we argue that in each level, there are at most 2 Type-B calls. Suppose otherwise and let $u_1, u_2, \ldots, u_t$ be the nodes on the same level of the tree on which a Type-B call of findAll() is executed, where $t \geq 3$ and the keys of $u_1, u_2, \ldots, u_t$ are strictly increasing. Let $p_1, p_2, \ldots, p_t$ be the parents of these nodes. Since the t calls are of Type-B, for each $i \leq t$ we have either $p_i.key > k_2$ or $p_i.key < k_1$. Since $t \geq 3$, there exists p_i, p_j satisfying the same inequality, say, $p_1.key < k_1$ and $p_2.key < k_1$. Then we have the contradiction because when the function is call on the lowest common ancestor of p_1 and p_2 (whose key value is also less than k_1), only one Type-B call will happen on the right sub-tree. In other words, the call on node p_1 will not happen. Since there are at most $O(\log n)$ levels, the total number of Type-B calls is $O(\log n)$.

6. (20 pts) **Different implementations of Priority Queue**.

   Implement a priority queue data structure class to store a collection of different numbers that supports the following operations:

   - insert(e) : insert an element e into the priority queue;
   - min() : return the minimum element in the priority queue;
   - removeMin() : remove the minimum element in the priority queue;
   - size() : return the total number of elements in the priority queue;
   - isEmpty() : return True if the priority queue is empty; False otherwise.
   - printPQ() : list all elements in the priority queue. For a heap, list the elements from top-level to bottom-level, and for each level from left to right.

   Use the following data structures for three different implementations:

   - Unsorted Doubly Linked List;
   - Sorted Doubly Linked List;
   - Heap (implemented using an array).

   In the main function, we read an array $A = \{a_1, a_2, \ldots, a_n\}$ of n different integers, and use the three different implementations of priority queue to do sorting.

   In particular, we do the following for each version of priority queues.

   We first initialize a priority queue, which is empty. Then we insert the numbers in A one-by-one, and sort the numbers into another array $B = \{b_1, b_2, \ldots, b_n\}$ for output by repeatedly calling min() and removeMin().

7. (30 pts) **Different implementations of Binary Search Tree**.

   Implement a binary search tree (BST) data structure class to store a collection of different numbers that supports that following operations

   - insert(e) : inset element e into the BST;
   - find(k) : return the pointer that points to an element with key $= k$; if no such element exists, return Null;
   - remove(k) : remove the element with key $= k$ if such element exists;
   - remove(p) : remove the element pointed by pointer p;
   - size() : return the total number of elements in the BST;
   - isEmpty() : return True if the BST is empty; False otherwise.
   - printTree() : print the whole BST (use indentation to show the structure).

   Use the following data structures for the two different implementations:

   - Binary tree without height balance property;
   - AVL tree.

   In the main function, we will read an array $A = \{a_1, a_2, \ldots, a_n\}$ of n different integers, and insert the numbers into the two different implementations of BST one-by-one. Then we read another array $B \subseteq A$ of integers, each of which appeared in A, and remove the integers in B from the BST.

   Finally, we output the resulting BST using printTree().