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.

c logn √n n nlogn n² n³ ... n^c c^n n!

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

   No proof is needed.

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

1) False

2) True

3) True

4) True

5) False

definition

↳ $T(n) = O(f(n))$, $\exists N$, for $n \geq N$, $T(n) \leq c f(n)$

↳ $T(n) = \Omega(f(n))$, $\exists N$, for $n \geq N$ $T(n) \geq c f(n)$.

ascending

2. (10 pts) Given a sequence of $A = \{a_1, a_2, \ldots, a_n\}$ of n integers, where $a_1 \leq a_2 \leq \ldots \leq a_n$ 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.

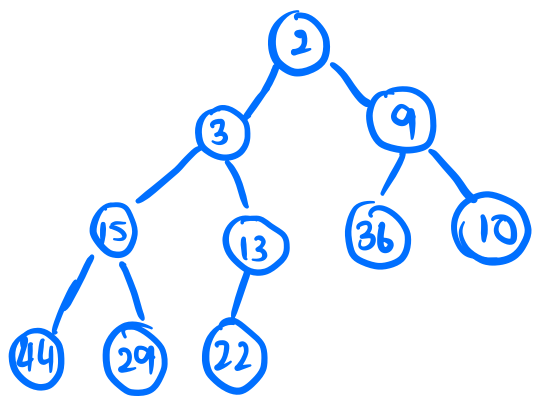
Algorithm 1: Sum_of_three(A, K)

```
let n ← |A| and assume A = {a_1, a_2, ..., a_n}.
for i = 1, 2, ..., n − 2 do
    j ← i + 1 and k ← n.
    while  j<k  do
        if a_i + a_j + a_k = K then
            Output: (i, j, k).
        else if a_i + a_j + a_k < K then
            j=j+1
        else if a_i + a_j + a_k > K then
            k=k-1
Output: "do not exist"
```

① complete binary ② heap property (parent < child)

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):

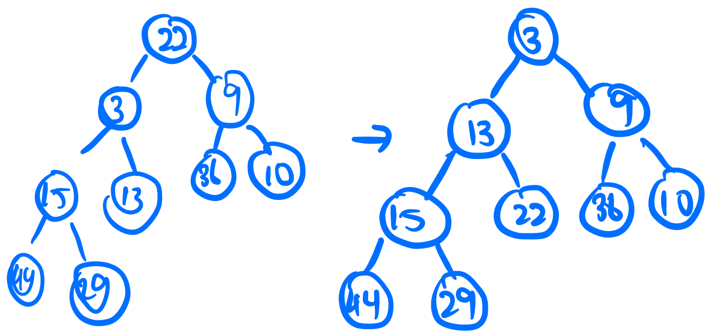
- insertions of elements 22, 15, 36, 44, 10, 3, 9, 13, 29, 2 (inserted one-by-one);



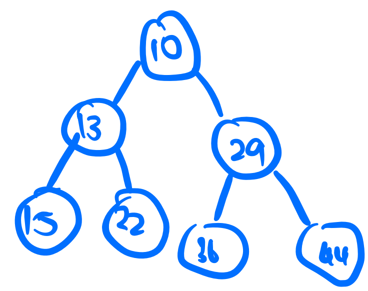
upbubbling

- removeMin() four times.

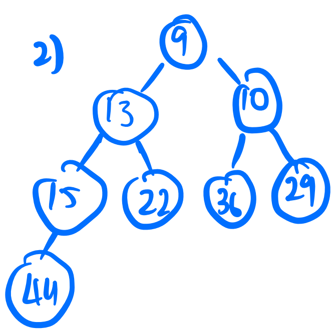
1)



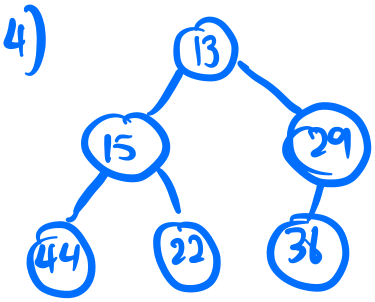
3)



2)



4)



down bubbling

↳ left ≤ middle ≤ right

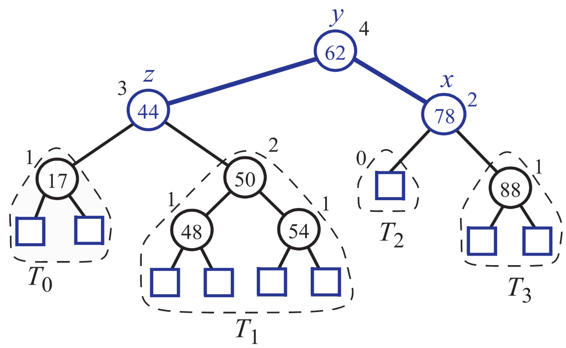
↳ height balance property.

insertion : insert like normal

identify z, y, x.

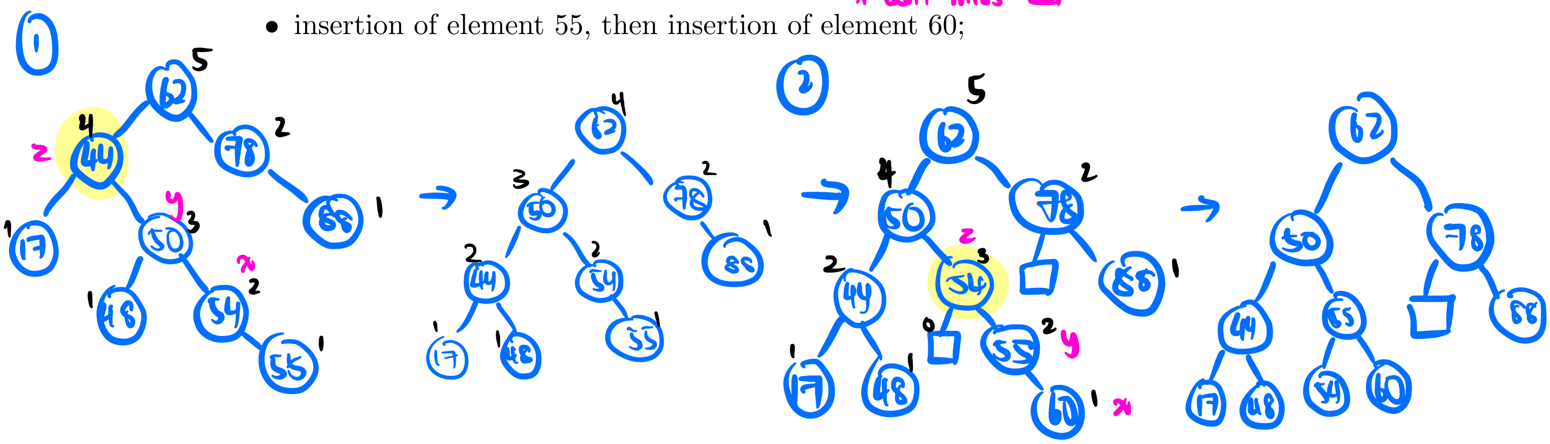
T1, 2, 3, 4

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

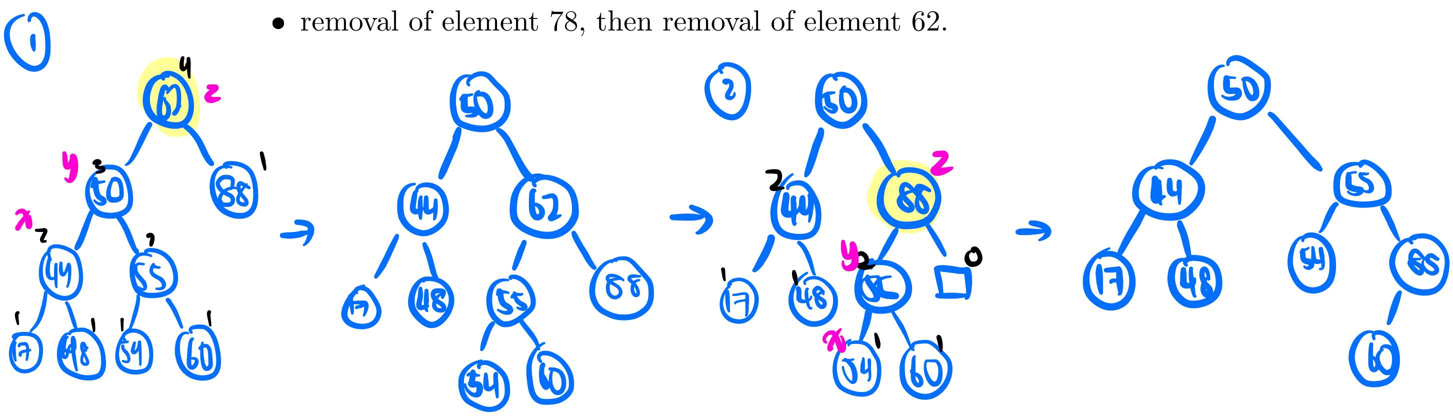


* don't miss □

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



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



remove : direct replace

or

(min of right child)

fix unbalance

(z : lowest unbalance)

(y : another child)

(x : longer child / same side)

(T1,2,3,4)

BST property

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.

① Present your algorithm in pseudocode, ② prove its correctness and analyze its complexity. You can assume that all elements have different integer key values.

known : Find (k) function with O(log n) complexity

my thoughts :

1) find k_1
2) all right subtree of k_1 is needed for output.
3) find k_2
4) all left subtree of k_2 is needed for output

```
Algorithm : findAll (k1, k2, r) :

 if  r = null ,
     output : "do not exits",
 else if   k1 ≤ r.value ≤ k2 ,
     output : r
     findAll (k1, k2, r.left)
     findAll (k1, k2, r.right).
 else if  r.value < k1 ,
     findAll (k1, k2 , r.right)
 else if  r.value > k2 ,
     findAll (k1, k2, r.left).
```

proof of correctness

We know that k1 < k2,

Therefore, if node value is smaller than k1, then it's left subtree must not be in the range of [k1,k2], therefore, we no need search it, we just have to focus on the right subtree.
Similarly, if node value is bigger than k2, then its right subtree must not be in range of [k1,k2]. We just have to search left subtree.
This is because of the property of Binary Search Tree, where left <= middle <= right.

And when value is between k1 and k2, that's the result we want, so we output it, at the same time, search it's left subtree and right subtree as both of them have the chance to be in range.

Analyze complexity

We need to prove that the algorithm's complexity is O(log n +s) time.
In the algorithm, we are using recursion.
For each node, they will only have one of the case due to property of binary search tree:

1. r= null
2. K1 <= r.value <= K2
3. r.value < k1
4. r.value > k2

For case(1), it takes O(1) time
For (3) and (4) are similar, we can discuss together. When (3) or (4) happen, only one recursion will happen, therefore, at most there is o(logn) time, as we know log n is the height of tree (prove in lecture previously)
For case (2), there are at most 2s recursion call (s is the output size), complexity will be T(log n +2s) = O(log n + s)

Therefore O(1) + O(log n) + O(log n +s) = O(log n +s)

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().