1a.

```
Each account a has:
   stack of amounts
   count[0..T]

Insert(a, t):
   push t onto account[a].stack
   count[a][t] = count[a][t] + 1

Search(a, t):
   if count[a][t] > 0 then
      return true
   else
      return false

Most_recent(a):
   if stack is empty then
      return NULL
   else
      return top value of stack

Delete_most_recent(a):
   if stack is empty then
      return
   t = pop from stack
   count[a][t] = count[a][t] - 1
```

Insert(a, t):
When we add a transaction, we push t onto the stack and increment count[a][t]. Both steps take constant time.
Time complexity: O(1)

Search(a, t):
To check if an amount t exists, we simply test if count[a][t] > 0. Array access is constant time.
Time complexity: O(1)

Most_recent(a):
To find the most recent transaction, we return the top element of the stack. Accessing the top of a stack is constant time.
Time complexity: O(1)

Delete_most_recent(a):
To delete the most recent transaction, we pop the top of the stack and decrement count[a][t]. Both operations are constant time.
Time complexity: O(1)

1b.

```
Each account a has:
   stack of amounts
   balanced BST (amount → frequency)

Insert(a, t):
   push t onto account[a].stack
   if t exists in BST then
      freq[t] = freq[t] + 1
   else
      insert t into BST with freq = 1

Search(a, t):
   if t exists in BST then
      return true
   else
      return false

Most_recent(a):
   if stack is empty then
      return NULL
   else
      return top value of stack

Delete_most_recent(a):
   if stack is empty then
      return
   t = pop from stack
   freq[t] = freq[t] - 1
   if freq[t] == 0 then
      remove t from BST
```

INSERT(a, t):
Push t onto the stack (O(1)), then search for t in the BST. If found, increment its frequency; otherwise, insert a new node. Searching and inserting in a BST both take O(log n).
Time complexity: O(log n)

Search(a, t):
Look up t in the BST. Searching in a balanced BST takes O(log n).
Time complexity: O(log n)

Most_recent(a):
Return the top element of the stack. Stack top lookup takes constant time.
Time complexity: O(1)

Delete_most_recent(a):
Pop the top of the stack (O(1)), then search for that amount t in the BST and decrement its frequency. If the frequency reaches zero, remove the node from the BST. Both searching and deleting in a BST take O(log n).
Time complexity: O(log n)

2a.

```
Build_bst(arr, start, end):
  if start > end then
    return NULL

  mid = (start + end) / 2
  root = new Node(arr[mid])

  root.left = Build_bst(arr, start, mid - 1)
  root.right = Build_bst(arr, mid + 1, end)

  return root
```

2b.

```
Merge_trees(T1, T2):
  list1 = Inorder(T1)
  list2 = Inorder(T2)

  merged = Merge_sorted(list1, list2)

  root = Build_bst(merged, 0, length(merged) - 1)
```

```
    return root


Inorder(root):
    if root = NULL:
        return []
    return Inorder(root.left) + [root.key] + Inorder(root.right)


Merge_sorted(list1, list2):
    i = 0, j = 0
    merged = []
    while i < length(list1) and j < length(list2):
        if list1[i] < list2[j]:
            merged.append(list1[i])
            i = i + 1
        else:
            merged.append(list2[j])
            j = j + 1
    append remaining elements of list1 or list2
    return merged
```