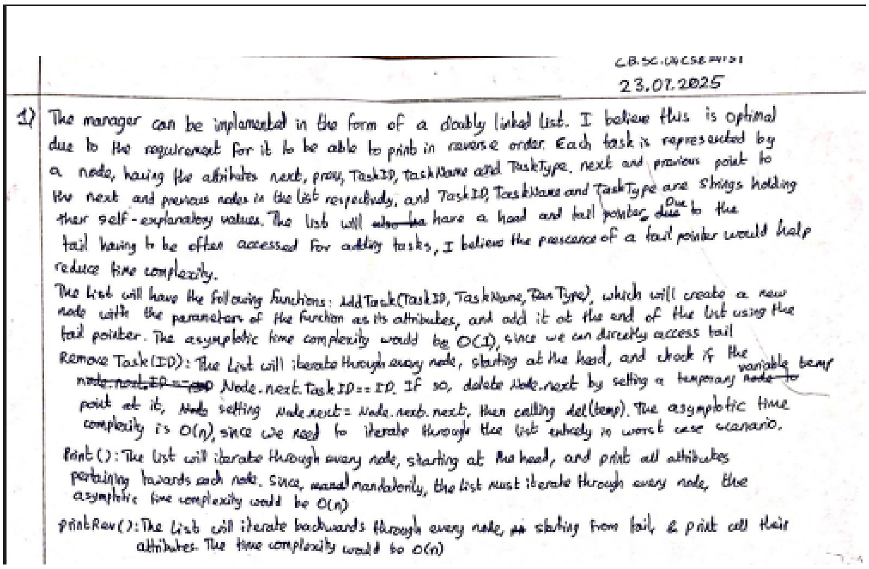
DSA Report

Sudharshan Sambathkumar
CB.SC.U4CSE24151

Question 1: Task Manager via Doubly Linked List



CB.SC.U4CSE24151
23.07.2025

1) The manager can be implemented in the form of a doubly linked list. I believe this is optimal due to the requirement for it to be able to print in reverse order. Each task is represented by a node, having the attributes next, prev, TaskID, taskName and TaskType. next and previous point to the next and previous nodes in the list respectively, and TaskID, TaskName and TaskType are strings holding their self-explanatory values. The list will ~~also be~~ have a head and tail pointer. ~~due~~ Due to the tail having to be often accessed for adding tasks, I believe the presence of a tail pointer would help reduce time complexity.

The list will have the following functions: AddTask(TaskID, TaskName, TaskType), which will create a new node with the parameters of the function as its attributes, and add it at the end of the list using the tail pointer. The asymptotic time complexity would be O(1), since we can directly access tail

RemoveTask(ID): The list will iterate through every node, starting at the head, and check if the ~~node.next.ID == ID~~ Node.next.TaskID==ID. If so, delete Node.next by setting a temporary variable temp ~~node to~~ point at it, ~~Node~~ setting Node.next = Node.next.next, then calling del(temp). The asymptotic time complexity is O(n), since we need to iterate through the list entirely in worst case scenario.

Print(): The list will iterate through every node, starting at the head, and print all attributes pertaining towards each node. Since, ~~mand~~ mandatorily, the list must iterate through every node, the asymptotic time complexity would be O(n)

printRev(): The list will iterate backwards through every node, starting from tail, & print all their attributes. The time complexity would be O(n)

Figure 1: Initial solution pseudo code.

AI usage declaration:

Used AI as a search engine for general brainstorming

Time Complexities

- **Add Task (taskId, taskName, taskType):** $O(1)$
  A new node is created and directly appended to the tail node. this makes the function independent of list size.
- **Remove Task (taskId):** $O(n)$
  The list begins at the head node and iterates through every node to find the node to delete, and then deletes it. In worst case scenario where the node to be deleted is last, or if the id doesn't exist in the list, then the function must iterate through every node. This method is hence O(n) time complexity
- **Print():** $O(n)$
  Each node is visited, starting at the head node and iterating forward until the tail node, and their attributes are printed. This makes it O(n) time complexity
- **Print Reverse():** $O(n)$
  A pointer is initialized at the tail pointer, and moves back through the list using the prev pointer of each node, printing each node's attributes. If it reaches the head node, it terminates. Hence, it is O(n) time complexity.

Pseudo code

```
Class: Task
Properties:
    taskId
    taskName
    taskType
    next
    prev

Constructor(taskId, taskName, taskType):
    Set taskId, taskName, taskType
    next ← null
    prev ← null


Class: Manager
Properties:
    head
    tail

Constructor():
    head ← null
    tail ← null

Method: add_task(taskId, taskName, taskType)
    Create new Task node
    If head is null:
        head ← newTask
        tail ← newTask
    Else:
        tail.next ← newTask
        newTask.prev ← tail
        tail ← newTask
    Print task added

Method: remove_task(taskId)
    current ← head
    While current is not null:
        If current.taskId equals taskId:
            If current is head:
                head ← current.next
                If head is not null:
                    head.prev ← null
            Else if current is tail:
                tail ← current.prev
                If tail is not null:
                    tail.next ← null
            Else:
                current.prev.next ← current.next
                current.next.prev ← current.prev
            Print task deleted
            Exit loop
        Move to next node
    Print task not found (if not deleted)

Method: print_tasks()
    Print "forward direction"
    current ← head
    While current is not null:
        Print taskId, taskName, taskType
        current ← current.next

Method: print_tasks_reverse()
    Print "reverse direction"
    current ← tail
    While current is not null:
        Print taskId, taskName, taskType
        current ← current.prev
```

Evolution of Solution

When I asked ChatGPT(my usual choice of generative AI for browsing) it initially suggested using a hash map. But due to hash map's higher space usage, i wasn't inclined to model my solution with it. Hence, I persisted with my original proposition from the writing section, and after a bit of rumination and peer discussion, i deemed it as reasonable.

Test Cases

Note that the driver used takes in a .txt file, making these large test cases convenient to execute.

Test Case 1

```
A 1 Steins;Gate anime
A 2 DSA_questions study
A 3 小市民シリーズ anime
A 4 DBMS_notes study
A 5 君の名は anime
A 6 AI_paper study
A 7 Blockchain101 study
A 8 おれがいる anime
A 9 CV_notes study
A 10 死ノート anime
P
F 6
F 100
R 2
R 8
R 99
PR
S anime
S study
P
```

Test Case 2

```
A 1 Steins;Gate anime
A 2 DSA_questions anime
A 3 小市民シリーズ anime
A 4 DBMSMocktest study
A 5 天気の子 anime
A 6 サニーボーイ anime
A 7 Notes study
A 8 暗号理論 anime
A 9 ココロコネクト anime
A 10 MachineLearning study
A 11 ゾッム百 anime
R 99
R 7
F 1
F 9
F 100
P
A 5 天気の子 anime
A 12 ヨルシカ anime
A 13 日本語を練習しろ study
A 14 トモダチゲーム anime
A 15 演習問題集 study
PR
S anime
S study
P
```

Test Case 3

```
A 100 Title_100 anime
A 101 Title_101 study
A 102 Title_102 study
A 103 Title_103 anime
A 104 Title_104 anime
A 105 Title_105 study
A 106 Title_106 study
A 107 Title_107 anime
A 108 Title_108 anime
A 109 Title_109 study
A 110 Title_110 study
A 111 Title_111 anime
A 112 Title_112 anime
A 113 Title_113 study
A 114 Title_114 study
A 115 Title_115 anime
A 116 Title_116 anime
A 117 Title_117 study
A 118 Title_118 study
A 119 Title_119 anime
A 120 Title_120 anime
A 121 Title_121 study
A 122 Title_122 study
A 123 Title_123 anime
A 124 Title_124 anime
```

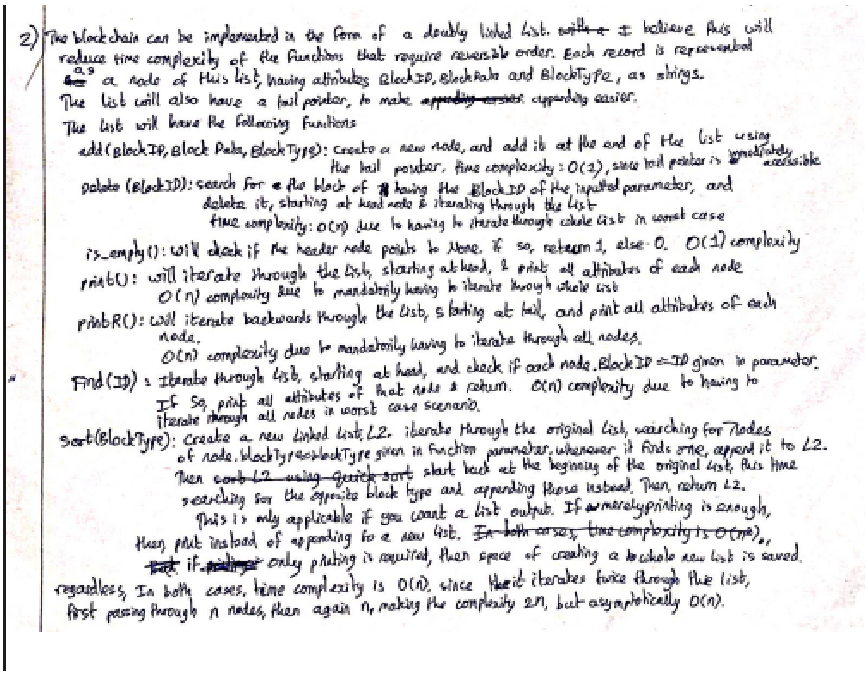
```
A 125 Title_125 study
A 126 Title_126 study
A 127 Title_127 anime
A 128 Title_128 anime
A 129 Title_129 study
A 130 Title_130 study
A 131 Title_131 anime
A 132 Title_132 anime
A 133 Title_133 study
A 134 Title_134 study
A 135 Title_135 anime
A 136 Title_136 anime
A 137 Title_137 study
A 138 Title_138 study
A 139 Title_139 anime
A 140 Title_140 anime
A 141 Title_141 study
A 142 Title_142 study
A 143 Title_143 anime
A 144 Title_144 anime
A 145 Title_145 study
A 146 Title_146 study
A 147 Title_147 anime
A 148 Title_148 anime
A 149 Title_149 study
A 150 Title_150 study
A 151 Title_151 anime
A 152 Title_152 anime
A 153 Title_153 study
A 154 Title_154 study
A 155 Title_155 anime
A 156 Title_156 anime
A 157 Title_157 study
A 158 Title_158 study
A 159 Title_159 anime
A 160 Title_160 anime
A 161 Title_161 study
A 162 Title_162 study
A 163 Title_163 anime
A 164 Title_164 anime
A 165 Title_165 study
A 166 Title_166 study
A 167 Title_167 anime
A 168 Title_168 anime
A 169 Title_169 study
A 170 Title_170 study
A 171 Title_171 anime
A 172 Title_172 anime
A 173 Title_173 study
A 174 Title_174 study
A 175 Title_175 anime
A 176 Title_176 anime
A 177 Title_177 study
A 178 Title_178 study
A 179 Title_179 anime
A 180 Title_180 anime
A 181 Title_181 study
A 182 Title_182 study
```

```
A 183 Title_183 anime
A 184 Title_184 anime
A 185 Title_185 study
A 186 Title_186 study
A 187 Title_187 anime
A 188 Title_188 anime
A 189 Title_189 study
A 190 Title_190 study
A 191 Title_191 anime
A 192 Title_192 anime
A 193 Title_193 study
A 194 Title_194 study
A 195 Title_195 anime
A 196 Title_196 anime
A 197 Title_197 study
A 198 Title_198 study
A 199 Title_199 anime
F 100
F 150
F 199
F 888
R 101
R 111
R 121
R 131
R 141
R 151
R 161
R 171
R 181
R 191
R 1000
P
PR
S anime
S study
P
```

Final Code

```python
class Manager:

    #implemented as a doubly linked list, with both head and tail pointers
    class Task:
        #each task is a node in the doubly linked list
        def __init__(self, task_id, name, task_type):
            self.taskId = task_id
            self.taskName = name
            self.taskType = task_type
            self.next = None
            self.prev = None

    def __init__(self):
        self.head = None
        self.tail = None

    def add_task(self, task_id, name, task_type):
        #adds a new task to the end of the list. time complexity: O(1)
        newTask = self.Task(task_id, name, task_type)

        if self.head == None:
            # First task in the list
            self.head = self.tail = newTask
        else:
            # Append to the end and update tail
            self.tail.next = newTask
            newTask.prev = self.tail
            self.tail = newTask
        print(f"Added␣Task:␣TaskID:␣{newTask.taskId},␣TaskName:␣{newTask.taskName},␣TaskType:␣{newTask.taskType}␣")

    def remove_task(self, task_id):

        #removes the task with the specified ID from the list. time complexity: O(n)
        currentTask = self.head

        currentTask = self.head
        while currentTask is not None:
            if currentTask.taskId == task_id:
                # Case 1: Deleting the head
                if currentTask == self.head:
                    self.head = currentTask.next
                    if self.head:
                        self.head.prev = None

                # Case 2: Deleting the tail
                elif currentTask == self.tail:
                    self.tail = currentTask.prev
                    if self.tail:
                        self.tail.next = None

                # Case 3: Deleting a middle node
                else:
                    currentTask.prev.next = currentTask.next
                    currentTask.next.prev = currentTask.prev
                print(f"Deleted:␣TaskID:␣{currentTask.taskId},␣TaskName:␣{currentTask.taskName},␣TaskType:␣{currentTask.taskType}")
                del currentTask
                break  #deleting only first instance
            currentTask = currentTask.next
        print(f"Could␣not␣delete␣id:{task_id}␣␣␣ID␣not␣found")



    def print_tasks(self):
        #prints all tasks in forward order. time complexity: O(n)
        print("------Printing␣in␣forward␣direction-------")
        currentTask = self.head
        while currentTask!=None:
            print(f"TaskID:␣{currentTask.taskId},␣TaskName:␣{currentTask.taskName},␣TaskType:␣{currentTask.taskType}")
            currentTask = currentTask.next

    def print_tasks_reverse(self):
        #prints all tasks in reverse order. time complexity: O(n)
        print("------Printing␣in␣reverse␣direction-------")

        currentTask = self.tail
        while currentTask!=None:
            print(f"TaskID:␣{currentTask.taskId},␣TaskName:␣{currentTask.taskName},␣TaskType:␣{currentTask.taskType}")
            currentTask = currentTask.prev



def main():
    manager = Manager()
    filePath = "testCase3.txt"
    try:
        file = open(filePath,"r", encoding="utf-8")
    except:
        print(f"File␣not␣found:␣{filePath}")
        return
    with file:
        for line in file:
            parts = line.strip().split()

            if not parts:
                continue  # skip empty lines

            command = parts[0]

            if command == "A" and len(parts) == 4:
                # Eg: A 1 steins;gate anime (create task with id 1, name steins;gate, and type anime)
                task_id = parts[1]
                name = parts[2]
                task_type = parts[3]
                manager.add_task(task_id, name, task_type)

            elif command == "R" and len(parts) == 2:
                # Eg: R 1 (removes task with id 1)
                task_id = parts[1]
                manager.remove_task(task_id)

            elif command == "P":
                manager.print_tasks()

            elif command == "PR":
                manager.print_tasks_reverse()

            else:
                print(f"Invalid␣command:␣{line.strip()}")
main()
```

Question 2: Blockchain via Doubly Linked List



2) The blockchain can be implemented in the form of a doubly linked list. ~~with a~~ I believe this will reduce time complexity of the functions that require reversible order. Each record is represented as a node of this list, having attributes BlockID, BlockData and BlockType, as strings.
The list will also have a tail pointer, to make ~~appending easier~~ appending easier.
The list will have the following funtions

```
add(BlockID, Block Data, BlockType): Create a new node, and add it at the end of the list using
                                     the tail pointer. time complexity: O(1), since tail pointer is immediately accessible
Delete (BlockID): search for the block of having the BlockID of the inputted parameter, and
                  delete it, starting at head node & iterating through the list
                  time complexity: O(n) due to having to iterate through whole list in worst case
is_empty(): will check if the header node points to None. If so, return 1, else 0.  O(1) complexity
print(): will iterate through the list, starting at head, & print all attributes of each node
         O(n) complexity due to mandatorily having to iterate through whole list
printR(): will iterate backwards through the list, starting at tail, and print all attributes of each
          node.
          O(n) complexity due to mandatorily having to iterate through all nodes.
Find(ID): Iterate through list, starting at head, and check if each node.BlockID == ID given in parameter.
          If So, print all attributes of that node & return.  O(n) complexity due to having to
          iterate through all nodes in worst case scenario.
Sort(BlockType): Create a new linked list, L2. iterate through the original list, searching for nodes
                 of node.blockType==blockType given in function parameter. whenever it finds one, append it to L2.
                 Then ~~sort L2 using Quick sort~~ start back at the beginning of the original list, this time
                 searching for the opposite block type and appending those instead. Then, return L2.
                   This is only applicable if you want a list output. If merely printing is enough,
            then print instead of appending to a new list. ~~In both cases, time complexity is O(n²)~~,
            if only printing is required, then space of creating a whole new list is saved.
regardless, In both cases, time complexity is O(n), since it iterates twice through the list,
first passing through n nodes, then again n, making the complexity 2n, but asymptotically O(n).
```

Figure 2: Initial solution pseudo code.

AI usage declaration:

Used AI as a search engine for general brainstorming

Time Complexities

- **Add (id, data, type):** $O(1)$
  Since there is a tail pointer directly available, to add a new node, it is as trivial as simple appending it to the tail. No traversal or iteration needed, hence it is merely O(1) time complexity

- **Delete (id):** $O(n)$
  Searching for the node that contains the matching ID requires iteration through the list. In the worst case scenario that the matching id is in the last node, or that there doesn't exist a node with a matching ID, the function must mandatorily traverse the entire list, making the time complexity O(n)

- **is_empty():** $O(1)$
  By simply checking if the head and tail pointers point to the same object, it is possible to check if the list is empty or not in O(1) time complexity, since again no iteration or traversal is required, making the function independent of list size.

- **Print():** $O(n)$
  traversal through the list and printing all attributes is a mandatory step, making the time complexity of this function O(n)

- **Print Reverse():** $O(n)$
  Similar to the print function, but instead starting at the tail node and traversing backwards. Hence, since they both fundamentally perform the same operations, this function too has a time complexity of O(n)

- **Find (id):** $O(n)$
  Similar to Delete function, in that the entire list must be traversed in the worst case scenario that the node to be found doesnt exist, or exists at the end. Hence, this shares a time complexity of O(n)

- **Sort by Type:** $O(n)$
  We traverse the list once, each time detaching each node and re-hooking its own next/prev pointers into one of two sublists, based on the node's type. Apart from this, other operations, such as linking the 2 lists, are entirely independent on list size. Hence, it is O(n) time complexity, though with the caveat that we are creating a new lists which wastes space.

Pseudo code of final solution

```
Class: Block
    Properties:
        id
        data
        type
        next
        prev

    Constructor(id, data, type):
        Set id, data, type
        next ← null
        prev ← null

Class: Blockchain
    Properties:
        head
        tail

    Constructor():
        head ← null
        tail ← null

Method: is_empty()
    Return head == null

Method: add(blockId, data, blockType)
    Create newBlock ← Block(blockId, data, blockType)
    If head is null:
        head ← newBlock
        tail ← newBlock
    Else:
        tail.next ← newBlock
        newBlock.prev ← tail
        tail ← newBlock
    Print "Block added"

Method: delete(blockId)
    current ← head
    While current is not null:
        If current.id equals blockId:
            If current is head:
                head ← current.next
                If head is not null:
                    head.prev ← null
            Else if current is tail:
                tail ← current.prev
                If tail is not null:
                    tail.next ← null
            Else:
                current.prev.next ← current.next
                current.next.prev ← current.prev
            Print "Block deleted"
            Return true
        current ← current.next
    Print "Block not found"
    Return false

Method: find(blockId)
    current ← head
    While current is not null:
        If current.id equals blockId:
            Print current.id, current.data, current.type
            Return current
        current ← current.next
    Return null

Method: print_chain()
    current ← head
    While current is not null:
        Print current.id, current.data, current.type
        current ← current.next

Method: print_chain_reverse()
    current ← tail
    While current is not null:
        Print current.id, current.data, current.type
        current ← current.prev

Method: sort_by_type(targetType)
    If head is null OR head.next is null:
        Print "Nothing to sort"
        Return
    Initialize matchListHead, matchListTail, otherListHead, otherListTail ← null
    current ← head
    While current is not null:
        nextNode ← current.next
        current.prev ← null
        current.next ← null
        If current.type equals targetType:
            If matchListHead is null:
                matchListHead, matchListTail ← current, current
            Else:
                matchListTail.next ← current
                current.prev ← matchListTail
                matchListTail ← current
        Else:
            If otherListHead is null:
                otherListHead, otherListTail ← current, current
            Else:
                otherListTail.next ← current
                current.prev ← otherListTail
                otherListTail ← current
            current ← nextNode
    If matchListTail is not null:
        matchListTail.next ← otherListHead
        If otherListHead is not null:
            otherListHead.prev ← matchListTail
        head ← matchListHead
        tail ← (otherListTail is not null ? otherListTail : matchListTail)
    Else:
        head ← otherListHead
        tail ← otherListTail
```

Evolution of Solution

Similar to the first question, when asking AI, I was again advised to use hash maps. Again, I dismissed this idea, and instead ruminated upon my original idea of doubly linked list. Though i didn't find any problems at first, while typing the python code for it, i realized that the sorting algorithm that I proposed wasn't in-place, and hence was wasting alot of space. So, with the help of a little AI to brainstorm a better in-place sorting algorithm, I settled on a modified version of my original idea, which is in place and works by unhooking and hooking the original nodes themselves into sub lists before concatenating the 2 sub lists, rather than being out of place and working by copying the nodes entirely

Test Cases

Note that the driver used takes in a .txt file, making these large test cases convenient to execute.

Test Case 1

```
A 1 0100 Success
A 2 1000 Fail
A 3 0010 Fail
A 4 1111 Success
A 5 1100 Fail
A 6 0001 Success
A 7 1010 Fail
A 8 0110 Success
A 9 0000 Fail
A 10 1110 Success
A 11 1001 Fail
A 12 0111 Success
A 13 1101 Fail
A 14 0011 Success
A 15 1011 Fail
A 16 0001 Success
A 17 1111 Fail
A 18 0101 Success
A 19 1001 Fail
A 20 0110 Success
P
R 2
R 5
R 7
R 11
R 17
R 21
P
PR
F 1
F 10
F 19
F 25
S Success
S Fail
P
```

Test Case 2

```
A 100 alpha Success
A 101 beta Success
A 102 gamma Fail
A 103 delta Success
A 104 epsilon Fail
A 105 zeta Success
A 106 eta Fail
A 107 theta Success
A 108 iota Fail
A 109 kappa Success
A 110 lambda Fail
A 111 mu Success
A 112 nu Fail
A 113 xi Success
A 114 omicron Fail
A 115 pi Success
A 116 rho Fail
A 117 sigma Success
A 118 tau Fail
A 119 upsilon Success
A 120 phi Fail
A 121 chi Success
A 122 psi Fail
A 123 omega Success
A 124 aleph Fail
A 125 beth Success
A 126 gimel Fail
A 127 daleth Success
A 128 he Fail
A 129 waw Success
F 100
F 105
F 115
F 125
F 200
F 999
R 102
R 110
R 118
R 124
R 777
R 999
P
PR
S Success
S Fail
P
```

Test Case 3

```
A 1000 Task_1000 Success
A 1001 Task_1001 Fail
A 1002 Task_1002 Success
A 1003 Task_1003 Fail
A 1004 Task_1004 Success
A 1005 Task_1005 Fail
A 1006 Task_1006 Success
A 1007 Task_1007 Fail
A 1008 Task_1008 Success
A 1009 Task_1009 Fail
A 1010 Task_1010 Success
A 1011 Task_1011 Fail
A 1012 Task_1012 Success
A 1013 Task_1013 Fail
A 1014 Task_1014 Success
A 1015 Task_1015 Fail
A 1016 Task_1016 Success
A 1017 Task_1017 Fail
A 1018 Task_1018 Success
A 1019 Task_1019 Fail
A 1020 Task_1020 Success
A 1021 Task_1021 Fail
A 1022 Task_1022 Success
A 1023 Task_1023 Fail
A 1024 Task_1024 Success
A 1025 Task_1025 Fail
A 1026 Task_1026 Success
A 1027 Task_1027 Fail
A 1028 Task_1028 Success
A 1029 Task_1029 Fail
A 1030 Task_1030 Success
A 1031 Task_1031 Fail
A 1032 Task_1032 Success
A 1033 Task_1033 Fail
A 1034 Task_1034 Success
A 1035 Task_1035 Fail
A 1036 Task_1036 Success
A 1037 Task_1037 Fail
A 1038 Task_1038 Success
A 1039 Task_1039 Fail
A 1040 Task_1040 Success
A 1041 Task_1041 Fail
A 1042 Task_1042 Success
A 1043 Task_1043 Fail
A 1044 Task_1044 Success
A 1045 Task_1045 Fail
A 1046 Task_1046 Success
A 1047 Task_1047 Fail
A 1048 Task_1048 Success
A 1049 Task_1049 Fail
A 1050 Task_1050 Success
A 1051 Task_1051 Fail
A 1052 Task_1052 Success
A 1053 Task_1053 Fail
A 1054 Task_1054 Success
A 1055 Task_1055 Fail
```

```
A 1056 Task_1056 Success
A 1057 Task_1057 Fail
A 1058 Task_1058 Success
A 1059 Task_1059 Fail
A 1060 Task_1060 Success
A 1061 Task_1061 Fail
A 1062 Task_1062 Success
A 1063 Task_1063 Fail
A 1064 Task_1064 Success
A 1065 Task_1065 Fail
A 1066 Task_1066 Success
A 1067 Task_1067 Fail
A 1068 Task_1068 Success
A 1069 Task_1069 Fail
A 1070 Task_1070 Success
A 1071 Task_1071 Fail
A 1072 Task_1072 Success
A 1073 Task_1073 Fail
A 1074 Task_1074 Success
A 1075 Task_1075 Fail
A 1076 Task_1076 Success
A 1077 Task_1077 Fail
A 1078 Task_1078 Success
A 1079 Task_1079 Fail
A 1080 Task_1080 Success
A 1081 Task_1081 Fail
A 1082 Task_1082 Success
A 1083 Task_1083 Fail
A 1084 Task_1084 Success
A 1085 Task_1085 Fail
A 1086 Task_1086 Success
A 1087 Task_1087 Fail
A 1088 Task_1088 Success
A 1089 Task_1089 Fail
A 1090 Task_1090 Success
A 1091 Task_1091 Fail
A 1092 Task_1092 Success
A 1093 Task_1093 Fail
A 1094 Task_1094 Success
A 1095 Task_1095 Fail
A 1096 Task_1096 Success
A 1097 Task_1097 Fail
A 1098 Task_1098 Success
A 1099 Task_1099 Fail
F 1000
F 1050
F 1099
F 9999
R 1003
R 1011
R 1025
R 1049
R 1065
R 1087
R 1099
R 8888
P
PR
```

```
S Success
P
S Fail
P
P
```

Final Code

```python
class Blockchain:
    class Block:
        # Represents a single block in the blockchain. Each block contains an ID, data, and a type.
        def __init__(self, block_id, data, block_type):
            self.id = block_id
            self.data = data
            self.type = block_type
            self.next = None
            self.prev = None

    # A doubly linked list implementation of a simple blockchain. Supports efficient insertion at the end and reverse traversal.
    def __init__(self):
        self.head = None
        self.tail = None

    def is_empty(self):
        # Returns True if the blockchain is empty. Time complexity: O(1)
        return self.head is None

    def add(self, block_id, data, block_type):
        # Appends a new block to the end of the chain. Time complexity: O(1)
        new_block = self.Block(block_id, data, block_type)

        if self.head is None:
            self.head = self.tail = new_block
        else:
            self.tail.next = new_block
            new_block.prev = self.tail
            self.tail = new_block

    def delete(self, block_id):
        # Deletes the block with the given ID, if found. Time complexity: O(n)
        current = self.head

        while current:
            if current.id == block_id:
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next  # Removing head

                if current.next:
                    current.next.prev = current.prev
                else:
                    self.tail = current.prev  # Removing tail
                print(f"Deleted␣id:␣{current.id},␣data:␣{current.data},␣type:␣{current.type}")
                del current
                return True
            current = current.next
        print(f"Could␣not␣delete␣id:{block_id}␣␣ID␣not␣found")
        return False  # Block not found

    def find(self, block_id):
        # Searches for and prints the block with the specified ID. Time complexity: O(n)
        current = self.head
        while current:
            if current.id == block_id:
                print(f"ID:␣{current.id},␣Data:␣{current.data},␣Type:␣{current.type}")
                return current
            current = current.next

        print(f"ID␣{block_id}␣not␣found.")
        return None

    def print_chain(self):
        # Prints all blocks in forward order. Time complexity: O(n)
        print("------Printing␣in␣forward␣direction-------")

        current = self.head
        while current:
            print(f"ID:␣{current.id},␣Data:␣{current.data},␣Type:␣{current.type}")
            current = current.next

    def print_chain_reverse(self):
        # Prints all blocks in reverse order. Time complexity: O(n)
        print("------Printing␣in␣reverse␣direction-------")

        current = self.tail
        while current:
            print(f"ID:␣{current.id},␣Data:␣{current.data},␣Type:␣{current.type}")
            current = current.prev

    def sort_by_type(self, block_type):
        # In-place partition: group matching type first, preserve order, O(n) time
        if not self.head or not self.head.next:
            print(f"List␣sorted␣with␣{block_type}␣first.")
            return

        # Pointers for two lists
        match_head = match_tail = None
        other_head = other_tail = None
        current = self.head

        while current:
            nxt = current.next
            # Detach
            current.prev = current.next = None

            if current.type == block_type:
                if not match_head:
                    match_head = match_tail = current
                else:
                    match_tail.next = current
                    current.prev = match_tail
                    match_tail = current
            else:
                if not other_head:
                    other_head = other_tail = current
                else:
                    other_tail.next = current
                    current.prev = other_tail
                    other_tail = current

            current = nxt

        # Combine lists
        if match_tail:
            match_tail.next = other_head
            if other_head:
                other_head.prev = match_tail
            self.head = match_head
            self.tail = other_tail or match_tail
        else:
            self.head = other_head
            self.tail = other_tail

        print(f"List␣sorted␣with␣{block_type}␣first.")


def main():
    chain = Blockchain()
    filePath = "testCase3.txt"
    try:
        f = open(filePath, "r", encoding="utf-8")
    except:
        print(f"File␣not␣found:␣{filePath}")
        return

    with f:
        for line in f:
            parts = line.strip().split()
            if not parts:
                continue  # skip empty lines

            cmd = parts[0]

            if cmd == "A" and len(parts) == 4:
                # A id data type
                block_id = parts[1]
                data = parts[2]
                block_type = parts[3]
                chain.add(block_id, data, block_type)

            elif cmd == "R" and len(parts) == 2:
                # R id
                removed = chain.delete(parts[1])
                if not removed:
                    print(f"No␣block␣with␣ID␣'{parts[1]}'␣to␣remove.")

            elif cmd == "F" and len(parts) == 2:
                # F id
                chain.find(parts[1])

            elif cmd == "P":
                chain.print_chain()

            elif cmd == "PR":
                chain.print_chain_reverse()

            elif cmd == "S" and len(parts) == 2:
                chain.sort_by_type(parts[1])

            else:
                print(f"Invalid␣command:␣{line.strip()}")

main()
```