1.

(a)

Yes, we can use greedy algorithm to solve this problem. In this case each time we store the smallest remaining file until no more files can be stored into the memory. Assuming that k files have been stored, then the minimum total size of these k files is and we can get , , so it’s impossible to store k+1 files into the memory.

(b)

No, we cannot use greedy algorithm to solve this problem. Each time we store the smallest remaining file into the memory but it cannot make sure the maximum usage of memory. For example, if the memory size is 5GB and three files which are 1GB, 2GB and 4GB, respectively. By greedy algorithm we will store 1GB file and 2GB file, but the optimal solution is storing 1GB file and 4GB file to use as much of the memory capacity as possible.

2.

(a)

**function** CheckSubSet(X,Y)

Quicksort(Y[]) // Reference: lecture 11 slide 17

**for** i1 to m **do**

**if** BinSearch(Y[], ) == -1 **do** // Reference: lecture 10 slide 10

**return** False

**return** Ture

**end function**

Firstly, I sort Y (I saw on the discussion board that I can directly use Y[] as an array) and then binary search every integer of X in Y. If is not found in Y then returns False. It will return Ture if and only if every integer of X can be found in Y which means X is a subset of Y.

(b)

**function** CheckSubSet(X,Y)

Hash.initialize // Initialize a hashtable

// Reference: lecture 17 slide 3

**for** i1 to n **do**

Hash.insert(hashtable, ) // Reference: lecture 17 slide 3

**for** j1 to m **do**

**if** Hash.find(hashtable, ) == False **do** // Find in the hashtable

// Reference: lecture 17 slide 3

**return** False

**return** True

**end function**

Firstly, initialize a hashtable and insert all of the integers in Y into the hashtable. To avoid complex insertion and searching operation, we can choose appropriate, efficient hash function (DJB Hash, BKDR Hash) and better collision handling strategies (Double Hashing) to make sure an O(n) complexity.

3.

**function** FindMaxProduct(T)

// Input: T a binary tree that stores an integer key value in each node

// Output: MaxProduct the maxmimun product of the key values on all possible paths in tree T

initialize a queue /\* This queue will be manipulated by function UpdateQueue(),

so it’s like a “global variable” \*/

**if** T is not an empty tree **do**

UpdateQueue(T)

MaxProduct

**if** the queue is not empty **do**

remove the front vertex u from the queue

**if** u > MaxProduct **do**

MaxProductu

**return** MaxProduct

**else** **return** *null*

**end function**

**function** UpdateQueue(T)

**if** T.left is not empty **do**

Larray of zeros of length 2 // Initialize an array of length 2, L[0]=L[1]=0

LUpdateQueue(T.left) // UpdateQueue() returns A, so L[0]=A[0], L[1]=A[1]

**if** T.right is not empty **do**

Rarray of zeros of length 2 // Initialize an array of length 2, R[0]=R[1]=0

RUpdateQueue(T.right) // UpdateQueue() returns A, so R[0]=A[0], R[1]=A[1]

**if** T.left and T.right are empty **do**

Aarray of zeros of length 2 // Initialize an array of length 2 which will be returned

A[0], A[1]T.key // In this case, the maximum and minimum values are equal

add A[0] to the queue

**return** A

**else** **if** T.right is empty **do**

Aarray of zeros of length 2

A[0]max(L[0]T.key, L[1]T.key, T.key) // A[0] stores the maximum value

A[1]min(L[0]T.key, L[1]T.key, T.key) // A[1] stores the minimum value

add A[0] to the queue

**return** A

**else if** T.left is empty **do**

Aarray of zeros of length 2

A[0]max(R[0]T.key, R[1]T.key, T.key)

A[1]min(R[0]T.key, R[1]T.key, T.key)

add A[0] to the queue

**return** A

**else**

Maxmax(L[0]T.keyR[0], L[0]T.keyR[1],

L[1]T.keyR[0], L[1]T.keyR[1])

add Max to the queue

A[0]max(L[0]T.key, L[1]T.key, R[0]T.key, R[1]T.key, T.key)

A[1]min(L[0]T.key, L[1]T.key, R[0]T.key, R[1]T.key, T.key)

add A[0] to the queue

**return** A

**end function**

I use divide-and-conquer technique to solve this problem. According to the Master Theorem, the time complexity of my algorithm is O(n) where a=2, b=2, d=0. For each recursion, the temporary max(A[0]), min values(A[1]) will be returned for further processing and potential maximum products(A[0], Max) will be injected into the queue.

(b)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | node | L[0] | L[1] | R[0] | R[1] | A[0] | A[1] | Max | queue |
| 1st | 5 |  |  |  |  | 5 | 5 |  | 5 |
| 2nd | 1 |  |  |  |  | 1 | 1 |  | 1,5 |
| 3rd | 0 | 1 | 1 |  |  | 0 | 0 |  | 0,1,5 |
| 4th | -2 | 5 | 5 | 0 | 0 | 0 | -10 | 0 | 0,0,0,1,5 |
| 5th | -2 |  |  |  |  | -2 | -2 |  | -2,0,0,0,1,5 |
| 6th | 2 |  |  |  |  | 2 | 2 |  | 2,-2,0,0,0,1,5 |
| 7th | 4 | -2 | -2 | 2 | 2 | 8 | -8 | -16 | 8,-16,2,-2,0,0,0,1,5 |
| 8th | 3 | 0 | -10 | 8 | -8 | 24 | -30 | 240 | 24,240,-16,8,2,-2,0,0,0,1,5 |

Finally, we select the maximum value in the queue which is 240.

4.

(a)

****

(b)

**function** BuildTreap(R)

// Input: R a set of records in which each has a key and a priority

// Output: T a binary search tree with a modified way of ordering the nodes

T**new** Tree

T.key, T.priority.priority

**for** i to n **do**

Insert(T, )

**return** T

**end function**

**function** Insert(T,R)

**if** R.key < T.key **do**

**if** T.left.key != *null* **do**

Insert(T.left, R)

**else** T.left.keyR.key, T.left.priorityR.priority

**if** T.priority > T.left.priority **do**

RightRotation(T)

**else**

**if** T.right.key != *null* **do**

Insert(T.right, R)

**else** T.right.keyR.key, T.right.priorityR.priority

**if** T.priority > T.right.priority **do**

LeftRotation(T)

**return** T

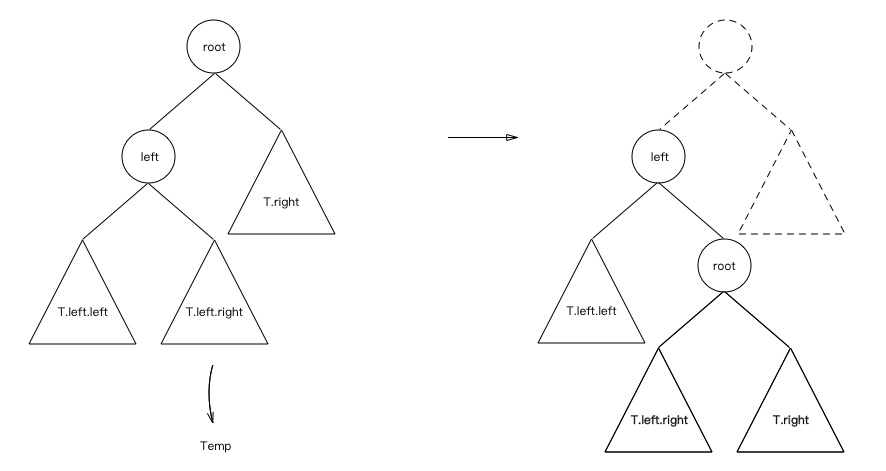
**end function**

**function** RightRotation(T)

Temp **new** Tree

TempT.left.right

T.left.right.keyT.key, T.left.right.priorityT.priority

 T.left.right.rightT.right

T.left.right.leftTemp

**return** T.left

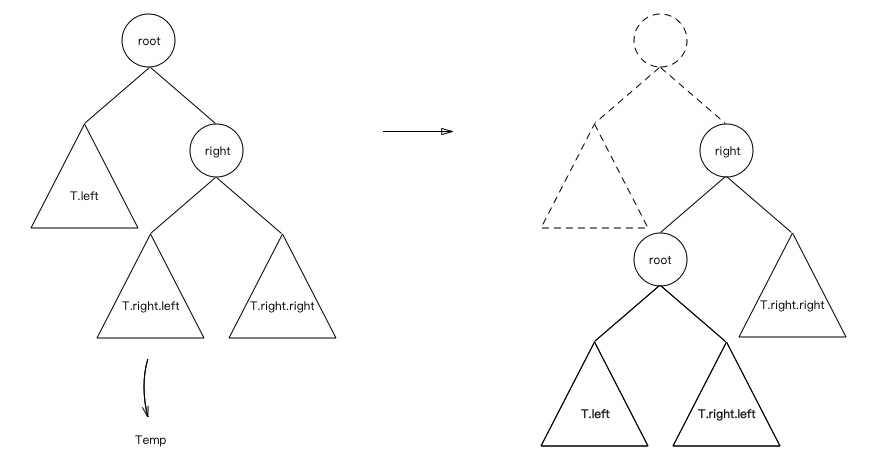
**end function**

**function** LeftRotation(T)

Temp **new** Tree

TempT.right.left

T.right.left.keyT.key, T.right.left.priorityT.priority

 T.right.left.leftT.left

T.right.left.rightTemp

**return** T.right

**end function**