Design an efficient data structure for given operations

Design a Data Structure for the following operations. The data structure should be efficient enough to accommodate the operations according to their frequency.

```
1) findMin() : Returns the minimum item.
   Frequency: Most frequent

2) findMax() : Returns the maximum item.
    Frequency: Most frequent

3) deleteMin() : Delete the minimum item.
    Frequency: Moderate frequent

4) deleteMax() : Delete the maximum item.
    Frequency: Moderate frequent

5) Insert() : Inserts an item.
    Frequency: Least frequent

6) Delete() : Deletes an item.
    Frequency: Least frequent.
```

A **simple solution** is to maintain a sorted array where smallest element is at first position and largest element is at last. The time complexity of findMin(), findMAx() and deleteMax() is O(1). But time complexities of deleteMin(), insert() and delete() will be O(n).

Can we do the most frequent two operations in O(1) and other operations in O(Logn) time?.
The idea is to use two binary heaps (one max and one min heap). The main challenge is, while deleting an item, we need to delete from both min-heap and max-heap. So, we need some kind of mutual data structure. In the following design, we have used doubly linked list as a mutual data structure. The doubly linked list contains all input items and indexes of corresponding min and max heap nodes. The nodes of min and max heaps store addresses of nodes of doubly linked list. The root node of min heap stores the address of minimum item in doubly linked list. Similarly, root of max heap stores address of maximum item in doubly linked list. Following are the details of operations.

1) findMax(): We get the address of maximum value node from root of Max Heap. So this is a O(1) operation.

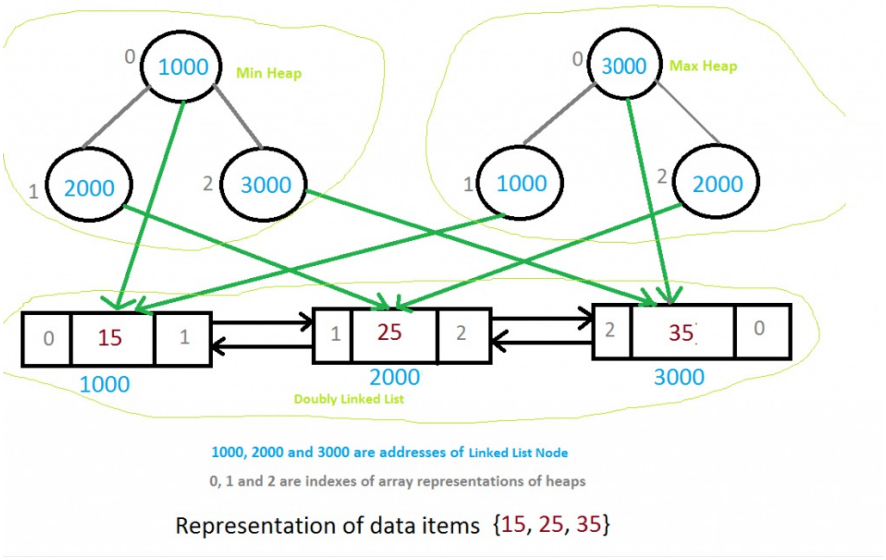
1) findMin(): We get the address of minimum value node from root of Min Heap. So this is a O(1) operation.

3) deleteMin(): We get the address of minimum value node from root of Min Heap. We use this address to find the node in doubly linked list. From the doubly linked list, we get node of Max Heap. We delete node from all three. We can delete a node from doubly linked list in O(1) time. delete() operations for max and min heaps take O(Logn) time.

4) deleteMax(): is similar to deleteMin()

5) Insert() We always insert at the beginning of linked list in O(1) time. Inserting the address in Max and Min Heaps take O(Logn) time. So overall complexity is O(Logn)

6) Delete() We first search the item in Linked List. Once the item is found in O(n) time, we delete it from linked list. Then using the indexes stored in linked list, we delete it from Min Heap and Max Heaps in O(Logn) time. *So overall complexity of this operation is O(n). The Delete operation can be optimized to O(Logn) by using a balanced binary search tree instead of doubly linked list as a mutual data structure. Use of balanced binary search will not effect time complexity of other operations as it will act as a mutual data structure like doubly Linked List.*



Representation of data items {15, 25, 35}

Following is C implementation of the above data structure.

```
// C program for efficient data structure
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A node of doubly linked list
struct LNode
{
    int data;
    int minHeapIndex;
    int maxHeapIndex;
    struct LNode *next, *prev;
};

// Structure for a doubly linked list
struct List
{
    struct LNode *head;
};

// Structure for min heap
struct MinHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// Structure for max heap
struct MaxHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// The required data structure
struct MyDS
{
    struct MinHeap* minHeap;
    struct MaxHeap* maxHeap;
    struct List* list;
};

// Function to swap two integers
void swapData(int* a, int* b)
{ int t = *a;   *a = *b;   *b = t; }

// Function to swap two List nodes
void swapLNode(struct LNode** a, struct LNode** b)
```

```
{ struct LNode* t = *a; *a = *b; *b = t; }

// A utility function to create a new List node
struct LNode* newLNode(int data)
{
    struct LNode* node =
        (struct LNode*) malloc(sizeof(struct LNode));
    node->minHeapIndex = node->maxHeapIndex = -1;
    node->data = data;
    node->prev = node->next = NULL;
    return node;
}

// Utility function to create a max heap of given capacity
struct MaxHeap* createMaxHeap(int capacity)
{
    struct MaxHeap* maxHeap =
     (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = 0;
    maxHeap->capacity = capacity;
    maxHeap->array =
     (struct LNode**) malloc(maxHeap->capacity * sizeof(struct LNode*));
    return maxHeap;
}

// Utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
       (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
       (struct LNode**) malloc(minHeap->capacity * sizeof(struct LNode*));
    return minHeap;
}

// Utility function to create a List
struct List* createList()
{
    struct List* list =
      (struct List*) malloc(sizeof(struct List));
    list->head = NULL;
    return list;
}

// Utility function to create the main data structure
// with given capacity
struct MyDS* createMyDS(int capacity)
{
    struct MyDS* myDS =
        (struct MyDS*) malloc(sizeof(struct MyDS));
    myDS->minHeap = createMinHeap(capacity);
    myDS->maxHeap = createMaxHeap(capacity);
    myDS->list = createList();
    return myDS;
}

// Some basic operations for heaps and List
int isMaxHeapEmpty(struct MaxHeap* heap)
{  return (heap->size == 0); }

int isMinHeapEmpty(struct MinHeap* heap)
{  return heap->size == 0; }

int isMaxHeapFull(struct MaxHeap* heap)
{  return heap->size == heap->capacity; }

int isMinHeapFull(struct MinHeap* heap)
{  return heap->size == heap->capacity; }

int isListEmpty(struct List* list)
{  return !list->head;  }
```

```
{ return !list->head; }

int hasOnlyOneLNode(struct List* list)
{    return !list->head->next && !list->head->prev; }


// The standard minheapify function.  The only thing it does extra
// is swapping indexes of heaps inside the List
void minHeapify(struct MinHeap* minHeap, int index)
{
    int smallest, left, right;
    smallest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( minHeap->array[left] &&
         left < minHeap->size &&
         minHeap->array[left]->data < minHeap->array[smallest]->data
       )
        smallest = left;

    if ( minHeap->array[right] &&
         right < minHeap->size &&
         minHeap->array[right]->data < minHeap->array[smallest]->data
       )
        smallest = right;

    if (smallest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&(minHeap->array[smallest]->minHeapIndex),
                 &(minHeap->array[index]->minHeapIndex));

        // Now swap pointers to List nodes
        swapLNode(&minHeap->array[smallest],
                  &minHeap->array[index]);

        // Fix the heap downward
        minHeapify(minHeap, smallest);
    }
}

// The standard maxHeapify function.  The only thing it does extra
// is swapping indexes of heaps inside the List
void maxHeapify(struct MaxHeap* maxHeap, int index)
{
    int largest, left, right;
    largest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( maxHeap->array[left] &&
         left < maxHeap->size &&
         maxHeap->array[left]->data > maxHeap->array[largest]->data
       )
        largest = left;

    if ( maxHeap->array[right] &&
         right < maxHeap->size &&
         maxHeap->array[right]->data > maxHeap->array[largest]->data
       )
        largest = right;

    if (largest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&maxHeap->array[largest]->maxHeapIndex,
                 &maxHeap->array[index]->maxHeapIndex);

        // Now swap pointers to List nodes
        swapLNode(&maxHeap->array[largest],
```

```
                  &maxHeap->array[index]);

        // Fix the heap downward
        maxHeapify(maxHeap, largest);
    }
}

// Standard function to insert an item in Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct LNode* temp)
{
    if (isMinHeapFull(minHeap))
        return;

    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && temp->data < minHeap->array[(i - 1) / 2]->data )
    {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        minHeap->array[i]->minHeapIndex = i;
        i = (i - 1) / 2;
    }

    minHeap->array[i] = temp;
    minHeap->array[i]->minHeapIndex = i;
}

// Standard function to insert an item in Max Heap
void insertMaxHeap(struct MaxHeap* maxHeap, struct LNode* temp)
{
    if (isMaxHeapFull(maxHeap))
        return;

    ++maxHeap->size;
    int i = maxHeap->size - 1;
    while (i && temp->data > maxHeap->array[(i - 1) / 2]->data )
    {
        maxHeap->array[i] = maxHeap->array[(i - 1) / 2];
        maxHeap->array[i]->maxHeapIndex = i;
        i = (i - 1) / 2;
    }

    maxHeap->array[i] = temp;
    maxHeap->array[i]->maxHeapIndex = i;
}


// Function to find minimum value stored in the main data structure
int findMin(struct MyDS* myDS)
{
    if (isMinHeapEmpty(myDS->minHeap))
        return INT_MAX;

    return myDS->minHeap->array[0]->data;
}

// Function to find maximum value stored in the main data structure
int findMax(struct MyDS* myDS)
{
    if (isMaxHeapEmpty(myDS->maxHeap))
        return INT_MIN;

    return myDS->maxHeap->array[0]->data;
}

// A utility function to remove an item from linked list
void removeLNode(struct List* list, struct LNode** temp)
{
    if (hasOnlyOneLNode(list))
        list->head = NULL;

    else if (!(*temp)->prev) // first node
    {
        list->head = (*temp)->next;
```

```
        (*temp)->next->prev = NULL;
    }
    // any other node including last
    else
    {
        (*temp)->prev->next = (*temp)->next;
        // last node
        if ((*temp)->next)
            (*temp)->next->prev = (*temp)->prev;
    }
    free(*temp);
    *temp = NULL;
}

// Function to delete maximum value stored in the main data structure
void deleteMax(struct MyDS* myDS)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMaxHeapEmpty(maxHeap))
        return;
    struct LNode* temp = maxHeap->array[0];

    // delete the maximum item from maxHeap
    maxHeap->array[0] =
        maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[0]->maxHeapIndex = 0;
    maxHeapify(maxHeap, 0);

    // remove the item from minHeap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to delete minimum value stored in the main data structure
void deleteMin(struct MyDS* myDS)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMinHeapEmpty(minHeap))
        return;
    struct LNode* temp = minHeap->array[0];

    // delete the minimum item from minHeap
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[0]->minHeapIndex = 0;
    minHeapify(minHeap, 0);

    // remove the item from maxHeap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to enList an item to List
void insertAtHead(struct List* list, struct LNode* temp)
{
    if (isListEmpty(list))
        list->head = temp;
```

```
    else
    {
        temp->next = list->head;
        list->head->prev = temp;
        list->head = temp;
    }
}

// Function to delete an item from List. The function also
// removes item from min and max heaps
void Delete(struct MyDS* myDS, int item)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isListEmpty(myDS->list))
        return;

    // search the node in List
    struct LNode* temp = myDS->list->head;
    while (temp && temp->data != item)
        temp = temp->next;

    // if item not found
    if (!temp || temp && temp->data != item)
        return;

    // remove item from min heap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove item from max heap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove node from List
    removeLNode(myDS->list, &temp);
}

// insert operation for main data structure
void Insert(struct MyDS* myDS, int data)
{
    struct LNode* temp = newLNode(data);

    // insert the item in List
    insertAtHead(myDS->list, temp);

    // insert the item in min heap
    insertMinHeap(myDS->minHeap, temp);

    // insert the item in max heap
    insertMaxHeap(myDS->maxHeap, temp);
}

// Driver program to test above functions
int main()
{
    struct MyDS *myDS = createMyDS(10);
    // Test Case #1
    /*Insert(myDS, 10);
    Insert(myDS, 2);
    Insert(myDS, 32);
    Insert(myDS, 40);
    Insert(myDS, 5);*/

    // Test Case #2
    Insert(myDS, 10);
    Insert(myDS, 20);
```

```
    Insert(myDS, 30);
    Insert(myDS, 40);
    Insert(myDS, 50);

    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMax(myDS);  // 50 is deleted
    printf("After deleteMax()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMin(myDS); // 10 is deleted
    printf("After deleteMin()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    Delete(myDS, 40); // 40 is deleted
    printf("After Delete()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n", findMin(myDS));

    return 0;
}
```

Output:

```
Maximum = 50
Minimum = 10

After deleteMax()
Maximum = 40
Minimum = 10

After deleteMin()
Maximum = 40
Minimum = 20

After Delete()
Maximum = 30
Minimum = 20
```