Priority Queues and Heaps

Declaration of Heap :

```
struct Heap {
    int *array;
    int count;
    int capacity;
    int heap_type; // max heap (1) or min heap (0)
};
```

Creating Heap :

```
Heap* createHeap(int capacity, int heap_type) {
    Heap* h = new Heap();
    h->heap_type = heap_type;
    h->capacity = capacity;
    h->count = 0;
    h->array = new int[capacity];
    return h;
}
```

Parent of a Node :

```
int parent(Heap* h, int i) {
    if (i <= 0 || i >= h->count) return -1;
    return (i - 1) / 2;
}
```

Children of a Node :

```
int leftChild(Heap* h, int i) {
    int left = 2 * i + 1;
    return (left >= h->count) ? -1 : left;
}

int rightChild(Heap* h, int i) {
    int right = 2 * i + 2;
    return (right >= h->count) ? -1 : right;
}
```

Getting the maximum element :

```
int getMax(Heap* h) {
    if (h->count == 0) return -1;
    return h->array[0];
}
```

Heapifying an element :

```
void heapify(Heap* h, int i) {
    int largest = i;
    int left = leftChild(h, i);
    int right = rightChild(h, i);

    if (left != -1 && h->array[left] > h->array[largest])
        largest = left;

    if (right != -1 && h->array[right] > h->array[largest])
        largest = right;

    if (largest != i) {
        std::swap(h->array[i], h->array[largest]);
        heapify(h, largest);
    }
}
```

Deleting an element :

```
int deleteMax(Heap* h) {
    if (h->count == 0) return -1;
    int data = h->array[0];
    h->array[0] = h->array[h->count - 1];
    h->count--;
    heapify(h, 0);
    return data;
}
```

Inserting an element :

```
void insert(Heap* h, int data) {
    if (h->count == h->capacity) return;
    h->count++;
    int i = h->count - 1;
    while (i > 0 && data > h->array[parent(h, i)]) {
        h->array[i] = h->array[parent(h, i)];
        i = parent(h, i);
    }
    h->array[i] = data;
}
```

Destroying Heap :

```
void destroyHeap(Heap* h) {
    delete[] h->array;
    delete h;
}
```

Heapifying the array :

```
Heap* heapifyArray(int* arr, int n, int heap_type) {
    Heap* h = createHeap(n, heap_type);
    h->array = arr;
    h->count = n;
    for (int i = (n - 2) / 2; i >= 0; i--)
        heapify(h, i);
    return h;
}
```

Heapsort :

```
void heapSort(int* arr, int n) {
    Heap* h = heapifyArray(arr, n, 1);
    for (int i = n - 1; i > 0; i--) {
        std::swap(h->array[0], h->array[i]);
        h->count--;
        heapify(h, 0);
    }
    delete h;
}
```

Problems

Maximum element in a min-heap :

```
int findMaxInMinHeap(Heap* h) {
    if (h->count == 0) return -1;
    int maxElement = h->array[h->count / 2];
    for (int i = h->count / 2 + 1; i < h->count; i++) {
        if (h->array[i] > maxElement) {
            maxElement = h->array[i];
        }
    }
    return maxElement;
}
```

Deleting the ith element in a min-heap :

```
// involves rearranging the heap to maintain the heap property after
// the removal
void deleteAtIndex(Heap* h, int i) {
    if (i < 0 || i >= h->count) return; // Index out of bounds

    // Replace the element at index i with the last element
    h->array[i] = h->array[h->count - 1];
    h->count--; // Reduce heap size
```

```
    // Perform heapify to restore heap property
    heapify(h, i);
}

void heapify(Heap* h, int i) {
    ...
}
```

Finding the kth smallest element in a min-heap :

```
int extractMin(Heap* h) {
    if (h->count == 0) return -1; // Empty heap

    int minElement = h->array[0];
    h->array[0] = h->array[h->count - 1];
    h->count--; // Reduce heap size
    heapify(h, 0); // Restore heap property
    return minElement;
}

int kthSmallest(Heap* h, int k) {
    if (k <= 0 || k > h->count) return -1; // Invalid k

    // Create a copy of the heap
    Heap* tempHeap = new Heap;
    tempHeap->array = new int[h->capacity];
    tempHeap->count = h->count;
    tempHeap->capacity = h->capacity;
    tempHeap->heap_type = h->heap_type;

    for (int i = 0; i < h->count; i++) {
        tempHeap->array[i] = h->array[i];
    }

    // Extract the minimum k times
    int result = -1;
    for (int i = 0; i < k; i++) {
        result = extractMin(tempHeap);
    }

    // Clean up
    delete[] tempHeap->array;
    delete tempHeap;

    return result;
}
```

```
void heapify(Heap* h, int i) {
    ...
}
```

Implementation of a stack using a heap :

```
class StackUsingHeap {
    priority_queue<pair<int, int>> maxHeap;
    int timestamp;

public:
    StackUsingHeap() : timestamp(0) {}

    void push(int x) {
        maxHeap.push({timestamp++, x});
    }

    int pop() {
        if (maxHeap.empty()) return -1;
        int topElement = maxHeap.top().second;
        maxHeap.pop();
        return topElement;
    }

    int top() {
        if (maxHeap.empty()) return -1;
        return maxHeap.top().second;
    }

    bool isEmpty() {
        return maxHeap.empty();
    }
};
```

Implementation of a queue using a heap :

```
class QueueUsingHeap {
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> minHeap;
    int timestamp;

public:
    QueueUsingHeap() : timestamp(0) {}

    void enqueue(int x) {
```

```
        minHeap.push({timestamp++, x});
    }

    int dequeue() {
        if (minHeap.empty()) return -1;
        int frontElement = minHeap.top().second;
        minHeap.pop();
        return frontElement;
    }

    int front() {
        if (minHeap.empty()) return -1;
        return minHeap.top().second;
    }

    bool isEmpty() {
        return minHeap.empty();
    }
};
```

Maximum sum in sliding window :

```
#include <queue>
#include <vector>
using namespace std;

int maxSumInSlidingWindow(vector<int>& nums, int k) {
    priority_queue<pair<int, int>> maxHeap;
    int n = nums.size();
    int maxSum = 0, windowSum = 0;

    for (int i = 0; i < k; i++) {
        windowSum += nums[i];
        maxHeap.push({nums[i], i});
    }
    maxSum = windowSum;

    for (int i = k; i < n; i++) {
        windowSum += nums[i] - nums[i - k];
        maxHeap.push({nums[i], i});

        while (maxHeap.top().second <= i - k) {
            maxHeap.pop();
        }

        maxSum = max(maxSum, windowSum);
```

```
    }

    return maxSum;
}
```