```
// heap.h
// a binary min heap

#ifndef HEAP_H
#define HEAP_H

#include <iostream>

const int DEFAULT_SIZE = 100;

template <class KeyType> class MinPriorityQueue;

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);          // default constructor
    MinHeap(KeyType initA[], int n);        // construct heap from array
    MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
    ~MinHeap();                             // destructor

    void heapSort(KeyType sorted[]);  // heapsort, return result in sorted

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);  // assignment operator
    std::string toString() const;     // return string representation

  private:
    KeyType *A;     // array containing the heap
    int heapSize;   // size of the heap
    int capacity;   // size of A

          void heapify(int index);          // heapify subheap rooted at index
    void buildHeap();                 // build heap
        int leftChild(int index) { return 2 * index + 1; }  // return index of left child
        int rightChild(int index) { return 2 * index + 2; } // return index of right chil
d
        int parent(int index) { return (index - 1) / 2; }   // return index of parent
    void heapifyR(int index);                 // recursive heapify
    void heapifyI(int index);                 // iterative heapify
    void swap(int index1, int index2);        // swap elements in A
    void copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
    void destroy();                           // deallocate heap

    friend class MinPriorityQueue<KeyType>;
    friend void test_same(MinHeap<int>& heapA, MinHeap<int>& heapB);
    friend void test_isheap(MinHeap<int>& heap);
    friend void test_construct(MinHeap<int>& heap);
    friend void test_heapsort(int a[], int n);
    friend void test_isempty(MinHeap<int>& heap);

};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap);

//#include "heap.cpp"

#endif
```

```
// heap.cpp

#include <sstream>
#include <iostream>
#include "heap.h"

// Implement heap methods here.

// Use the following toString() for testing purposes.

//The default constructor of MinHeap making capacity
//default (100) and heapSise 0.
template <class KeyType>
MinHeap<KeyType>::MinHeap(int n){
        A = new KeyType[n];
        heapSize = 0;
        capacity = n;
}

//Calls heapifyR
//pre-condition: the parent of the node are in correct order
//post-condition: the node and its children are in correct order
template <class KeyType>
void MinHeap<KeyType>::heapify(int index){
        heapifyR(index);
}

//builds a correct min heap
template <class KeyType>
void MinHeap<KeyType>::buildHeap(){
        heapSize = capacity;
        for(int i = (capacity/2)-1; i >= 0; i--){
                heapify(i);
        }
}

//A recursive version of heapifyR
template <class KeyType>
void MinHeap<KeyType>::heapifyR(int index){
        int left = leftChild(index);
        int right = rightChild(index);
        int smallest = index;

        //checks if left side is smaller than index
        if ((left < heapSize) && (A[left] < A[index])){
                smallest = left;
        } else {
                smallest = index ;
        }

        //check if right side is smaller than index
        if ((right < heapSize) && (A[right] < A[smallest])){
                smallest = right;
        }

        //if the smallest is not at the index then swap and repeat
        if (smallest != index){
                swap(index, smallest);
                heapifyR(smallest);
        }
}

//An iterative version of heapify
template <class KeyType>
void MinHeap<KeyType>::heapifyI(int index){
        int left = leftChild(index);
        int right = rightChild(index);
        int smallest = index;
        int i = index;
```

```
        //runs a while loop until the index that is being looked
        //at is larger than the heapSize (when we reach past a leaf)
        while(i < heapSize){
                if ((left < heapSize) && (A[left] < A[i])){
                        smallest = left;
                } else {
                        smallest = i;
                }
                if ((right < heapSize) && (A[right] < A[smallest])){
                        smallest = right;
                }

                //if smallest is at the current index then swap and update
                //everything before rerunning, if not break loop.
                if (smallest != i){
                        swap(i, smallest);
                        i = smallest;
                        left = leftChild(i);
                        right = rightChild(i);
                }else{
                        break;
                }
        }
}

//Swaps two items in a heap using the index
template <class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2){
        KeyType temp = A[index1];
        A[index1] = A[index2];
        A[index2] = temp;
}

//Copies a heap into a new heap object
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap){
        heapSize = heap.heapSize;
        capacity = heap.capacity;
        A = new KeyType[capacity];
        for(int i = 0; i < heapSize; i++){
                A[i] = heap.A[i];
        }
}

//destroys heap
template <class KeyType>
void MinHeap<KeyType>::destroy(){
        heapSize = 0;
        capacity = 0;
        delete[] A;
}

//A second constructor for MinHeap, creates heap using a list and list size
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType initA[], int n){
        A = new KeyType[n];
        for(int i = 0; i < n; i++){
                A[i] = initA[i];
        }
        capacity = n;
        heapSize = n;
        buildHeap();
}

//Another constructor that creates a heap from another heap
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap){
        copy(heap);
}
```

```
//deconstructor
template <class KeyType>
MinHeap<KeyType>::~MinHeap(){
        destroy();
}

//makes a list with sorted values (ascending) from a heap, destorying
//the heap afterwards
template <class KeyType>
void MinHeap<KeyType>::heapSort(KeyType sorted[]){
        buildHeap();
        int x = 0;
        for(int i = heapSize - 1; i >= 0; i--){
                sorted[x] = A[0];
                swap(0, i);
                heapSize = heapSize - 1;
                heapify(0);
                x++;
        }
}

//Makes the equal operator copy one heap to another.
template <class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap){
        if (this != &heap){
                destroy();
                copy(heap);
        }
        return *this;
}

template <class KeyType>
std::string MinHeap<KeyType>::toString() const
{
        std::stringstream ss;

        if (capacity == 0)
                ss << "[ ]";
        else
        {
                ss << "[";
                if (heapSize > 0)
                {
                        for (int index = 0; index < heapSize - 1; index++)
                                ss << A[index] << ", ";
                        ss << A[heapSize - 1];
                }
                ss << " | ";
                if (capacity > heapSize)
                {
                        for (int index = heapSize; index < capacity - 1; index++)
                                ss << A[index] << ", ";
                        ss << A[capacity - 1];
                }
                ss << "]";
        }
        return ss.str();
}

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap)
{
        return stream << heap.toString();
}
```

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <cassert>
#include <sys/time.h>
#include "heap.cpp"
using namespace std;

void test_same(MinHeap<int>& heapA, MinHeap<int>& heapB){
  for(int i = 0; i <= heapA.heapSize - 1; i++){
    assert(heapA.A[i] == heapB.A[i]);
  }
}

void test_isheap(MinHeap<int>& heap){
  int left;
  int right;
  for(int i = 0; i <= heap.heapSize - 1; i++){
    left = heap.leftChild(i);
    right = heap.rightChild(i);
    if(left < heap.heapSize && right < heap.heapSize){
      assert(heap.A[i] <= heap.A[left]);
      assert(heap.A[i] <= heap.A[right]);
    }
  }
}

void test_construct(MinHeap<int>& heap){
  assert(heap.capacity != 0);
}

void test_heapsort(int a[], int n){
  for(int i = 1; i <= n-1; i++){
    assert(a[i-1] <= a[i]);
  }
}

void test_isempty(MinHeap<int>& heap){
  assert(heap.heapSize == 0);
}

void merge(int a[], int s1, int e1, int s2, int e2){
  int b[e1-s1+1];
  int i = s1;
  int j = s2;
  int k = 0;

  while(i <= e1 && j <= e2){
    if (a[i] < a[j]){
      b[k++] = a[i++];
    }else{
      b[k++] = a[j++];
    }
  }

  while(i <= e1){
    b[k++] = a[i++];
  }
  while(j <= e2){
    b[k++] = a[j++];
  }

  i = s1;
  k = 0;
  while(i <= e2){
    a[i++] = b[k++];
  }
}

void mergeSort(int a[], int low, int high){
```

```
  if(low < high){
    int mid = (low+high)/2;
    mergeSort(a,low,mid);
    mergeSort(a,mid+1,high);
    merge(a,low,mid,mid+1,high);
  }
}

void insertionSort(int a[], int n){
  for(int i = 1; i < n; i++){
    int j = i;
    while(j > 0 && a[j] < a[j-1]){
      int temp = a[j];
      a[j] = a[j-1];
      a[j-1] = temp;
      j--;
    }
  }
}

void heapTime(){
  for(int i = 1; i <= 100; i++){
    int n = 10000 + i * 500;
    int heapA[n];
    for(int j = 0; j < n; j++){
      int randNum = rand() % n + 1;
      heapA[j] = randNum;
    }
    MinHeap<int> heapT(heapA, n);

    timeval timeBef, timeAf;
    long diffSec, diffUSec;
    int sortedA[n];

    gettimeofday(&timeBef, NULL);
    heapT.heapSort(sortedA);
    gettimeofday(&timeAf, NULL);

    diffSec = timeAf.tv_sec - timeBef.tv_sec;
    diffUSec = timeAf.tv_usec - timeBef.tv_usec;
    cout << diffSec + diffUSec/1000000.0 << endl;
  }
}

void mergeTime(){
  for(int i = 1; i <= 100; i++){
    int n = 10000 + i * 500;
    int mergeA[n];
    for(int j = 0; j < n; j++){
      int randNum = rand() % n + 1;
      mergeA[j] = randNum;
    }

    timeval timeBef, timeAf;
    long diffSec, diffUSec;


    gettimeofday(&timeBef, NULL);
    mergeSort(mergeA, 0, n);
    gettimeofday(&timeAf, NULL);

    diffSec = timeAf.tv_sec - timeBef.tv_sec;
    diffUSec = timeAf.tv_usec - timeBef.tv_usec;
    cout << diffSec + diffUSec/1000000.0 << endl;
  }
}

void insertionTime(){
  for(int i = 1; i <= 100; i++){
    int n = 10000 + i * 500;
```

```
    int insertA[n];
    for(int j = 0; j < n; j++){
      int randNum = rand() % n + 1;
      insertA[j] = randNum;
    }

    timeval timeBef, timeAf;
    long diffSec, diffUSec;

    gettimeofday(&timeBef, NULL);
    insertionSort(insertA, n);
    gettimeofday(&timeAf, NULL);

    diffSec = timeAf.tv_sec - timeBef.tv_sec;
    diffUSec = timeAf.tv_usec - timeBef.tv_usec;
    cout << diffSec + diffUSec/1000000.0 << endl;
  }
}

int main(){
  int a[] = {1,2,3,4};
  int b[] = {2,1,3,4};
  int c[] = {1,4,3,5,9,8,3,2,6,5};

  MinHeap<int> heap0;
  test_isheap(heap0);
  test_construct(heap0);
  test_isempty(heap0);

  cout << "heap0: " << heap0 << endl;

  MinHeap<int> heap1 = MinHeap<int>(a,4);
  test_isheap(heap1);
  test_construct(heap1);

  MinHeap<int> heap2 = MinHeap<int>(b,4);
  test_isheap(heap2);
  test_construct(heap2);

  cout << "heap1: " << heap1 << endl;
  cout << "heap2: " << heap2 << endl;

  test_same(heap1, heap2);

  MinHeap<int> heap3 = MinHeap<int>(c,10);
  test_isheap(heap3);
  test_construct(heap3);

  cout << "heap3: " << heap3 << endl;

  int s[10];
  heap3.heapSort(s);
  test_heapsort(s, 10);

  cout << "sorted heap3 list: ";
  for(int i = 0; i < 10; i++){
    cout << s[i] << " ";
  }

  cout << endl;


  // heapTime();
  // mergeTime();
  // insertionTime();

  return 0;
}
```