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

#ifndef HEAP_H
#define HEAP_H

#include <iostream>
#include <string>

const int DEFAULT_SIZE = 100;

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

    // Unit tests
    friend void testDefaultConstructor();
    friend void testSwap();
    friend void testInitConstructor();
    friend void testHeapify();
    friend void testBuildHeap();
    friend void testParentChilds();

  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 child
    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
};

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

#include "heap.cpp"

#endif
```

```
/*
Class: CS 271, Fall 2021
Professor: Jessen Havill
Name: Tung Luu, Wilson Le
Date: October 4, 2021
Purpose: Answers to project 3
*/
// heap.cpp

#include <sstream>

/*
The default constructor. Allocate the array A of the heap with size n, set heapSize = 0 and
 capacity to n.
Preconditions: the parameter n is either missing, or n must be a non-negative integer.
Postconditions: Return a new instance of MinHeap whose heapSize is 0 and capacity is either
 n or DEFAULT_SIZE if no parameter n is provided
*/

template <class KeyType>
MinHeap<KeyType>::MinHeap(int n) {
    A = new KeyType[n];
    heapSize = 0;
    capacity = n;
}

/*
Allocate new array A of the heap, copy the content from initA to A and the call buildHeap t
o build a correct heap from A.
Preconditions: n must be a non-negative integer
Postconditions: Return a new instance of MinHeap whose internal array is the same as initA
and capacity is n.
*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType initA[], int n) {
    A = new KeyType[n];
    capacity = n;
    for(int i = 0; i < capacity; i++) {
        A[i] = initA[i];
    }
    buildHeap();
}

/*
Copy constructor
Preconditions: The parameter heap is a correct min heap
Postconditions: Return a new instance of MinHeap which is a copy of the parameter heap
*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap) {
    // Set A = nullptr so that copy method doesn’t delete[] A
    A = nullptr;
    copy(heap);
}

/*
Destructor
Preconditions: None
Postconditions: Completely remove this min heap from memory.
*/
template <class KeyType>
MinHeap<KeyType>::˜MinHeap() {
    destroy();
}

/*
Perform heapsort, return result in the sorted array
Preconditions: This heap is a correct min heap.
```

```
Postconditions: The array sorted is sorted
*/

template<class KeyType>
void MinHeap<KeyType>::heapSort(KeyType sorted[]) {
    int initialHeapSize = heapSize;

    for(int i = heapSize - 1; i >= 1; i--) {
        swap(0, i);
        heapSize--;
        heapify(0);
    }

    // Change the heapSize back into its initial value
    heapSize = initialHeapSize;

    // Now A is sorted in reverse order. We just need copy A to sorted in reverse order.
    for(int i = 0; i < heapSize; i++) {
        sorted[i] = A[heapSize - i - 1];
    }

    // Reverse A to turn A back into a correct heap
    for(int i = 0; i < heapSize; i++) {
        A[i] = sorted[i];
    }

}

/*
Heapify subheap rooted at index
Preconditions: Two subtrees at root index are correct min heaps
Post conditions: The tree at root index is a correct min heap.
*/
template<class KeyType>
void MinHeap<KeyType>::heapify(int index) {
    heapifyR(index);
}

/*
Build heap from the array A
Preconditions: None
Postconditions: This heap is a correct min heap.
*/
template <class KeyType>
void MinHeap<KeyType>::buildHeap() {
    heapSize = capacity;
    // Iterate from the node (capacity / 2) - 1 to 0, which is iterating in bottom-up metho
d, and call heapify in each iteration
    for(int i = (capacity / 2) - 1; i >= 0; i--) {
        heapify(i);
    }
}

/*
Heapify subheap rooted at index using recursive method
Preconditions: Two subtrees at root index are correct min heaps
Post conditions: The tree at root index is a correct min heap.
*/
template<class KeyType>
void MinHeap<KeyType>::heapifyR(int index) {
    int left = leftChild(index);
    int right = rightChild(index);
    int smallestChild = index;

    if(left < heapSize && A[left] < A[index]) {
        smallestChild = left;
    }
    if(right < heapSize && A[right] < A[smallestChild]) {
        smallestChild = right;
    }

    if(smallestChild != index) {
        swap(index, smallestChild);
        heapifyR(smallestChild);
    }
}

/*
Heapify subheap rooted at index using iterative method
Preconditions: Two subtrees at root index are correct min heaps
Post conditions: The tree at root index is a correct min heap.
*/
template<class KeyType>
void MinHeap<KeyType>::heapifyI(int index) {
    while(leftChild(index) < heapSize) {
        int left = leftChild(index);
        int right = rightChild(index);
        int smallestChild = index;

        if(left < heapSize && A[left] < A[index]) {
            smallestChild = left;
        }

        if(right < heapSize && A[right] < A[smallestChild]) {
            smallestChild = right;
        }

        if(smallestChild != index) {
            swap(index, smallestChild);
            index = smallestChild;
        }
        else {
            break;
        }
    }
}

/*
Swap two elements at index1 and index2 in A
Preconditions: index1 and index2 are valid element in the array A
Postconditions: two elements at index1 and index2 are swapped
*/
template <class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2) {
    KeyType temp = A[index1];
    A[index1] = A[index2];
    A[index2] = temp;
}

/*
Copy the parameter heap to this heap
Preconditions: The parameter heap is a correct min heap
Postconditions: This heap becomes a copy of the parameter heap
*/
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap) {
    destroy();

    heapSize = heap.heapSize;
    capacity = heap.capacity;
    A = new KeyType[capacity];
    for(int i = 0; i < heapSize; i++) {
        A[i] = heap.A[i];
    }
}
```

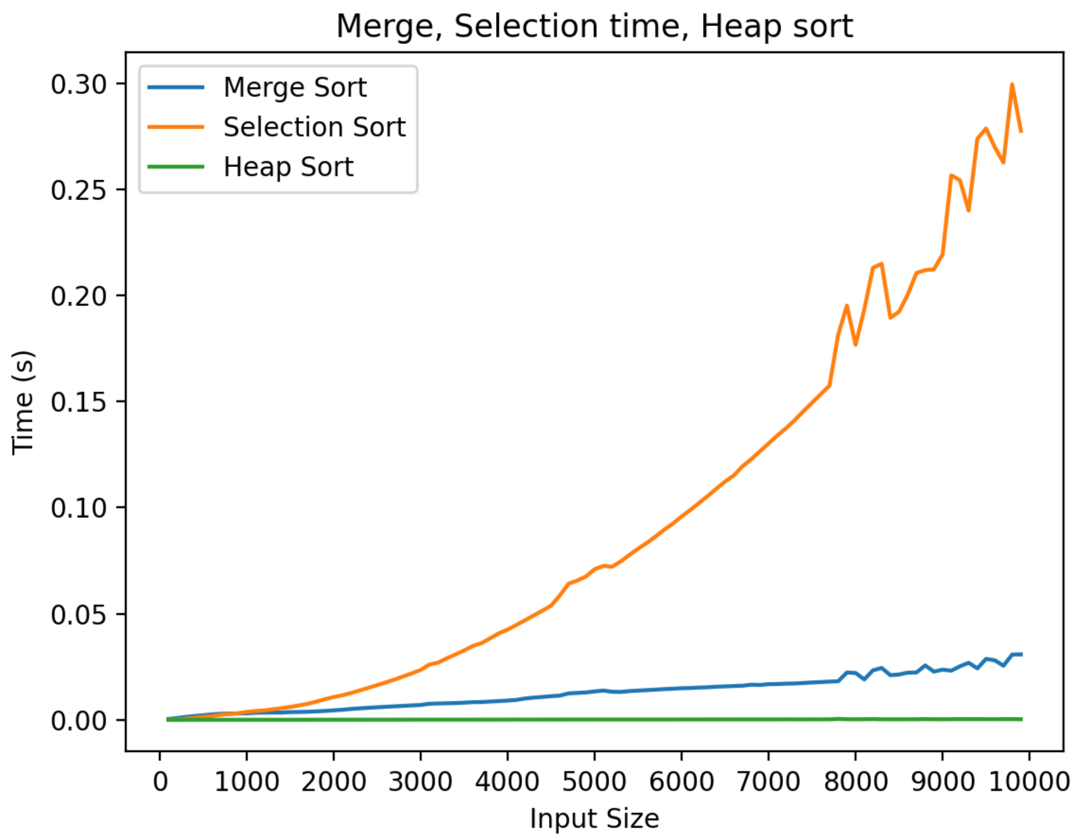
```
/*
Destroy is a function to deallocate this heap.
Preconditions: None.
Postconditions: The array A of this heap is removed from memory and instance variables heap
Size and capacity are set to 0.
*/
template <class KeyType>
void MinHeap<KeyType>::destroy() {
    if(A != nullptr) {
        delete[] A;
    }
    heapSize = 0;
    capacity = 0;
}

/*
Assignment operator.
Preconditions: The parameter heap is a correct min heap.
Postconditions: if this heap and the parameter heap is not the same (having the same addres
s), this heap becomes a copy of the parameter heap
*/
template<class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap) {
    if(this != &heap) {
        copy(heap);
    }
    return *this;
}

// Use the following toString() for testing purposes.
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();
}
```

```
/*
Class: CS 271, Fall 2021
Professor: Jessen Havill
Name: Tung Luu, Wilson Le
Date: October 4, 2021
Purpose: Answers to project 3
*/

#include<string>
#include <assert.h>
#include "heap.h"
using namespace std;

void testDefaultConstructor(){
        MinHeap<int> heap1 = MinHeap<int>();
        assert(heap1.capacity == 100);
        assert(heap1.heapSize == 0);
        MinHeap<int> heap2 = MinHeap<int>(50);
        assert(heap2.capacity == 50);
        assert(heap2.heapSize == 0);
        cout << "Default Constructor Passed" << endl;
}

void testParentChilds(){
        int initA[3] = {0, 1, 2};
        MinHeap<int> heap = MinHeap<int>(initA, 3);
        assert(heap.parent(1) == 0);
        assert(heap.parent(2) == 0);
        assert(heap.leftChild(0) == 1);
        assert(heap.rightChild(0) == 2);
        cout << "Test Parent Childs Passed" << endl;
}

void testSwap() {
        int n = 5;
    int initA[5] = {0, 1, 2, 3, 4};
    MinHeap<int> heap = MinHeap<int>(5);
    for(int i = 0; i < n; i++) {
        heap.A[i] = initA[i];
    }
    heap.heapSize = n;
    heap.swap(1,4);
    assert(heap.toString() == "[0, 4, 2, 3, 1 | ]");
    cout << "Test Swap passed" << endl;
}

void testHeapify(){
        int n = 5;
        int initA[5] = {0, 1, 2, 3, 4};
        MinHeap<int> heap = MinHeap<int>(n);
    for(int i = 0; i < n; i++) {
        heap.A[i] = initA[i];
    }
    heap.heapSize = n;
    heap.A[0] = 9;
        heap.heapify(0);
        assert(heap.toString() == "[1, 3, 2, 9, 4 | ]");
        cout << "Heapify " << heap.toString() << " Passed" << endl;
}

void testBuildHeap(){
        int n = 5;
        int initA[5] = {4,3,2,1,0};
        MinHeap<int> heap = MinHeap<int>(initA, n);
    for(int i = 0; i < n; i++) {
        heap.A[i] = initA[i];
    }
        heap.buildHeap();
        assert(heap.toString() == "[0, 1, 2, 4, 3 | ]");
        cout << "Build Heap " << heap.toString() << " Passed" << endl;
}

void testInitConstructor(){
        int initA[5] = {4,3,2,1,0};
        MinHeap<int> heap1 = MinHeap<int>(initA, 5);
        assert(heap1.capacity == 5);
        assert(heap1.heapSize == 5);
    assert(heap1.toString() == "[0, 1, 2, 4, 3 | ]");
        cout << "Init Constructor Passed" << endl;
}

// If this function is correct, then that means the copy function is also correct. Therefor
e, we don’t need to test the copy function
void testCopyConstructor(){
        int initA[5] = {0, 1, 2, 3, 4};
        MinHeap<int> heap1 = MinHeap<int>(initA, 5);
        MinHeap<int> heap2 = MinHeap<int>(heap1);
        assert(heap1.toString() == heap2.toString());
        cout << "Copy Constructor Passed" << endl;
}

void testAssignmentOperator(){
        int initA[5] = {0, 1, 2, 3, 4};
        MinHeap<int> heap1 = MinHeap<int>(initA, 5);
        MinHeap<int> heap2 = heap1;
        assert(heap1.toString() == heap2.toString());
        cout << "Assignment Operator Passed" << endl;
}

void testHeapSort(){
        int initA[10] = {10, 8, 9, 1, 2, 4, 5, 3, 7, 6};
        int n = 10;
        MinHeap<int> heap = MinHeap<int>(initA, n);
        heap.heapSort(initA);
        assert(heap.toString() == "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10 | ]");
        cout << "Heap Sort " << heap.toString() << " Passed" << endl;
}

int main(){
        testDefaultConstructor();
        testParentChilds();
    testSwap();
        testHeapify();
        testBuildHeap();

        testInitConstructor();
        testCopyConstructor();
        testAssignmentOperator();
        testHeapSort();
        return 0;
}
```

CS271 Project 3

Wilson Le, Tung Luu

October 2021



- Asymptotic time complexity of the heap sort algorithm on an array that is already sorted:
  - $\Theta(n \log n)$
- Asymptotic time complexity on an array that is in reverse order:
  - $\Theta(n \log n)$
- Best case asymptotic time complexity of heap sort
  - $\Theta(n)$
- What kind of input does the best case asymptotic time complexity occur
  - The best case asymptotic time complexity of heap sort will be $\Theta(1)$ in the case that all elements in the input array are the same. First, the buildHeap method will take $\Theta(n)$ when we initialize the heap. When we call heapSort method, since all elements are the same, the heapify method will take $\Theta(1)$ in each iteration of n iteration, since heapify doesn't need to move any element down the heap. Therefore the asymptotic time complexity of heap sort in this case is $\Theta(1)$.