

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

using namespace std;

//=====
// Default Constructor for MinHeap
// Pre-conditions:
//     none
// post-conditions:
//     It's a heap with nothing in it,
//     capacity n, heapSize 0
// Notes:
//     Now go out into the world. Don't return
//     until you have done all that is required.
//=====
template<class KeyType>
    MinHeap<KeyType>::MinHeap(int n)
{
    A = new KeyType[n];
    capacity = n;
    heapSize = 0;
}

//=====
// Array Initilazation Constructor for MinHeap
// Pre-Conditions:
//     n > 0
// Post-Conditions:
//     Congratulations! It's a MinHeap,
//     capacity n, heapSize n
// Notes:
//     Now go out into the world. Don't return
//     until you have done all that is required.
//=====
template<class KeyType>
    MinHeap<KeyType>::MinHeap(KeyType initA[], int n)
{
    capacity = n;
    heapSize = n;
    A = new KeyType[n];
    for(int i = 0; i < n; i++)
    {
        A[i] = initA[i];
    }
    buildHeap();
}

//=====
// Copy Constructor for MinHeap
// Pre-Conditions:
//     none
// Post-Conditions:
//     Congratulations! It's a MinHeap,
//     capacity heap.capacity,
//     heapSize heap.heapSize
// Notes:
//     Now go out into the world. Don't return
//     until you have done all that is required.
//=====
template<class KeyType>
    MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)
```

```
{
    copy(heap);
}

//=====
// Destructor for MinHeap
// Pre-Conditions:
//     None
// Post-Conditions:
//     None
// Notes:
//     And when you do return, I am here. And I
//     will destroy you. It is all I know.
//=====
template<class KeyType>
MinHeap<KeyType>::~MinHeap()
{
    destroy();
}

//=====
// Sorting Algorithm: heapSort
// Pre-Conditions:
//     The heap must be a MinHeap
// Post-Conditions:
//     sorted is now sorted in ascending order
//=====
template<class KeyType>
void MinHeap<KeyType>::heapSort(KeyType sorted[])
{
    sorted = new KeyType[capacity];
    //buildHeap();
    for(int i = capacity - 1; i >= 0; i--)
    {
        sorted[i] = A[0];
        swap(0,i);
        heapSize--;
        heapify(0);
    }
    heapSize = capacity;
}

//=====
// Assignment operator
// Pre-Conditions:
//     none
// Post-Conditions:
//     returns a new heap just like the heap
//     which was passed in
//=====
template<class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{
    destroy();
    copy(heap);
    return *this;
}

//=====
// String converter
// Pre-Conditions:
//     none
// Post-Conditions:
```

```
//          returns a string of the array in which
//          the heap is stored
//=====
/*
template<class KeyType>
string MinHeap<KeyType>::toString() const
{
    stringstream stm;
    stm << "{";
    for(int i = 0; i < heapSize - 1; i++)
    {
        stm << A[i] << ", ";
    }
    if(heapSize != 0) //make sure that the heap is not empty to avoid invalid indexing.
        stm << A[heapSize - 1] << "}";
    else
        stm << "}";

    return stm.str();
}
*/
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();
}
//=====
// makes a heap into a min heap
// Pre-Conditions:
//          Both children must be roots of a Min-Heap
// Post-Conditions:
//          The heap is a Min-Heap (if the
//          Pre-Condition is satisfied)
//=====
template<class KeyType>
void MinHeap<KeyType>::heapify(int index)
{
    int l = leftChild(index);
    int r = rightChild(index);
```

```
    int min;
    if(l < heapSize && A[index] > A[l])
        min = l;
    else
        min = index;
    if(r < heapSize && A[min] > A[r])
        min = r;
    if(min != index) //will do nothing if the value is already smaller than its children
    {
        swap(index, min);
        heapify(min);
    }
}

//=====
// builds a heap
// Pre-Conditions:
//     none
// Post-Conditions:
//     the heap is definitely a Min-Heap
//=====
template<class KeyType>
void MinHeap<KeyType>::buildHeap()
{
    heapSize = capacity;
    for(int i = heapSize / 2 - 1; i >= 0; i--)
    {
        heapify(i);
    }
}

//=====
// Swaps two items
// Pre-Conditions:
//     The indices are valid
// Post-Conditions:
//     The values at the indices
//     have been swapped
//=====
template<class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2)
{
    KeyType temp = A[index1];
    A[index1] = A[index2];
    A[index2] = temp;
}

//=====
// copies one heap into another
// Pre-Conditions:
//     none
// Post-Conditions:
//     This heap is just like the one passed in.
//     capacity heap.capacity,
//     heapSize heap.heapSize
//=====
template<class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap)
{
    A = new KeyType[heap.capacity];
    for(int i = 0; i < heap.capacity; i++)
        A[i] = heap.A[i];
    capacity = heap.capacity;
```

```
        heapSize = heap.heapSize;
    }

//=====
//
// Pre-Conditions:
//     none
// Post-Conditions:
//     none
// Notes:
//     But don't feel betrayed or singled out.
//     I heed nothing. I leave nothing. All are
//     destroyed. Everything shall be deleted.
//=====
template<class KeyType>
void MinHeap<KeyType>::destroy()
{
    delete A;
}
```

```
#include <iostream>
#include <cassert>
#include "heap.h"

using namespace std;

void test_constructor()
{
    MinHeap<int> heap(0);
    string str = heap.toString();
    assert(str == "[ ]");
}

void test_array_constructor()
{
    int a[5] = {1,2,3,4,5};
    MinHeap<int> heap(a, 5);
    string str = heap.toString();
    assert(str == "[1, 2, 3, 4, 5 | ]");
}

void test_copy_constructor()
{
    int a[5] = {1,2,3,4,5};
    MinHeap<int> heap(a, 5);
    MinHeap<int> heap2(heap);
    string str = heap2.toString();
    assert(str == "[1, 2, 3, 4, 5 | ]");
}

void test_heapSort()
{
    int a[5] = {3, 2, 4, 1, 5};
    MinHeap<int> heap(a, 5);
    int b[5];
    heap.heapSort(b);
    MinHeap<int> heap2(b, 5);
    string str = heap2.toString();
    assert(str == "[1, 2, 3, 4, 5 | ]");
}

void test_assignment()
{
    int a[6] = {1,2,3,4,5,6};
    MinHeap<int> heap(a, 6);
    MinHeap<int> heap2 = heap;
    string str = heap2.toString();
    assert(str == "[1, 2, 3, 4, 5, 6 | ]");
}

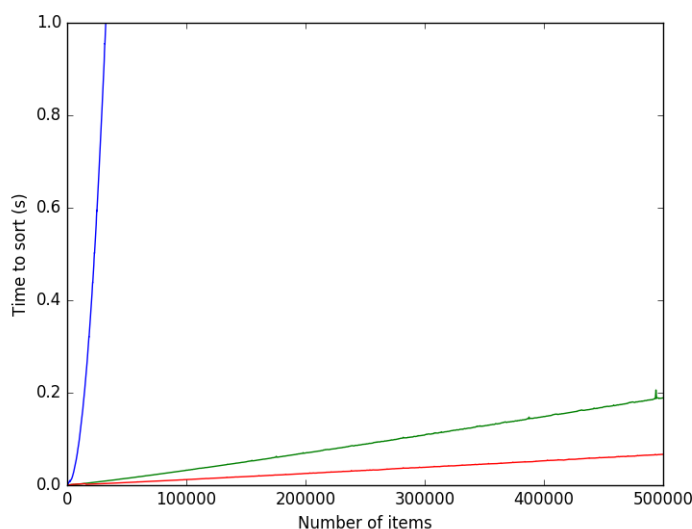
void test_heapify()
{
    int a[3] = {3,1,2};
    MinHeap<int> heap(a, 3);
    heap.heapify(0);
    string str = heap.toString();
    assert(str == "[1, 3, 2 | ]");
}

/*
void test_buildHeap()
{
    int a[5] = {2,1,4,5,3};
    MinHeap<int> heap(a,5);
    heap.buildHeap();
    string str = heap.toString();
    cout << str << endl;
}
```

```
}
*/
void test_swap()
{
    int a[7] = {1, 2, 3, 4, 5, 6, 7};
    MinHeap<int> heap(a, 7);
    heap.swap(0, 1);
    string str = heap.toString();
    assert(str == "[2, 1, 3, 4, 5, 6, 7 | ]");
}

int main ( void )
{
    test_constructor();
    test_array_constructor();
    test_copy_constructor();
    test_heapSort();
    test_assignment();
    test_heapify();
    //test_buildHeap();
    test_swap();
}
```

1. In a min-heap, the children of a given node will be larger than the node. Thus, the largest element in the entire heap will be located at a leaf node. However, it will not necessarily be the last leaf node.
2. Yes, if the array is sorted in ascending order. In a sorted array, for  $j > i$ ,  $A[j] > A[i]$ . In a heap, the indices of the children are greater than the index of the parent. This implies in a sorted array, that the values of the children are greater than the value of the parent, which is the defining feature of a min-heap.
3. See code.
4. Graph of Array size vs Sort Time  
Blue is Insertion Sort  
Green is Heap Sort  
Red is Merge Sort



5. The asymptotic time complexity of heap sort on an array that is already in sorted order is  $O(n \log n)$ . This is because the algorithm will go through every item in the array ( $\Theta(n)$ ) and call heapify which has time complexity of  $O(\log n)$ .  
The time complexity of heap sort on an array in reverse order will still be  $O(n \log n)$  because it still has to go through every element in the array ( $\Theta(n)$ ) and call heapify ( $O(\log n)$ ).  
The best case asymptotic time occurs when every item is equal and is  $\Theta(n)$ . This is because the algorithm will always go through every element  $\Theta(n)$  but every call to heapify which will be  $\Theta(1)$  because it will only have one iteration as no item will ever be greater than its children.