Heap Source Code

By Eric Suh

This source code is an implementation of the Heap Tree class and the Heap Sort algorithm. The class is implemented with templates.

For the templated class, the elements must have the operators >, =, and < defined.

To use the Heap sort that is built into the class, two separate steps must be taken. The first is to call the constructor, which organizes the array into a heap:

```
HeapTree<TYPE> HeapName(Array, Num, MaxNum);
```

TYPE is the data type of the elements, Array is the actual array to be sorted, and Num is the number of elements that are to be sorted. MaxNum normally sets the limit on the number of data nodes that the Heap can have. If you are only using the heap for sorting, you can set MaxNum to Num. (However, MaxNum should not be set to anything less than Num).

When the constructor is called, the Heap copies the Array. Thus, neither the Array variable nor what it points to will be modified by the Heap.

The second step is to call the actual sort, which will organize the heap into a sorted array:

```
NewArray *Sort();
```

This Sort() function will return a pointer to another array which is sorted. Any modifications done to NewArray or its contents will not affect the heap.

```
/*
     -------------------------------------------------------------------
    |                                                                   |
    |    HeapTree Class                                                 |
    |    ===========================================================    |
    |    This HeapTree Class has been implemented with templates.       |
    |                                                                   |
    |    To use the HeapSort that is built into the class, two          |
    |    separate steps must be taken. The first is the constructor:    |
    |                                                                   |
    |        HeapTree<Type> HeapName(Array, Num, MaxNum);               |
    |                                                                   |
    |    'Type' is the data type of the Array elements, 'Array' is a    |
    |    standard C++ array to be sorted and 'Num' is the number of     |
    |    elements in the array. MaxNum sets the limit on the number     |
    |    of data nodes that the Heap can have. If you are only using    |
    |    the heap for sorting, you can set MaxNum to Num. (However,     |
    |    MaxNum should not be set less than Num).                       |
    |                                                                   |
    |    When the constructor is called, the Heap *copies* the Array.   |
    |    Thus, neither the Array variable nor what it points to will    |
    |    be modified.                                                   |
    |                                                                   |
    |    The second step is to call the actual sort:                    |
    |                                                                   |
```

```
    |        NewArray *Sort();                                       |
    |                                                                |
    |    This sort will return a pointer to another array, which is  |
    |    the sorted array. Any modifications done to NewArray or its |
    |    contents will not affect the heap.                          |
    |                                                                |
     ---------------------------------------------------------------
*/

#ifndef __HeapTreeClassH__
#define __HeapTreeClassH__

#include <assert.h>    // For error-checking purposes

//-------------------------------------------------
// Main structure of HeapTree Class:
//-------------------------------------------------

template <class Elem>
class HeapTree
{
  public:
    HeapTree(int MaxSize=500);
    HeapTree(const HeapTree<Elem> &OtherTree);
    HeapTree(Elem *Array, int ElemNum, int MaxSize);
    Elem *Sort(void); // Built-in HeapSort Algorithm
    ~HeapTree(void);

    bool Add(const Elem &Item);  // Add the Item to Heap
    Elem Remove(void);           // Remove and return Item from Heap

    inline int GetSize(void);    // Returns the number of nodes in the Heap

  protected:
    Elem     *Data;              // Actual Data array
    int       CurrentNum;        // Current number of elements
    const int MAX_SIZE;          // The maximum number of elements

    void ShiftUp(int Node);      // Shift Node up into place
    void ShiftDown(int Node);    // Shift Node down into place

    inline int ParentOf(int Node);      // Returns Parent location
    inline int LeftChildOf(int Node);   // Returns Left Child location
};

//-------------------------------------------------
// Implementation of HeapTree Class:
//-------------------------------------------------

// HeapTree constructor function
template <class Elem>
HeapTree<Elem>::HeapTree(int MaxSize)
    : MAX_SIZE(MaxSize)
{
  Data       = new Elem[MAX_SIZE];
  CurrentNum = 0;
}

// HeapTree copy constructor function
template <class Elem>
HeapTree<Elem>::HeapTree(const HeapTree<Elem> &OtherTree)
```

```
    : MAX_SIZE(OtherTree.MAX_SIZE)
{
  Data       = new Elem[MAX_SIZE];
  CurrentNum = OtherTree.CurrentNum;

  // Copy the array
  for (int i = 0; i < OtherTree.CurrentNum; ++i)
    Data[i] = OtherTree.Data[i];
}

// HeapTree array constructor
template <class Elem>
HeapTree<Elem>::HeapTree(Elem *Array, int ElemNum, int MaxSize)
    : MAX_SIZE(MaxSize)
{
  Data       = new Elem[MAX_SIZE];
  CurrentNum = ElemNum;

  // This copies the array into the heap's internal array
  for (int i = 0; i < ElemNum; ++i)
        Data[i] = Array[i];

  // This organizes the Array into a proper HeapTree
  for (int i = ParentOf(CurrentNum - 1); i >= 0; --i)
    ShiftDown(i);
}

// Built-in Heap Sort algorithm
template <class Elem>
Elem *HeapTree<Elem>::Sort(void)
{
  // This is the array that will be returned
  Elem *NewArray = new Elem[CurrentNum];

  // The algorithm works back to front, with the sorted
  // elements being stored in NewArray
  for (int ElemNum = CurrentNum-1; ElemNum >=0; --ElemNum)
  {
    // Since the Remove() function alters CurrentNum by subtracting 1
    // from it each time, we must use a seperate variable to
    // index NewArray.
    NewArray[ElemNum] = Remove();
  }
  return NewArray;
}

// HeapTree destructor function
template <class Elem>
HeapTree<Elem>::~HeapTree(void)
{
  if (Data)
    delete Data;
}

// Add() function
template <class Elem>
bool HeapTree<Elem>::Add(const Elem &Item)
{
  if (CurrentNum >= MAX_SIZE)    // If we have reached our maximum capacity
    return false;
  Data[ CurrentNum ] = Item;
```

```
  ShiftUp(CurrentNum++);
  return true;
}

// Remove() function
template <class Elem>
Elem HeapTree<Elem>::Remove(void)
{
  assert(CurrentNum > 0);

  Elem Temp = Data[0];
  Data[0] = Data[--CurrentNum];  // Replace with the last element
  ShiftDown(0);
  return Temp;
}

// GetSize() function
template <class Elem>
inline int HeapTree<Elem>::GetSize(void)
{
  return CurrentNum;
}

// ShiftUp() function
template <class Elem>
void HeapTree<Elem>::ShiftUp(int Node)
{
  int  Current = Node,
       Parent  = ParentOf(Current);
  Elem Item    = Data[Current];

  while (Current > 0)  // While Current is not the RootNode
  {
    if (Data[Parent] < Item)
    {
      Data[Current] = Data[Parent];
      Current = Parent;
      Parent = ParentOf(Current);
    }
    else
      break;
  }
  Data[Current] = Item;
}

// ShiftDown() function
template <class Elem>
void HeapTree<Elem>::ShiftDown(int Node)
{
  int Current = Node,
      Child   = LeftChildOf(Current);
  Elem Item   = Data[Current];    // Used to compare values

  while (Child < CurrentNum)
  {
    if (Child < (CurrentNum - 1))
      if (Data[Child] < Data[Child+1])  // Set Child to largest Child node
        ++Child;

    if (Item < Data[Child])
    {    // Switch the Current node and the Child node
```

```
      Data[Current] = Data[Child];
      Current       = Child;
      Child         = LeftChildOf(Current);
    }
    else
      break;
  }
  Data[Current] = Item;
}

// ParentOf() function
template <class Elem>
inline int HeapTree<Elem>::ParentOf(int Node)
{
  assert(Node > 0);
  // This uses the fact that decimals are truncated during
  // the division of integers. Thus, (12 - 1) / 2 == 5
  return (Node - 1) / 2;
}

// LeftChildOf() function
template <class Elem>
inline int HeapTree<Elem>::LeftChildOf(int Node)
{
  return (Node * 2) + 1;
}

#endif /*__HeapTreeClassH__*/
```

Related Articles

More About Binary Trees