**Heap & HeapSort**

**HeapSort Algorithm**

|  |
| --- |
| **Max heap** is a special type of binary tree .The root of the max heap is greater than its child roots.  To **sort** a heap  Build Heap algorithm is used to build a heap out of the data set .Then remove the root element and replace the last element at the position of root node. Then rearrange the heap. Place the root node in an array. Follow these steps until all elements in heap are not placed into an array. The values in array will be in sorted order. **http://www.roseindia.net/java/beginners/arrayexamples/heap2.gifhttp://www.roseindia.net/java/beginners/arrayexamples/heap3.gifhttp://www.roseindia.net/java/beginners/arrayexamples/heap4.gifhttp://www.roseindia.net/java/beginners/arrayexamples/heap5.gifhttp://www.roseindia.net/java/beginners/arrayexamples/heap6.gifhttp://www.roseindia.net/java/beginners/arrayexamples/heap7.gif…** |

**HeapSort Efficiency**

|  |
| --- |
| Heapsort runs in **O(N\*logN) time**. Slower than quicksort, butit is less sensitive to the initial distribution of data. Sometimes, quicksort runs in O(N2) time, but heapsort runs in O(N\*logN) time no matter how the data is distributed. |

**The heapSort Method**

|  |
| --- |
| **public** **static** **void** heapSort(**int**[] a)  {  ***heapify***(a, a.**length**); //first place a in max-heap order  **for**(**int** i = a.**length** - 1; i > 0; i--)  {  **int** temp = a[i]; // swap the root with the least element  a[i] = a[0];  a[0] = temp;  ***siftDown***(a, 0, i - 1); //put the heap back in max-heap order  }  } |

**The heapify Method**

|  |
| --- |
| Start is the index of the last parent node. Sift down the node at index start to the proper place that all nodes below the start are in heap order |
| **public** **static** **void** heapify(**int**[] a, **int** count)  {  **for**(**int** start = (count - 2) / 2; start >= 0; start--)  *siftDown*(a, start, count - 1);  } |

**The siftDown Method**

|  |
| --- |
| **public** **static** **void** siftDown(**int**[] a, **int** start, **int** end)  {  **int** root = start; //end represents the limit of how far down the heap to sift  **while**((root \* 2 + 1) <= end) //While the root has at least one child  {  **int** child = root \* 2 + 1; //root\*2+1 points to the left child  //if the child has a sibling and the child's value is less than its sibling's...  **if**(child + 1 <= end && a[child] < a[child + 1])  child = child + 1; //... then point to the right child instead  **if**(a[root] < a[child]) //out of max-heap order  {  **int** tmp = a[root];  a[root] = a[child];  a[child] = tmp;  root = child; //repeat to continue sifting down the child now  } **else** **return**;  }  } |