Heap

1. **Keep the heap property(assuming it is all max heap)**

Function: heapfy()

1. **Algorithm:**

1. If left > root, swap(left, root), or if right > root, swap(right, root)
2. Recursively call heapfy() on the swap the child node till end so the heap property is reserved.
3. **Code**

// n size of array, i is start index.

Void heapfy(int[] arr, int n, int i)

{

int largest = i;

int l = 2i+1; // binary tree index relations

int r = 2i+2;

if(l<n && arr[l] > arr[largest])

largest = i;

if(r<n && arr[r]> arr[largest])

largest = r;

//swap

If(largest != i)

{

swap(arr[i], arr[largest]);

heapfy(arr, n, largest);

}

}

1. **Time complexity**

h is the height of the tree

O(h) = O(log\_2 n)

1. **Build Heap**
2. **Algorithm**

do heapfy() on all the nodes that have children, these nodes have index(n/2-1, 0) (based on binary tree property)

1. **Code**

void buildHeap(int[] arr, int n)

{

for(int i=n/2 -1; i>=0; i--)

{

heapfy(arr, n, i);

}

}

1. **Time Complexity**

Time = \sum \_{h=0} ^ {log\_2n} [\frac{n}{2^{h+1}}] O(h)

O(h) : time to call one heapfy()

[\frac{n}{2^{h+1}}]: number of nodes on the same tree level

\sum \_{h=0} ^ {log\_2^n} : sum over all the levels.

So time = O(n \sum\_{h=0}^(lgn) h/(2^h))

= O(2n)

=O(n)

**3．Heapsort**

1. **Algorithm**
2. Build the heap
3. Start from the last element(index n-1),
4. swap it with the first element. Then the last element becomes sorted
5. Do heapfy on the unsorted elements
6. Move to the next unsorted element, in this case it is the last but two(index n-2), repeat 3) and 4) until all elements become sorted.
7. **Code**

Void heapsort(int arr[], int n)

// build heap

void buildHeap(arr, n);

for(int i = n-1; i>=0; i--)

{

swap(arr[0], arr[i]);

heapfy(arr, i, 0);

}

1. **Time complexity**
2. Build heap O(n)
3. Sort nlog\_2 n
4. **Comparison**
5. Compare to quicksort

|  |  |  |  |
| --- | --- | --- | --- |
|  | QS | HS | comment |
| Average Speed | Faster |  | 12nlogn vs 16 nlog n |
| Worst Speed |  | Faster | n^2 vs nlogn |
|  |  |  |  |

1. Compare to mergesort

|  |  |  |  |
| --- | --- | --- | --- |
|  | MS | HS | comment |
| Space | O(n) | O(1) | MS needs additional space for merge |
| Stable **s**ort | Yes |  | MS keeps order for same element |
| Better Cache performance | Yes |  | MS accesses the caches that are near to each other. |
|  |  |  |  |

1. **Binary heap time complexity analysis**
2. **Time complexity table**

|  |  |  |
| --- | --- | --- |
|  | Average | Worst |
| Search | O(n) | O(n) |
| Insert | O(1) | O(h) = O(long(n)) |
| Delete | O(log(n)) | O(long(n)) |
| Peek | O(1) | O(1) |

1. **Insert**
2. Add the element to the bottom level of the heap
3. Compare the added element with its parent

If they are in correct order, stop

1. If not swap the element with its parent and return to the 2) by keeping comparing the parent
2. **Insert average time complexity**

Assuming a uniform distribution of numbers, which means for any element in the heap, it has a one-half chance of being greater than its parent. And it has one-fourth chance of being greater than its grandparent. So the expected number of swap during the insertion is

Probability of swapping with 1st parent \* number of swap + Probability of swapping with 2st parent \* number of swap + …+ Probability of swapping with mst parent

= ½ \* 1 + ¼ \* 2 + 1/8\* 3 + ½^m \* m

= \sum \frac{m}{2^m}

= 2 when m goes to \inf

Therefore the averaged time complexity is O(1)