GeeksQuiz

Computer science mock tests for geeks

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2.

Heap Sort Algorithm for sorting in increasing order:
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
                4(0)
/   \
     10(1)   3(2)
            /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
 4(0)
/   \
            10(1)    3(2)
           /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
/  \
     5(1)  3(2)
           /   \
         4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

```
// C implementation of Heap Sort
#include <stdio.h>
#include <stdlib.h>

// A heap has current size and array of elements
struct MaxHeap
{
    int size;
    int* array;
};

// A utility function to swap to integers
void swap(int* a, int* b) { int t = *a; *a = *b;  *b = t; }

// The main function to heapify a Max Heap. The function
// assumes that everything under given root (element at
// index idx) is already heapified
void maxHeapify(struct MaxHeap* maxHeap, int idx)
{
    int largest = idx;  // Initialize largest as root
    int left = (idx << 1) + 1;  // left = 2*idx + 1
    int right = (idx + 1) << 1; // right = 2*idx + 2

    // See if left child of root exists and is greater than
    // root
    if (left < maxHeap->size &&
        maxHeap->array[left] > maxHeap->array[largest])
        largest = left;
```

```
    // See if right child of root exists and is greater than
    // the largest so far
    if (right < maxHeap->size &&
        maxHeap->array[right] > maxHeap->array[largest])
        largest = right;

    // Change root, if needed
    if (largest != idx)
    {
        swap(&maxHeap->array[largest], &maxHeap->array[idx]);
        maxHeapify(maxHeap, largest);
    }
}

// A utility function to create a max heap of given capacity
struct MaxHeap* createAndBuildHeap(int *array, int size)
{
    int i;
    struct MaxHeap* maxHeap =
              (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = size;   // initialize size of heap
    maxHeap->array = array; // Assign address of first element of array

    // Start from bottommost and rightmost internal mode and heapify all
    // internal modes in bottom up way
    for (i = (maxHeap->size - 2) / 2; i >= 0; --i)
        maxHeapify(maxHeap, i);
    return maxHeap;
}

// The main function to sort an array of given size
void heapSort(int* array, int size)
{
    // Build a heap from the input data.
    struct MaxHeap* maxHeap = createAndBuildHeap(array, size);

    // Repeat following steps while heap size is greater than 1.
    // The last element in max heap will be the minimum element
    while (maxHeap->size > 1)
    {
        // The largest item in Heap is stored at the root. Replace
        // it with the last item of the heap followed by reducing the
        // size of heap by 1.
        swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);
        --maxHeap->size;  // Reduce heap size

        // Finally, heapify the root of tree.
        maxHeapify(maxHeap, 0);
    }
}

// A utility function to print a given array of given size
void printArray(int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr)/sizeof(arr[0]);
```

```
    printf("Given array is \n");
    printArray(arr, size);

    heapSort(arr, size);

    printf("\nSorted array is \n");
    printArray(arr, size);
    return 0;
}
```

Output:

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

Notes:
Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort
1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice.
Nevertheless, the Heap data structure itself is enormously used. See Applications of Heap Data Structure

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Category: Searching and Sorting

Recommend Share Sort by Best

Join the discussion…

vaibhav gupta • 10 months ago
In the starting of example its 10(1) i.e. 10 at position 1, but its showing 0(1)
4 • Reply • Share ›

GeeksforGeeks Mod vaibhav gupta • 4 months ago
Thanks for pointing this out. We have updated the example.
• Reply • Share ›

dnrajesh • a year ago
createAndBuildHeap() is calling maxheapify() in loop for internal nodes. Time complexity of maxheapify() is O(logn) but time complexity of createAndBuildHeap() is said to be O(n), how? can anyone explain?
2 • Reply • Share ›

itengineer dnrajesh • 9 months ago
funcation maxheapify will run for either left or right child in the tree ... so O(log n)

but the createAndBuildHeap() will be executed for all elements in the array...so O(n)

concludes O(n log n)...hope this helps.
1 • Reply • Share ›

Jerry Goyal • 7 days ago
complexity :-
complexity of function createAndBuildHeap is indeed O(n) because of choosing "bottommost and rightmost internal node" instead of last node(which has no child)

read more:http://stackoverflow.com/quest...
http://www.geeksforgeeks.org/g...

so overall is n+(nlogn) ~ (nlogn)

let me know if I'm wrong
• Reply • Share ›

Asit Srivastava • 8 days ago
Please describe well ...
• Reply • Share ›

skeller88 • 2 months ago

I'm still not 100% confident in my understanding about how createAndBuildHeap() has a time complexity of O(n). Wikipedia has a good summary in its heap sort page, that seems to indicate that siftdown (which is what heapify in the example above is implemented as) has an average O(1) time complexity when siftdown is executed in the loop in createAndBuildHeap(), because the majority of nodes are deep nodes requiring 0 or 1 swaps.

But that time complexity seems unlikely because siftdown has an average time complexity of O(log n). So what's the right explanation for createAndBuildHeap having an O(n) time complexity?

• Reply • Share ›

tony • 2 months ago

the explanation is too brief along with the demonstration ..

• Reply • Share ›

jim • 3 months ago

can we replace maxHeap->size-1 with "maxHeap->size - 2" in above code. please suggest me why cant.

• Reply • Share ›

rajesh • 4 months ago

is the above code is for decreasing order(maxheap)

• Reply • Share ›

randomguy202 ↗ rajesh • 4 months ago

The sorting is in increasing order.

I found this video explains the concept well.

https://www.youtube.com/watch?...

• Reply • Share ›

General • 4 months ago

*int left = (idx << 1) + 1; // left = 2*idx + 1*
*int right = (idx + 1) << 1; // right = 2*idx + 2*
why using shift instead of direct calculation?

• Reply • Share ›

Sri ↗ General • 4 months ago

The compiler finds shifting easier to do than multiplying, ie when looking at it from a hardware pov, the architecture is such that shifts are easier compared to arithmetic operations.

1 • Reply • Share ›

debugger • 6 months ago

Instead of doing heapification in bottom up, can we start from root and recursively do for left and right subtree ?

• Reply • Share ›

JK debugger • 5 months ago

NO. Example :

([1]), (2, 3), (4, 5, 6, 7) -->
(3), ([2], 1), (4, 5, 6, 7) -->
(3), (5, [1]), (4, 2, 6, 7) -->
(3), (5, 7), (4, 2, 6, 1)

Now
3 < 5

• Reply • Share ›

coder • 8 months ago

I don't understand why it is calling in recursion

maxHeapify(maxHeap, largest);

as "heapification must be performed in the bottom up order."

and largest index node is child node of idx node

Please explain.

• Reply • Share ›

noobinmaking coder • 5 months ago

we are starting from the bottom up order ,recursion is to check whether after swapping, lower nodes are heapified or not
NOTE: "largest" variable after first heap call points to the leaf node

• Reply • Share ›

Rahul Ramesh • 10 months ago

In maxHeapify function above:

In the construct below, is it necessary to call "maxHeapify(maxHeap, largest)" ?

```
if (largest != idx)
{
swap(&maxHeap->array[largest], &maxHeap->array[idx]);
maxHeapify(maxHeap, largest);
```

}

• Reply • Share ›

piyush jain ➚ Rahul Ramesh • 10 months ago

yes, because you dont know the element that was at idx is bigger or smaller than child nodes of arr[largest]. Its like value at arr[idx] "float down" in the max heap so that tree at 'idx' becomes max heap.

1 • Reply • Share ›

anshul • a year ago

int largest = idx; // Initialize largest as root
int left = (idx << 1) + 1; // left = 2*idx + 1
int right = (idx + 1) << 1; // right = 2*idx + 2
can sm1 plz explain me this ?

• Reply • Share ›

Kartik ➚ anshul • a year ago

The expression (idx << 1) + 1 is equivalent to idx*2 + 1.

And the expression (idx + 1) << 1 is equivalent to (idx+1)*2

Please refer http://www.geeksforgeeks.org/i... for details of bitwise operators << and >>

• Reply • Share ›

rj • a year ago

@geeksforgeeks

In the maxheapify() function If at any index stucture is something like as

2(root)
(left)5 3(right)

then first it would assign left child index into largest then it replace by right child.Then it swap root and right. And at next call it would swap root with left child.Then finally o/p is like as
5
(left)3 2(right)

But o/p mustbe like that
5
(left)2 3(right)

correct me?if i wrong?

• Reply • Share ›

Thrinadh ↗ rj • a year ago

@rj no you are wrong.
2(0)
5(1) 3(2)
In max heapify() function i value=0. The function first compares root with left child. here left child is greater than root. so largest=1.
Next it compares largest index value with right child. again largest index value is greater than right child. So largest value remains same.
Next it compares i value with largest value.here i != largest
So swapping occurs between A[i] and A[largest].
So finally it becomes 5(0)
2(1) 3(2)

^ | v • Reply • Share ›

anonymous ↗ rj • a year ago

The ads displayed here are covering up the contents partially :(

^ | v • Reply • Share ›

Guru Gorantla ↗ anonymous • 4 months ago

use adblock

^ | v • Reply • Share ›