Nathaniel Gatpandan June 10, 2023

Nomer Dela Cruz DCIT25 – Data Structures & Algorithms

Ashlie Diaz

Vien Mar Gelantaga-an

Reiniel Jhon Javier

**Heap Sort Algorithm**

Heapify asynchronous function:

async function heapify(n, i) {

    let blocksHeight = document.querySelectorAll(".block\_height");

    let root = i; // Initialize largest as root

    let left = 2 \* i + 1; // left = 2 \* i + 1

    let right = 2 \* i + 2; // right = 2 \* i + 2

    if (left < n && Number(blocksHeight[left].childNodes[0].value) > Number(blocksHeight[root].childNodes[0].value)) {

        root = left; // Check if the left child is larger than the root

    }

    if (right < n && Number(blocksHeight[right].childNodes[0].value) > Number(blocksHeight[root].childNodes[0].value)) {

        root = right; // Check if the right child is larger than the root or largest so far

    }

    if (root != i) { // If largest is not root

        let temp1 = blocksHeight[i].style.height; // Swap the heights and values of the root and largest elements

        let temp2 = blocksHeight[i].childNodes[0].value;

        blocksHeight[i].style.height = blocksHeight[root].style.height;

        blocksHeight[root].style.height = temp1;

        blocksHeight[i].childNodes[0].value =

            blocksHeight[root].childNodes[0].value;

        blocksHeight[root].childNodes[0].value = temp2;

        await new Promise((resolve) => // Wait for a short delay using Promise and setTimeout to visualize the swapping

            setTimeout(() => {

                resolve();

            }, 250)

        );

        if (!isSortingRestarted) { // Recursively Heapify the affected sub-tree

            await new Promise((resolve) => setTimeout(resolve, 250));

            await heapify(n, root);

        }

    }

}

The "heapify" function is an asynchronous function used in the heap sort algorithm. The heapify operation is performed to an array of blocks that represents the heap.

Breakdown of heapify async function:

* It is given two parameters: “**n**” (the heap's size) and “**i**” (the index of the current node being considered).
* Using document.**querySelectorAll**("**block\_height**"), it chooses every element with the "**block\_height**" class and adds them to the ‘**blocksHeight’** variable.
* It initializes “**root**” with the value of “**i**”.
* The **left** child's index is determined (**left = 2 \* i + 1**), and the **right** child's index is determined (**right = 2 \* i + 2**).
* It determines if the left child's value is greater than the root's value. If true, it updates “**root**” to the left child's index.
* It determines whether the right child's value is higher than the root or the largest value now. If true, it updates “**root**” to the appropriate child's index.
* If “**root**” is not equal to “**i**”, a child node with a higher value exists. In this instance, it switches the elements' heights and values at indexes “**i**” and “**root**”.
* To illustrate the element swapping, it adds a little delay using “**setTimeout**”.
* If the sorting process has not been restarted, it recursively calls “**heapify**” on the affected sub-tree with the new “**root**” as the starting index.

The purpose of this function is to maintain the heap property by ensuring that the parent node has a larger value than its children. By recursively calling “**heapify**", the largest element "bubbles down" to its correct position in the heap.

Heapsort asynchronous function:

async function heapSort(n) {

    let blocksHeight = document.querySelectorAll(".block\_height");

    startSortingButton.disabled = true; // Disable the "Start Sorting" button

    getArrayLengthSlider.disabled = true; // Disable the "Set Length" button

    rngArrayButton.disabled = true;

    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) { // Build heap (rearrange array)

        if (!isSortingRestarted) {

            await heapify(n, i);

        }

        if (isSortingRestarted || isSortingCompleted) {

            return;

        }

    }

    let sortedNumbers = [];

    for (let i = n - 1; i > 0; i--) { // One by one extract an element from heap

        let temp1 = blocksHeight[i].style.height; // Move current root to end

        let temp2 = blocksHeight[i].childNodes[0].value;

        blocksHeight[i].style.height = blocksHeight[0].style.height;

        blocksHeight[0].style.height = temp1;

        blocksHeight[i].childNodes[0].value = blocksHeight[0].childNodes[0].value;

        blocksHeight[0].childNodes[0].value = temp2;

        sortedNumbers.unshift(blocksHeight[i].childNodes[0].value); //display sorted numbers

        sortedNumbersContainer.innerHTML = sortedNumbers

            .map((number) => `<div class="sorted-number">&nbsp;${number}</div>`)

            .join("");

        await new Promise((resolve) => setTimeout(resolve, 250));

        if (!isSortingRestarted) { // Call max Heapify on the reduced heap

            await heapify(i, 0);

        }

        if (isSortingRestarted || isSortingCompleted) {

            return;

        }

    }

    sortedNumbers.unshift(blocksHeight[0].childNodes[0].value); //initialize sorted numbers

    sortedNumbersContainer.innerHTML = sortedNumbers

        .map((number) => `<div class="sorted-number">&nbsp;${number}</div>`)

        .join("");

    isSortingCompleted = true;

    startSortingButton.disabled = false; // Enable the button after sorting completes

    getArrayLengthSlider.disabled = false; // Enable the "Set Length" button

    rngArrayButton.disabled = false; // Enable the rng button

    restartSortingButton.disabled = true; // Disable the "restart Sorting" button

    const appendAlert = (message) => { // Create Bootstrap alert when sorting is completed

        const wrapper = document.createElement("div");

        wrapper.innerHTML = [

            `<div class="alert alert-success alert-dismissible fade show" role="alert">`,

            `   <h3>${message}</h3>`,

            '   <button type="button" class="btn-close" data-bs-dismiss="alert"

aria-label="Close"></button>',

            "</div>",

        ].join("");

        alertPlaceholder.append(wrapper);

        setTimeout(() => { // Close the alert after 5 seconds

            wrapper.remove();

        }, 5000);

    };

    appendAlert("Sorting Completed!");

    isSortingCompleted = false; //enable to start heapsort again

}

The "heapSort" function implements the Heap Sort algorithm to sort an array of blocks.

Breakdown of heapsort async function:

* Using document.querySelectorAll("block\_height"), it chooses all elements of the class and adds them to the **blocksHeight** variable.
* To avoid distractions during sorting, it turns off the "**StartSorting**," "**SetLength**," and "**rng**" buttons.
* It calls the **heapify** method on each parent node to create the initial max heap.
* It examines if the sorting procedure has been started or finished inside the loop. If so, it terminates further execution and returns.
* The process then moves on to extracting the biggest element from the heap and inserting it at the end of the array after creating the initial max heap.
* The extracted element is added to the **sortedNumbers** array and displayed in the **sortedNumbersContainer** element, which added below of the heapsort process.
* In order to see the sorting procedure, it uses await new Promise((resolve) => setTimeout(resolve, 250)).
* If the sorting process has been restarted or finished, it is determined within the loop. In that case, it goes back and halts further execution.
* The final element is added to the **sortedNumbers** array when all elements have been extracted.
* The **appendAlert** function is used to display a completion message and enable/disable the corresponding buttons.
* A Bootstrap alert is dynamically created and added to the **alertPlaceholder** element using the **appendAlert** function.
* The completion message is shown for five seconds before it closes on its own.
* In order to restart the heap sort, it sets **isSortingCompleted** to false at the end.