

Find the height of the binary tree using :

int getHeight(struct TreeNode\* root) {

if (root == NULL) {

// Empty tree, height is -1

return -1;

} else {

// Recursive case: Height is 1 + maximum of the heights of left and right subtrees

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

// Return the maximum height

return (leftHeight > rightHeight) ? (leftHeight + 1) : (rightHeight + 1);

}

}

#include <stdio.h>

#include <stdlib.h>

// Structure to represent the max heap

typedef struct {

int\* arr; // Array to store heap elements

int capacity; // Maximum capacity of the heap

int size; // Current number of elements in the heap

} MaxHeap;

// Function to create a new max heap

MaxHeap\* createMaxHeap(int capacity) {

MaxHeap\* maxHeap = (MaxHeap\*)malloc(sizeof(MaxHeap));

maxHeap->capacity = capacity;

maxHeap->size = 0;

maxHeap->arr = (int\*)malloc(capacity \* sizeof(int));

return maxHeap;

}

// Helper function to swap two elements in the heap

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at the given index

void maxHeapify(MaxHeap\* maxHeap, int index) {

int largest = index;

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

// Find the largest among the current node, left child, and right child

if (leftChild < maxHeap->size && maxHeap->arr[leftChild] > maxHeap->arr[largest])

largest = leftChild;

if (rightChild < maxHeap->size && maxHeap->arr[rightChild] > maxHeap->arr[largest])

largest = rightChild;

// If the largest is not the current node, swap them and recursively heapify

if (largest != index) {

swap(&maxHeap->arr[index], &maxHeap->arr[largest]);

maxHeapify(maxHeap, largest);

}

}

// Function to insert a new element into the max heap

void insert(MaxHeap\* maxHeap, int value) {

if (maxHeap->size == maxHeap->capacity) {

printf("Heap is full. Cannot insert.\n");

return;

}

// Insert the new element at the end of the heap

int index = maxHeap->size;

maxHeap->arr[index] = value;

maxHeap->size++;

// Heapify the tree from bottom to top to maintain the max heap property

while (index > 0 && maxHeap->arr[index] > maxHeap->arr[(index - 1) / 2]) {

swap(&maxHeap->arr[index], &maxHeap->arr[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

// Function to extract the maximum element from the max heap

int extractMax(MaxHeap\* maxHeap) {

if (maxHeap->size == 0) {

printf("Heap is empty. Cannot extract maximum.\n");

return -1;

}

// The maximum element is at the root

int maxElement = maxHeap->arr[0];

// Replace the root with the last element in the heap

maxHeap->arr[0] = maxHeap->arr[maxHeap->size - 1];

maxHeap->size--;

// Heapify the tree from the root to maintain the max heap property

maxHeapify(maxHeap, 0);

return maxElement;

}

// Function to print the elements of the max heap

void printHeap(MaxHeap\* maxHeap) {

printf("Max Heap: ");

for (int i = 0; i < maxHeap->size; i++) {

printf("%d ", maxHeap->arr[i]);

}

printf("\n");

}

int main() {

// Create a max heap with a capacity of 10

MaxHeap\* maxHeap = createMaxHeap(10);

// Insert elements into the heap

insert(maxHeap, 4);

insert(maxHeap, 2);

insert(maxHeap, 8);

insert(maxHeap, 1);

insert(maxHeap, 9);

// Print the heap

printHeap(maxHeap); // Output: Max Heap: 9 8 4 2 1

// Extract the maximum element

int max = extractMax(maxHeap);

printf("Extracted maximum element: %d\n", max); // Output: Extracted maximum element: 9

// Print the heap after extraction

printHeap(maxHeap); // Output: Max Heap: 8 2 4 1

// Clean up and free memory

free(maxHeap->arr);

free(maxHeap);

return 0;

}

Max Heap with delete Operation

#include <stdio.h>

#include <stdlib.h>

// Structure to represent the max heap

typedef struct {

int\* arr; // Array to store heap elements

int capacity; // Maximum capacity of the heap

int size; // Current number of elements in the heap

} MaxHeap;

// Function to create a new max heap

MaxHeap\* createMaxHeap(int capacity) {

MaxHeap\* maxHeap = (MaxHeap\*)malloc(sizeof(MaxHeap));

maxHeap->capacity = capacity;

maxHeap->size = 0;

maxHeap->arr = (int\*)malloc(capacity \* sizeof(int));

return maxHeap;

}

// Helper function to swap two elements in the heap

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at the given index

void maxHeapify(MaxHeap\* maxHeap, int index) {

int largest = index;

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

// Find the largest among the current node, left child, and right child

if (leftChild < maxHeap->size && maxHeap->arr[leftChild] > maxHeap->arr[largest])

largest = leftChild;

if (rightChild < maxHeap->size && maxHeap->arr[rightChild] > maxHeap->arr[largest])

largest = rightChild;

// If the largest is not the current node, swap them and recursively heapify

if (largest != index) {

swap(&maxHeap->arr[index], &maxHeap->arr[largest]);

maxHeapify(maxHeap, largest);

}

}

// Function to insert a new element into the max heap

void insert(MaxHeap\* maxHeap, int value) {

if (maxHeap->size == maxHeap->capacity) {

printf("Heap is full. Cannot insert.\n");

return;

}

// Insert the new element at the end of the heap

int index = maxHeap->size;

maxHeap->arr[index] = value;

maxHeap->size++;

// Heapify the tree from bottom to top to maintain the max heap property

while (index > 0 && maxHeap->arr[index] > maxHeap->arr[(index - 1) / 2]) {

swap(&maxHeap->arr[index], &maxHeap->arr[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

// Function to extract the maximum element from the max heap

int extractMax(MaxHeap\* maxHeap) {

if (maxHeap->size == 0) {

printf("Heap is empty. Cannot extract maximum.\n");

return -1;

}

// The maximum element is at the root

int maxElement = maxHeap->arr[0];

// Replace the root with the last element in the heap

maxHeap->arr[0] = maxHeap->arr[maxHeap->size - 1];

maxHeap->size--;

// Heapify the tree from the root to maintain the max heap property

maxHeapify(maxHeap, 0);

return maxElement;

}

// Function to delete a specified element from the max heap

void deleteMax(MaxHeap\* maxHeap, int key) {

// Find the index of the element to be deleted

int index = -1;

for (int i = 0; i < maxHeap->size; i++) {

if (maxHeap->arr[i] == key) {

index = i;

break;

}

}

if (index == -1) {

printf("Element not found in the heap.\n");

return;

}

// Replace the element to be deleted with the last element in the heap

maxHeap->arr[index] = maxHeap->arr[maxHeap->size - 1];

maxHeap->size--;

// Heapify the tree from the modified element to maintain the max heap property

maxHeapify(maxHeap, index);

}

// Function to print the elements of the max heap

void printHeap(MaxHeap\* maxHeap) {

printf("Max Heap: ");

for (int i = 0; i < maxHeap->size; i++) {

printf("%d ", maxHeap->arr[i]);

}

printf("\n");

}

int main() {

// Create a max heap with a capacity of 10

MaxHeap\* maxHeap = createMaxHeap(10);

// Insert elements into the heap

insert(maxHeap, 4);

insert(maxHeap, 2);

insert(maxHeap, 8);

insert(maxHeap, 1);

insert(maxHeap, 9);

// Print the heap

printHeap(maxHeap); // Output: Max Heap: 9 8 4 2 1

// Delete an element from the heap (e.g., delete 8)

deleteMax(maxHeap, 8);

// Print the heap after deletion

printHeap(maxHeap); // Output: Max Heap: 9 2 4 1

// Clean up and free memory

free(maxHeap->arr);

free(maxHeap);

return 0;

}

Deletion operation on min heap :

#include <stdio.h>

#include <stdlib.h>

// Structure to represent the min heap

typedef struct {

int\* arr; // Array to store heap elements

int capacity; // Maximum capacity of the heap

int size; // Current number of elements in the heap

} MinHeap;

// Function to create a new min heap

MinHeap\* createMinHeap(int capacity) {

MinHeap\* minHeap = (MinHeap\*)malloc(sizeof(MinHeap));

minHeap->capacity = capacity;

minHeap->size = 0;

minHeap->arr = (int\*)malloc(capacity \* sizeof(int));

return minHeap;

}

// Helper function to swap two elements in the heap

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at the given index

void minHeapify(MinHeap\* minHeap, int index) {

int smallest = index;

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

// Find the smallest among the current node, left child, and right child

if (leftChild < minHeap->size && minHeap->arr[leftChild] < minHeap->arr[smallest])

smallest = leftChild;

if (rightChild < minHeap->size && minHeap->arr[rightChild] < minHeap->arr[smallest])

smallest = rightChild;

// If the smallest is not the current node, swap them and recursively heapify

if (smallest != index) {

swap(&minHeap->arr[index], &minHeap->arr[smallest]);

minHeapify(minHeap, smallest);

}

}

// Function to insert a new element into the min heap

void insert(MinHeap\* minHeap, int value) {

if (minHeap->size == minHeap->capacity) {

printf("Heap is full. Cannot insert.\n");

return;

}

// Insert the new element at the end of the heap

int index = minHeap->size;

minHeap->arr[index] = value;

minHeap->size++;

// Heapify the tree from bottom to top to maintain the min heap property

while (index > 0 && minHeap->arr[index] < minHeap->arr[(index - 1) / 2]) {

swap(&minHeap->arr[index], &minHeap->arr[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

// Function to extract the minimum element from the min heap

int extractMin(MinHeap\* minHeap) {

if (minHeap->size == 0) {

printf("Heap is empty. Cannot extract minimum.\n");

return -1;

}

// The minimum element is at the root

int minElement = minHeap->arr[0];

// Replace the root with the last element in the heap

minHeap->arr[0] = minHeap->arr[minHeap->size - 1];

minHeap->size--;

// Heapify the tree from the root to maintain the min heap property

minHeapify(minHeap, 0);

return minElement;

}

// Function to delete a specified element from the min heap

void deleteMin(MinHeap\* minHeap, int key) {

// Find the index of the element to be deleted

int index = -1;

for (int i = 0; i < minHeap->size; i++) {

if (minHeap->arr[i] == key) {

index = i;

break;

}

}

if (index == -1) {

printf("Element not found in the heap.\n");

return;

}

// Replace the element to be deleted with the last element in the heap

minHeap->arr[index] = minHeap->arr[minHeap->size - 1];

minHeap->size--;

// Heapify the tree from the modified element to maintain the min heap property

minHeapify(minHeap, index);

}

// Function to print the elements of the min heap

void printHeap(MinHeap\* minHeap) {

printf("Min Heap: ");

for (int i = 0; i < minHeap->size; i++) {

printf("%d ", minHeap->arr[i]);

}

printf("\n");

}

int main() {

// Create a min heap with a capacity of 10

MinHeap\* minHeap = createMinHeap(10);

// Insert elements into the heap

insert(minHeap, 4);

insert(minHeap, 2);

insert(minHeap, 8);

insert(minHeap, 1);

insert(minHeap, 9);

// Print the heap

printHeap(minHeap); // Output: Min Heap: 1 2 8 4 9

// Delete an element from the heap (e.g., delete 8)

deleteMin(minHeap, 8);

// Print the heap after deletion

printHeap(minHeap); // Output: Min Heap: 1 2 9 4

// Clean up and free memory

free(minHeap->arr);

free(minHeap);

return 0;

}