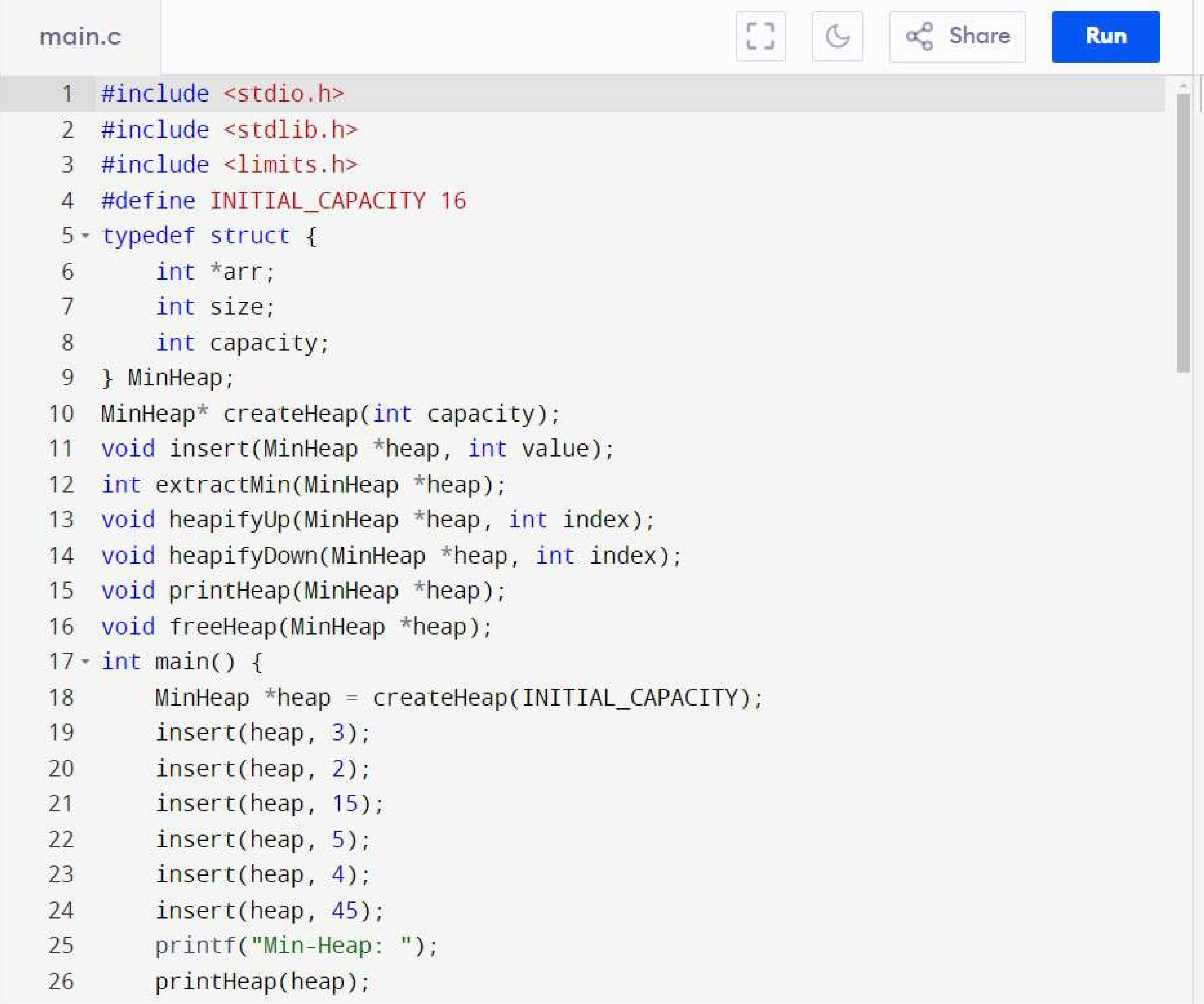
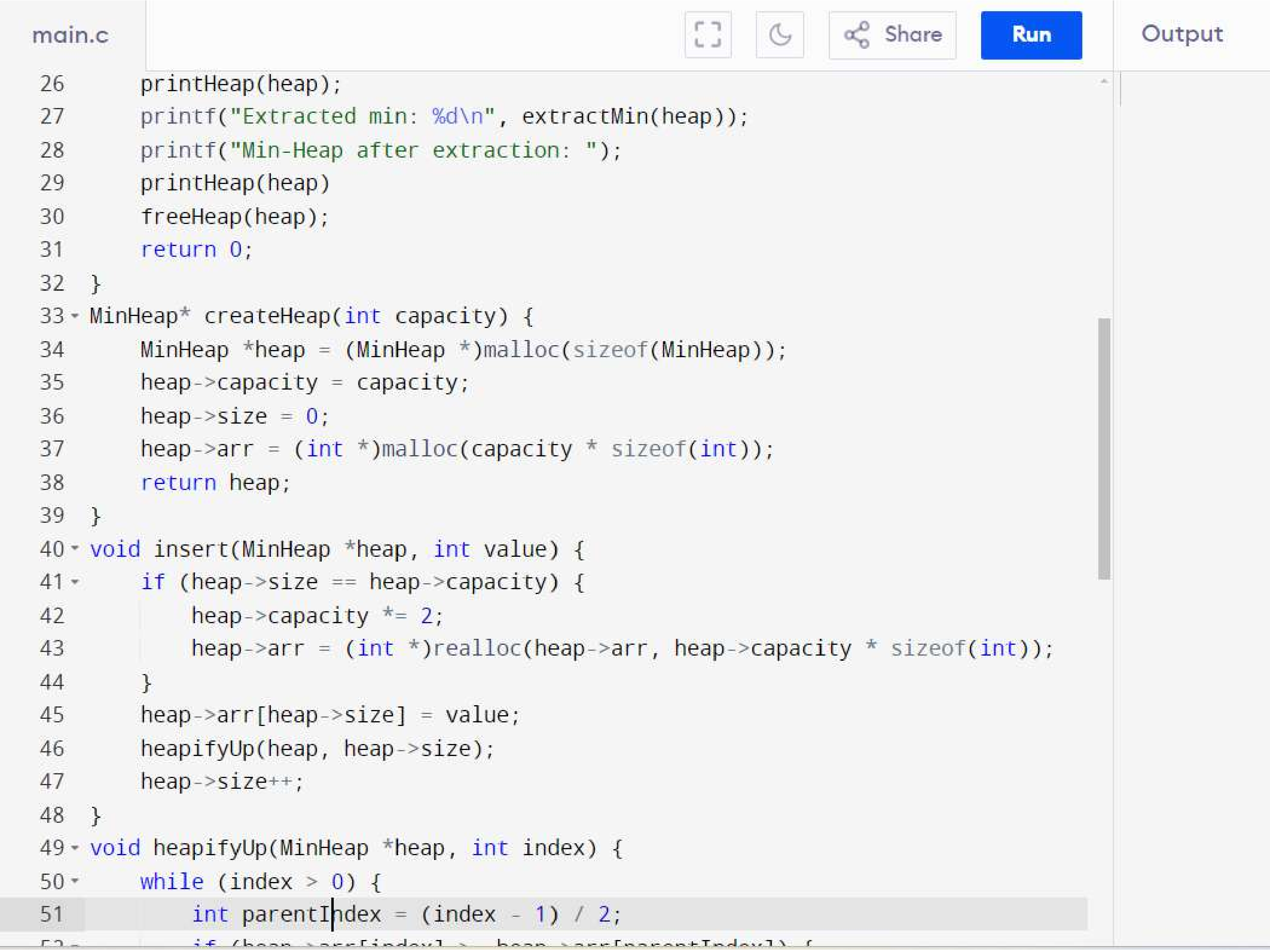
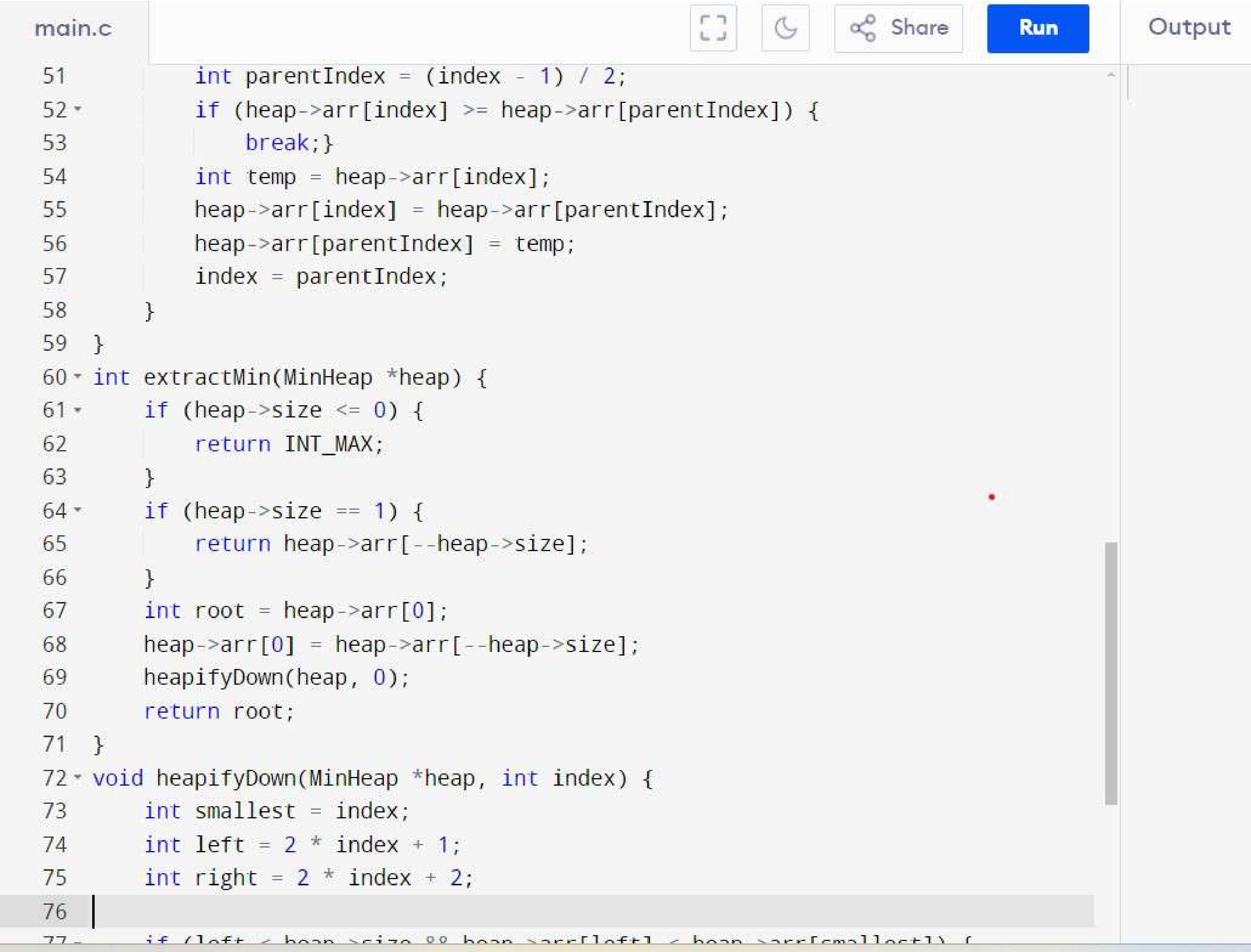


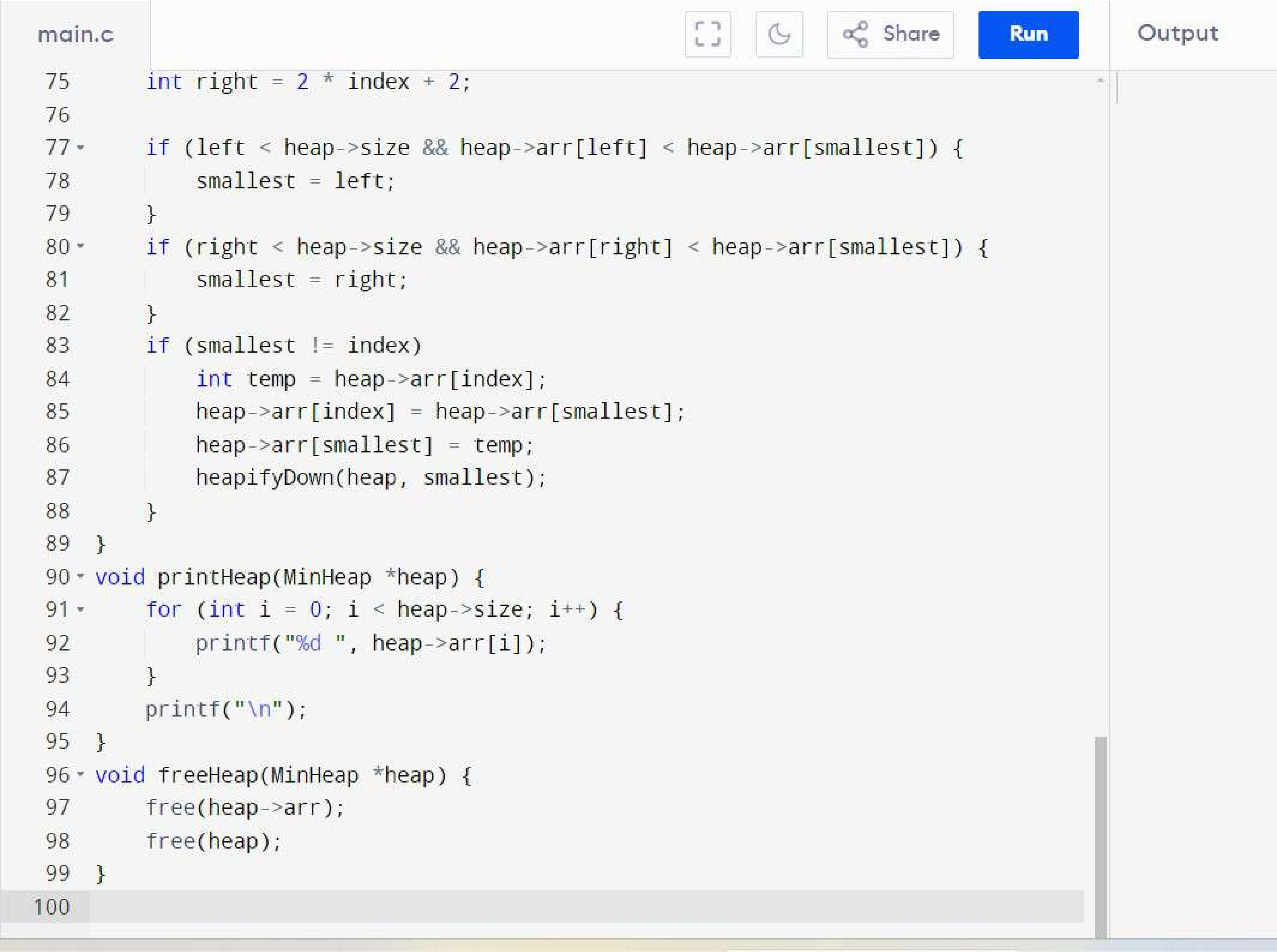
```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define INITIAL_CAPACITY 16
typedef struct {
    int *arr;
    int size;
    int capacity;
} MinHeap;
MinHeap* createHeap(int capacity);
void insert(MinHeap *heap, int value);
int extractMin(MinHeap *heap);
void heapifyUp(MinHeap *heap, int index);
void heapifyDown(MinHeap *heap, int index);
void printHeap(MinHeap *heap);
void freeHeap(MinHeap *heap);
int main() {
    MinHeap *heap = createHeap(INITIAL_CAPACITY);
    insert(heap, 3);
    insert(heap, 2);
    insert(heap, 15);
    insert(heap, 5);
    insert(heap, 4);
    insert(heap, 45);
    printf("Min-Heap: ");
    printHeap(heap);
```



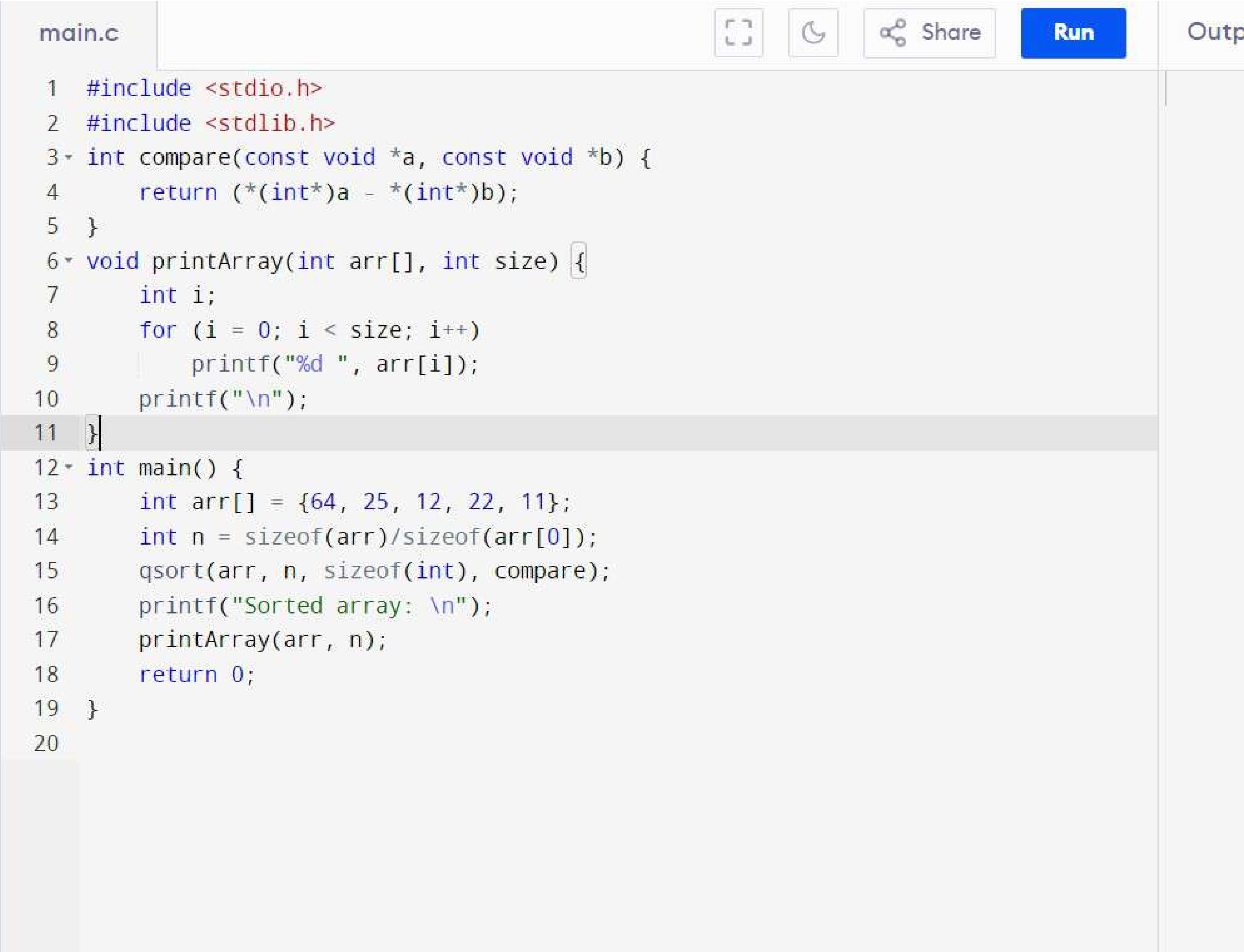
```c
    printHeap(heap);
    printf("Extracted min: %d\n", extractMin(heap));
    printf("Min-Heap after extraction: ");
    printHeap(heap)
    freeHeap(heap);
    return 0;
}
MinHeap* createHeap(int capacity) {
    MinHeap *heap = (MinHeap *)malloc(sizeof(MinHeap));
    heap->capacity = capacity;
    heap->size = 0;
    heap->arr = (int *)malloc(capacity * sizeof(int));
    return heap;
}
void insert(MinHeap *heap, int value) {
    if (heap->size == heap->capacity) {
        heap->capacity *= 2;
        heap->arr = (int *)realloc(heap->arr, heap->capacity * sizeof(int));
    }
    heap->arr[heap->size] = value;
    heapifyUp(heap, heap->size);
    heap->size++;
}
void heapifyUp(MinHeap *heap, int index) {
    while (index > 0) {
        int parentIndex = (index - 1) / 2;
```



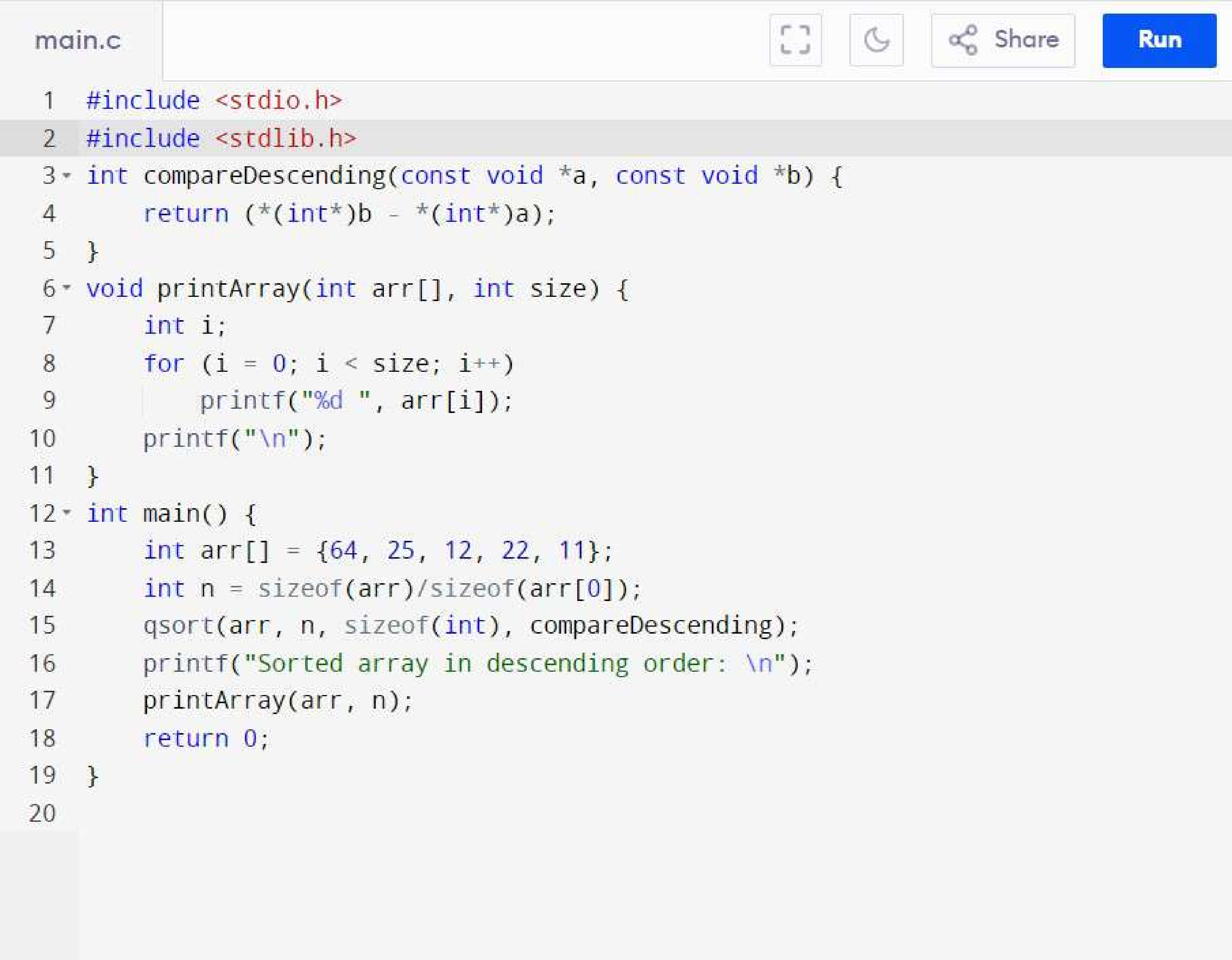
```c
        int parentIndex = (index - 1) / 2;
        if (heap->arr[index] >= heap->arr[parentIndex]) {
            break;}
        int temp = heap->arr[index];
        heap->arr[index] = heap->arr[parentIndex];
        heap->arr[parentIndex] = temp;
        index = parentIndex;
    }
}
int extractMin(MinHeap *heap) {
    if (heap->size <= 0) {
        return INT_MAX;
    }
    if (heap->size == 1) {
        return heap->arr[--heap->size];
    }
    int root = heap->arr[0];
    heap->arr[0] = heap->arr[--heap->size];
    heapifyDown(heap, 0);
    return root;
}
void heapifyDown(MinHeap *heap, int index) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {
```



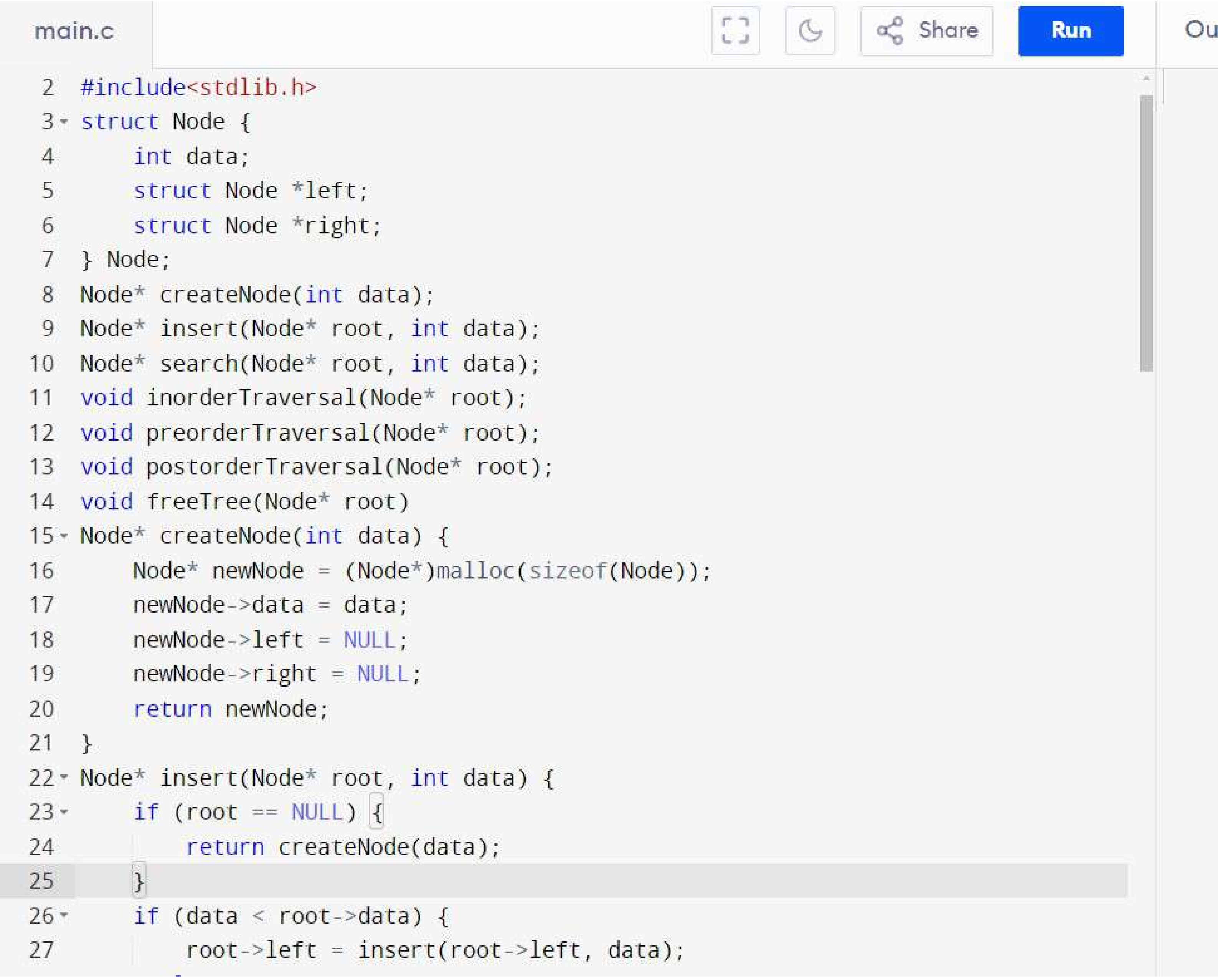
```c
    int right = 2 * index + 2;

    if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {
        smallest = left;
    }
    if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {
        smallest = right;
    }
    if (smallest != index)
        int temp = heap->arr[index];
        heap->arr[index] = heap->arr[smallest];
        heap->arr[smallest] = temp;
        heapifyDown(heap, smallest);
    }
}
void printHeap(MinHeap *heap) {
    for (int i = 0; i < heap->size; i++) {
        printf("%d ", heap->arr[i]);
    }
    printf("\n");
}
void freeHeap(MinHeap *heap) {
    free(heap->arr);
    free(heap);
}

```



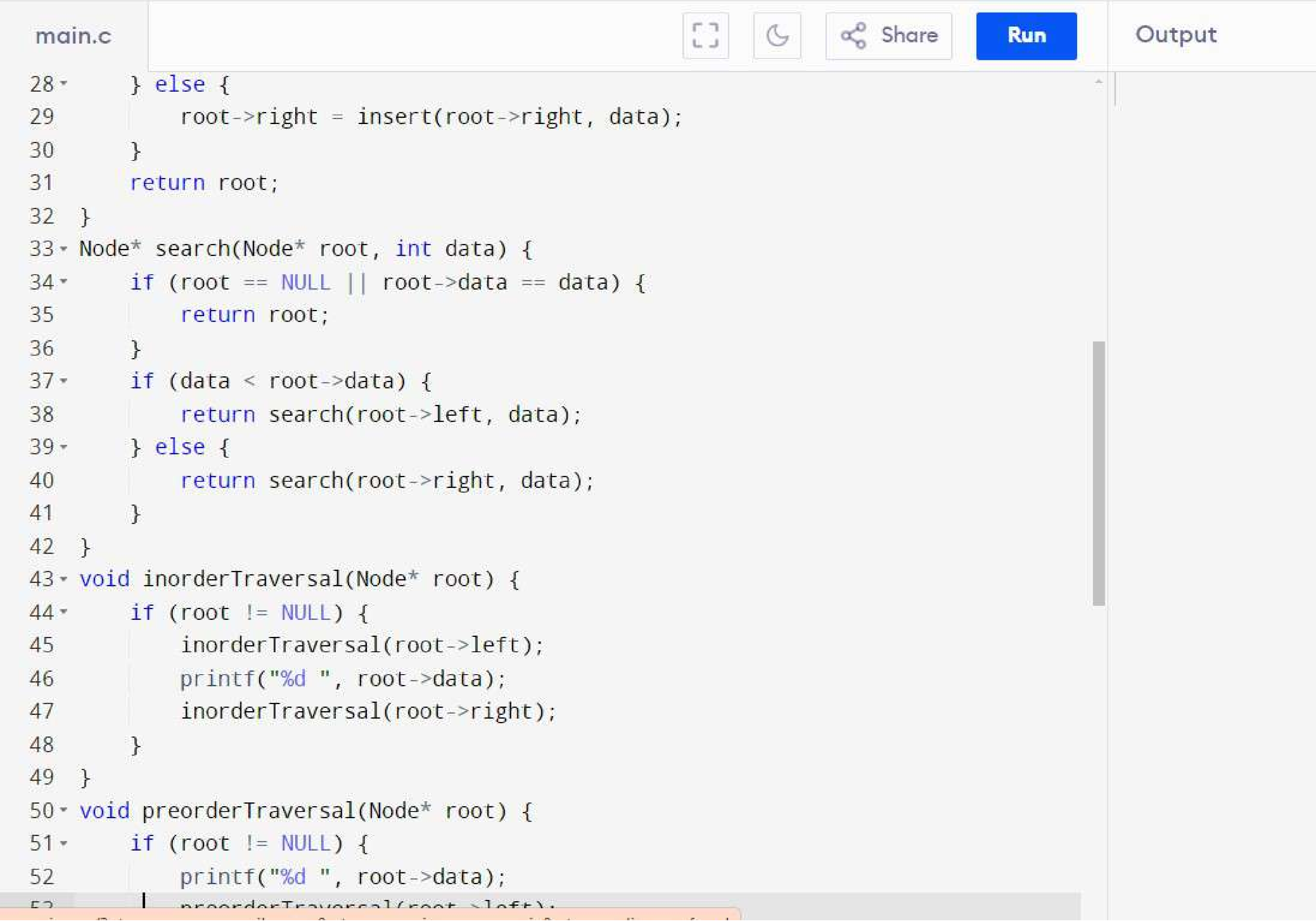
```c
#include <stdio.h>
#include <stdlib.h>
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    qsort(arr, n, sizeof(int), compare);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```



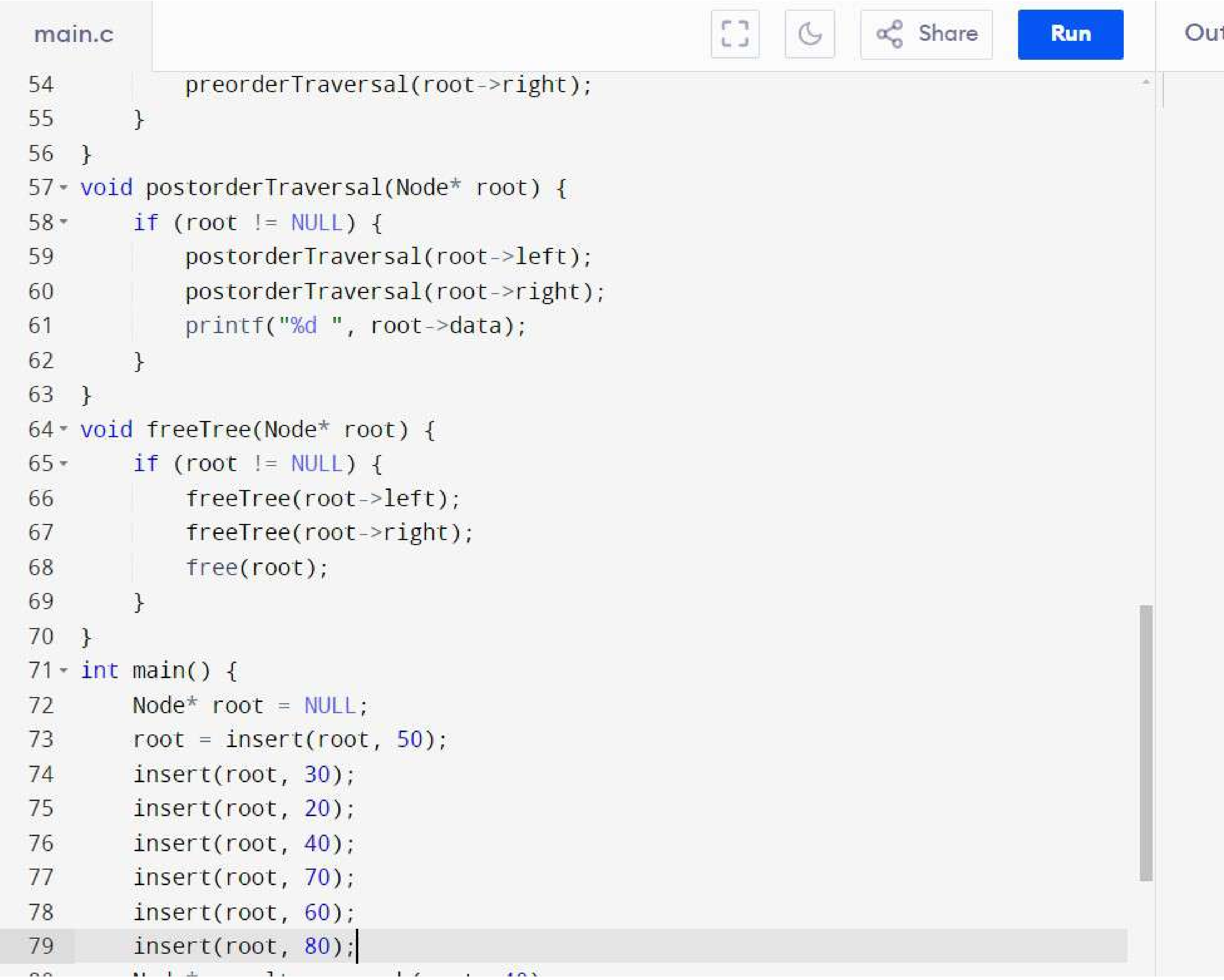
```c
#include <stdio.h>
#include <stdlib.h>
int compareDescending(const void *a, const void *b) {
    return (*(int*)b - *(int*)a);
}
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    qsort(arr, n, sizeof(int), compareDescending);
    printf("Sorted array in descending order: \n");
    printArray(arr, n);
    return 0;
}
```



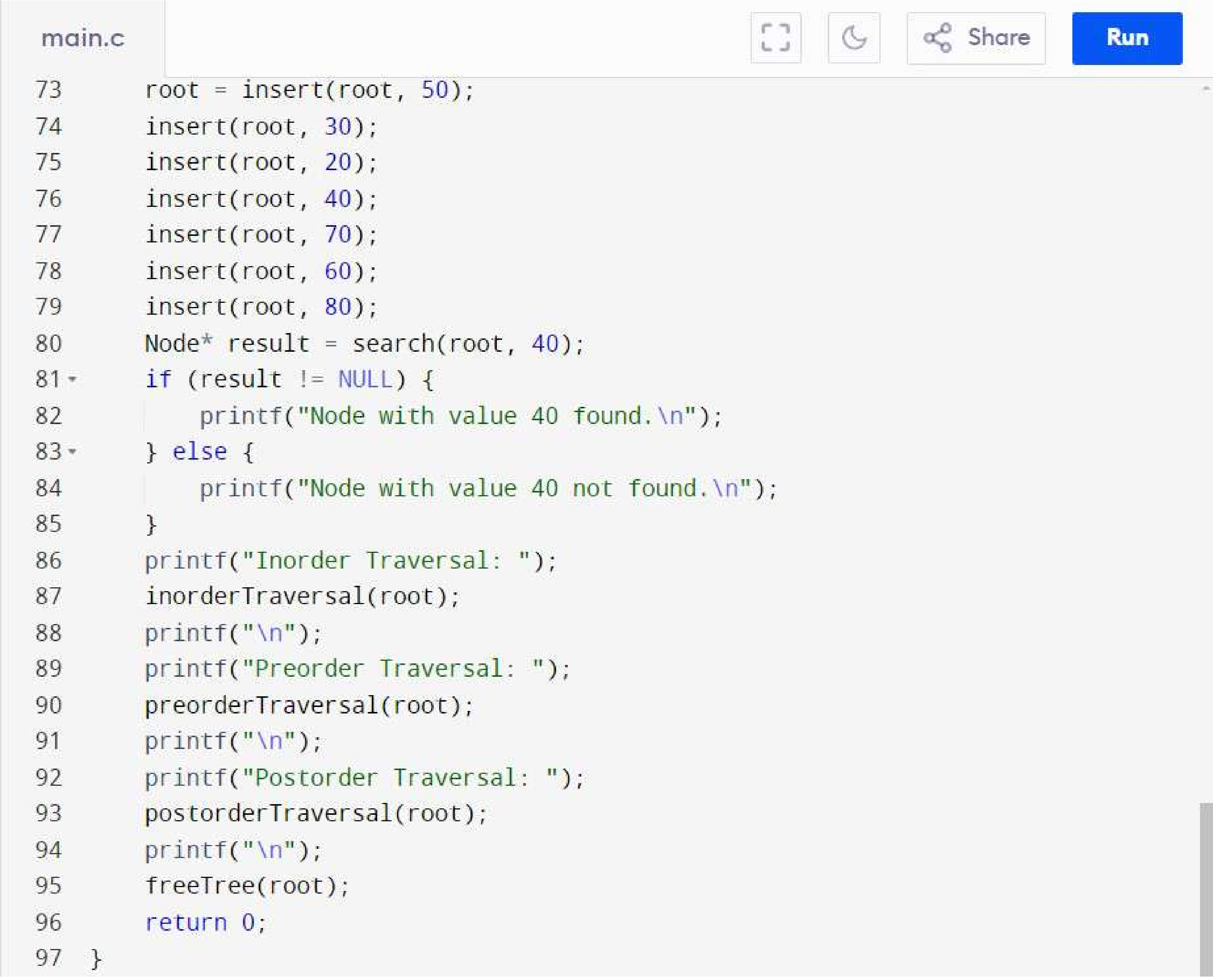
```c
#include<stdlib.h>
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
} Node;
Node* createNode(int data);
Node* insert(Node* root, int data);
Node* search(Node* root, int data);
void inorderTraversal(Node* root);
void preorderTraversal(Node* root);
void postorderTraversal(Node* root);
void freeTree(Node* root)
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
Node* insert(Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
```



```c
    } else {
        root->right = insert(root->right, data);
    }
    return root;
}
Node* search(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
    }
}
void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
void preorderTraversal(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
```



```c
        preorderTraversal(root->right);
    }
}
void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
int main() {
    Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```



```c
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    Node* result = search(root, 40);
    if (result != NULL) {
        printf("Node with value 40 found.\n");
    } else {
        printf("Node with value 40 not found.\n");
    }
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");
    freeTree(root);
    return 0;
}
```