

main.c



Share

Run

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #define INITIAL_CAPACITY 16
5 typedef struct {
6     int *arr;
7     int size;
8     int capacity;
9 } MinHeap;
10 MinHeap* createHeap(int capacity);
11 void insert(MinHeap *heap, int value);
12 int extractMin(MinHeap *heap);
13 void heapifyUp(MinHeap *heap, int index);
14 void heapifyDown(MinHeap *heap, int index);
15 void printHeap(MinHeap *heap);
16 void freeHeap(MinHeap *heap);
17 int main() {
18     MinHeap *heap = createHeap(INITIAL_CAPACITY);
19     insert(heap, 3);
20     insert(heap, 2);
21     insert(heap, 15);
22     insert(heap, 5);
23     insert(heap, 4);
24     insert(heap, 45);
25     printf("Min-Heap: ");
26     printHeap(heap);
```

main.c



Share

Run

Output

```
26 printHeap(heap);
27 printf("Extracted min: %d\n", extractMin(heap));
28 printf("Min-Heap after extraction: ");
29 printHeap(heap);
30 freeHeap(heap);
31 return 0;
32 }
33 MinHeap* createHeap(int capacity) {
34     MinHeap *heap = (MinHeap *)malloc(sizeof(MinHeap));
35     heap->capacity = capacity;
36     heap->size = 0;
37     heap->arr = (int *)malloc(capacity * sizeof(int));
38     return heap;
39 }
40 void insert(MinHeap *heap, int value) {
41     if (heap->size == heap->capacity) {
42         heap->capacity *= 2;
43         heap->arr = (int *)realloc(heap->arr, heap->capacity * sizeof(int));
44     }
45     heap->arr[heap->size] = value;
46     heapifyUp(heap, heap->size);
47     heap->size++;
48 }
49 void heapifyUp(MinHeap *heap, int index) {
50     while (index > 0) {
51         int parentIndex = (index - 1) / 2;
52         if (heap->arr[index] > heap->arr[parentIndex]) {
```

main.c



Share

Run

Output

```
51     int parentIndex = (index - 1) / 2;
52     if (heap->arr[index] >= heap->arr[parentIndex]) {
53         break;}
54     int temp = heap->arr[index];
55     heap->arr[index] = heap->arr[parentIndex];
56     heap->arr[parentIndex] = temp;
57     index = parentIndex;
58 }
59 }
60 int extractMin(MinHeap *heap) {
61     if (heap->size <= 0) {
62         return INT_MAX;
63     }
64     if (heap->size == 1) {
65         return heap->arr[--heap->size];
66     }
67     int root = heap->arr[0];
68     heap->arr[0] = heap->arr[--heap->size];
69     heapifyDown(heap, 0);
70     return root;
71 }
72 void heapifyDown(MinHeap *heap, int index) {
73     int smallest = index;
74     int left = 2 * index + 1;
75     int right = 2 * index + 2;
76     if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {
77         smallest = left;
78     }
79     if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {
80         smallest = right;
81     }
82     if (smallest != index) {
83         int temp = heap->arr[index];
84         heap->arr[index] = heap->arr[smallest];
85         heap->arr[smallest] = temp;
86         heapifyDown(heap, smallest);
87     }
88 }
```

main.c



Share

Run

Output

```
75     int right = 2 * index + 2;
76
77     if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {
78         smallest = left;
79     }
80     if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {
81         smallest = right;
82     }
83     if (smallest != index)
84     {
85         int temp = heap->arr[index];
86         heap->arr[index] = heap->arr[smallest];
87         heap->arr[smallest] = temp;
88         heapifyDown(heap, smallest);
89     }
90 void printHeap(MinHeap *heap) {
91     for (int i = 0; i < heap->size; i++) {
92         printf("%d ", heap->arr[i]);
93     }
94     printf("\n");
95 }
96 void freeHeap(MinHeap *heap) {
97     free(heap->arr);
98     free(heap);
99 }
```

100

main.c



Share

Run

Output

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int compare(const void *a, const void *b) {
4      return (*(int*)a - *(int*)b);
5  }
6  void printArray(int arr[], int size) {
7      int i;
8      for (i = 0; i < size; i++)
9          printf("%d ", arr[i]);
10     printf("\n");
11 }
12 int main() {
13     int arr[] = {64, 25, 12, 22, 11};
14     int n = sizeof(arr)/sizeof(arr[0]);
15     qsort(arr, n, sizeof(int), compare);
16     printf("Sorted array: \n");
17     printArray(arr, n);
18     return 0;
19 }
20
```


main.c



Share

Run

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int compareDescending(const void *a, const void *b) {
4     return (*(int*)b - *(int*)a);
5 }
6 void printArray(int arr[], int size) {
7     int i;
8     for (i = 0; i < size; i++)
9         printf("%d ", arr[i]);
10    printf("\n");
11 }
12 int main() {
13     int arr[] = {64, 25, 12, 22, 11};
14     int n = sizeof(arr)/sizeof(arr[0]);
15     qsort(arr, n, sizeof(int), compareDescending);
16     printf("Sorted array in descending order: \n");
17     printArray(arr, n);
18     return 0;
19 }
20
```

main.c



Share

Run

Out

```
2  #include<stdlib.h>
3  struct Node {
4      int data;
5      struct Node *left;
6      struct Node *right;
7  } Node;
8  Node* createNode(int data);
9  Node* insert(Node* root, int data);
10 Node* search(Node* root, int data);
11 void inorderTraversal(Node* root);
12 void preorderTraversal(Node* root);
13 void postorderTraversal(Node* root);
14 void freeTree(Node* root)
15 Node* createNode(int data) {
16     Node* newNode = (Node*)malloc(sizeof(Node));
17     newNode->data = data;
18     newNode->left = NULL;
19     newNode->right = NULL;
20     return newNode;
21 }
22 Node* insert(Node* root, int data) {
23     if (root == NULL) {
24         return createNode(data);
25     }
26     if (data < root->data) {
27         root->left = insert(root->left, data);
```

main.c



Share

Run

Output

```
28* } else {
29     root->right = insert(root->right, data);
30 }
31 return root;
32 }
33 Node* search(Node* root, int data) {
34*     if (root == NULL || root->data == data) {
35         return root;
36     }
37*     if (data < root->data) {
38         return search(root->left, data);
39*     } else {
40         return search(root->right, data);
41     }
42 }
43* void inorderTraversal(Node* root) {
44*     if (root != NULL) {
45         inorderTraversal(root->left);
46         printf("%d ", root->data);
47         inorderTraversal(root->right);
48     }
49 }
50* void preorderTraversal(Node* root) {
51*     if (root != NULL) {
52         printf("%d ", root->data);
53         preorderTraversal(root->left);
54         preorderTraversal(root->right);
55     }
56 }
```


main.c



Share

Run

Out

```
54     preorderTraversal(root->right);
55 }
56 }
57 void postorderTraversal(Node* root) {
58     if (root != NULL) {
59         postorderTraversal(root->left);
60         postorderTraversal(root->right);
61         printf("%d ", root->data);
62     }
63 }
64 void freeTree(Node* root) {
65     if (root != NULL) {
66         freeTree(root->left);
67         freeTree(root->right);
68         free(root);
69     }
70 }
71 int main() {
72     Node* root = NULL;
73     root = insert(root, 50);
74     insert(root, 30);
75     insert(root, 20);
76     insert(root, 40);
77     insert(root, 70);
78     insert(root, 60);
79     insert(root, 80);
```

main.c



Share

Run

```
73     root = insert(root, 50);
74     insert(root, 30);
75     insert(root, 20);
76     insert(root, 40);
77     insert(root, 70);
78     insert(root, 60);
79     insert(root, 80);
80     Node* result = search(root, 40);
81     if (result != NULL) {
82         printf("Node with value 40 found.\n");
83     } else {
84         printf("Node with value 40 not found.\n");
85     }
86     printf("Inorder Traversal: ");
87     inorderTraversal(root);
88     printf("\n");
89     printf("Preorder Traversal: ");
90     preorderTraversal(root);
91     printf("\n");
92     printf("Postorder Traversal: ");
93     postorderTraversal(root);
94     printf("\n");
95     freeTree(root);
96     return 0;
97 }
```