

1. AVL Tree Insertion

main.c	Output
<pre>1 #include <stdio.h> 2 #include <stdlib.h> 3 4 struct Node { 5 int data; 6 struct Node *left, *right; 7 int height; 8 }; 9 10 // Function to get the height of a node 11 int height(struct Node* node) { 12 if (node == NULL) return 0; 13 return node->height; 14 } 15 16 // Function to get the balance factor of a node 17 int balanceFactor(struct Node* node) { 18 if (node == NULL) return 0; 19 return height(node->left) - height(node->right); 20 } 21 22 // Right rotation (used for left-left case) 23 struct Node* rightRotate(struct Node* y) { 24 struct Node* x = y->left; 25 struct Node* T2 = x->right; 26 27 x->right = y; 28 y->left = T2; 29 30 y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right)); 31 x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right)); 32 33 return x; 34 } 35 36 // Left rotation (used for right-right case) 37 struct Node* leftRotate(struct Node* x) { 38 struct Node* y = x->right; 39 struct Node* T2 = y->left; 40 41 y->left = x; 42 x->right = T2; 43 44 x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));</pre>	<pre>/tmp/WOSM7Zz8OH.o In-order Traversal: 10 20 30 === Code Execution Successful ===</pre>
<pre>45 y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right)); 46 47 return y; 48 } 49 50 // Left-Right rotation (used for left-right case) 51 struct Node* leftRightRotate(struct Node* node) { 52 node->left = leftRotate(node->left); 53 return rightRotate(node); 54 } 55 56 // Right-Left rotation (used for right-left case) 57 struct Node* rightLeftRotate(struct Node* node) { 58 node->right = rightRotate(node->right); 59 return leftRotate(node); 60 } 61 62 // Insert a node and balance the tree 63 struct Node* insert(struct Node* node, int data) { 64 if (node == NULL) { 65 struct Node* newNode = malloc(sizeof(struct Node)); 66 newNode->data = data; 67 newNode->left = newNode->right = NULL; 68 newNode->height = 1; 69 return newNode; 70 } 71 72 if (data < node->data) { 73 node->left = insert(node->left, data); 74 } else if (data > node->data) { 75 node->right = insert(node->right, data); 76 } else { 77 return node; 78 } 79 80 node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height (node->right)); 81 82 int balance = balanceFactor(node); 83 84 if (balance > 1 && data < node->left->data) { 85 return rightRotate(node); 86 } 87 if (balance < -1 && data > node->right->data) {</pre>	<pre>/tmp/WOSM7Zz8OH.o In-order Traversal: 10 20 30 === Code Execution Successful ===</pre>

```
main.c
82 int balance = balanceFactor(node);
83
84 if (balance > 1 && data < node->left->data) {
85     return rightRotate(node);
86 }
87 if (balance < -1 && data > node->right->data) {
88     return leftRotate(node);
89 }
90 if (balance > 1 && data > node->left->data) {
91     return leftRightRotate(node);
92 }
93 if (balance < -1 && data < node->right->data) {
94     return rightLeftRotate(node);
95 }
96
97 return node;
98 }
99
100 // Print the tree (In-order Traversal)
101 void inOrder(struct Node* root) {
102     if (root != NULL) {
103         inOrder(root->left);
104         printf("%d ", root->data);
105         inOrder(root->right);
106     }
107 }
108
109 // Main function
110 int main() {
111     struct Node* root = NULL;
112
113     // Inserting nodes
114     root = insert(root, 10);
115     root = insert(root, 20);
116     root = insert(root, 30); // Causes Left-Left case (rotation)
117
118     printf("In-order Traversal: ");
119     inOrder(root); // Should print the nodes in sorted order
120     printf("\n");
121
122     return 0;
123 }
```

Output

/tmp/WOSM7Zz80H.o
In-order Traversal: 10 20 30

--- Code Execution Successful ---

1. AVL Tree Deletion

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Node {
5     int key;
6     struct Node *left;
7     struct Node *right;
8     int height;
9 };
10
11 // Get height of the node
12 int height(struct Node *N) {
13     if (N == NULL) return 0;
14     return N->height;
15 }
16
17 // Get maximum of two integers
18 int max(int a, int b) {
19     return (a > b) ? a : b;
20 }
21
22 // Right rotate subtree rooted with y
23 struct Node* rightRotate(struct Node *y) {
24     struct Node *x = y->left;
25     struct Node *T2 = x->right;
26
27     // Perform rotation
28     x->right = y;
29     y->left = T2;
30
31     // Update heights
32     y->height = max(height(y->left), height(y->right)) + 1;
33     x->height = max(height(x->left), height(x->right)) + 1;
34
35     // Return new root
36     return x;
37 }
38
39 // Left rotate subtree rooted with x
40 struct Node* leftRotate(struct Node *x) {
41     struct Node *y = x->right;
42     struct Node *T2 = y->left;
43
44     // Perform rotation
```

Output

/tmp/t3ophXwTZh.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30

--- Code Execution Successful ---

main.c	Run	Output
<pre>45 y->left = x; 46 x->right = T2; 47 48 // Update heights 49 x->height = max(height(x->left), height(x->right)) + 1; 50 y->height = max(height(y->left), height(y->right)) + 1; 51 52 // Return new root 53 return y; 54 } 55 56 // Get balance factor of node N 57 int getBalance(struct Node *N) { 58 if (N == NULL) return 0; 59 return height(N->left) - height(N->right); 60 } 61 62 // Recursive function to delete a node 63 struct Node* deleteNode(struct Node* root, int key) { 64 // Step 1: Perform normal BST delete 65 if (root == NULL) return root; 66 67 // If key to be deleted is smaller than the root's key, then it lies in left subtree 68 if (key < root->key) 69 root->left = deleteNode(root->left, key); 70 71 // If key to be deleted is greater than the root's key, then it lies in right subtree 72 else if (key > root->key) 73 root->right = deleteNode(root->right, key); 74 75 // If key is the same as root's key, then this is the node to be deleted 76 else { 77 // Node with only one child or no child 78 if ((root->left == NULL) (root->right == NULL)) { 79 struct Node *temp = root->left ? root->left : root->right; 80 81 // No child case 82 if (temp == NULL) { 83 temp = root; 84 root = NULL; 85 } else // One child case 86 *root = *temp; 87 88 free(temp);</pre>	<div>Run</div>	<pre>/tmp/t3ophXwTZn.o Preorder traversal before deletion: 10 20 5 25 30 Preorder traversal after deletion: 10 20 5 25 30 === Code Execution Successful ===</pre>

main.c	Run	Output
<pre>88 free(temp); 89 } else { 90 // Node with two children: Get the inorder successor (smallest in the right subtree 91 struct Node* temp = root->right; 92 while (temp->left != NULL) { 93 temp = temp->left; 94 } 95 96 // Copy the inorder successor's content to this node 97 root->key = temp->key; 98 99 // Delete the inorder successor 100 root->right = deleteNode(root->right, temp->key); 101 } 102 } 103 104 // If the tree had only one node, return 105 if (root == NULL) return root; 106 107 // Step 2: Update height of this ancestor node 108 root->height = max(height(root->left), height(root->right)) + 1; 109 110 // Step 3: Get the balance factor of this node (to check whether it became unbalanced) 111 int balance = getBalance(root); 112 113 // If this node becomes unbalanced, then there are 4 cases 114 115 // Left Left Case 116 if (balance > 1 && key < root->left->key) 117 return rightRotate(root); 118 119 // Right Right Case 120 if (balance < -1 && key > root->right->key) 121 return leftRotate(root); 122 123 // Left Right Case 124 if (balance > 1 && key > root->left->key) { 125 root->left = leftRotate(root->left); 126 return rightRotate(root); 127 } 128 129 // Right Left Case 130 if (balance < -1 && key < root->right->key) {</pre>	<div>Run</div>	<pre>/tmp/t3ophXwTZn.o Preorder traversal before deletion: 10 20 5 25 30 Preorder traversal after deletion: 10 20 5 25 30 === Code Execution Successful ===</pre>



main.c	Run	Output
<pre>130- if (balance < -1 && key < root->right->key) { 131- root->right = rightRotate(root->right); 132- return leftRotate(root); 133- } 134- 135- return root; 136- } 137- 138- // A utility function to print preorder traversal of the tree 139- void preOrder(struct Node *root) { 140- if (root != NULL) { 141- printf("%d ", root->key); 142- preOrder(root->left); 143- preOrder(root->right); 144- } 145- } 146- 147- // Helper function to create a new node 148- struct Node* newNode(int key) { 149- struct Node* node = (struct Node*)malloc(sizeof(struct Node)); 150- node->key = key; 151- node->left = node->right = NULL; 152- node->height = 1; // New node is initially at height 1 153- return node; 154- } 155- 156- int main() { 157- struct Node* root = NULL; 158- 159- // Insert nodes 160- root = newNode(10); 161- root->left = newNode(20); 162- root->right = newNode(30); 163- root->left->left = newNode(5); 164- root->left->right = newNode(25); 165- 166- printf("Preorder traversal before deletion:\n"); 167- preOrder(root); 168- printf("\n"); 169- 170- // Delete a node 171- root = deleteNode(root, 20); 172- 173- printf("Preorder traversal after deletion:\n");</pre>	<div>Run</div>	<pre>/tmp/t3ophXwTZn.o Preorder traversal before deletion: 10 20 5 25 30 Preorder traversal after deletion: 10 20 5 25 30 === Code Execution Successful ===</pre>

main.c	Run	Output
<pre>137- 138- // A utility function to print preorder traversal of the tree 139- void preOrder(struct Node *root) { 140- if (root != NULL) { 141- printf("%d ", root->key); 142- preOrder(root->left); 143- preOrder(root->right); 144- } 145- } 146- 147- // Helper function to create a new node 148- struct Node* newNode(int key) { 149- struct Node* node = (struct Node*)malloc(sizeof(struct Node)); 150- node->key = key; 151- node->left = node->right = NULL; 152- node->height = 1; // New node is initially at height 1 153- return node; 154- } 155- 156- int main() { 157- struct Node* root = NULL; 158- 159- // Insert nodes 160- root = newNode(10); 161- root->left = newNode(20); 162- root->right = newNode(30); 163- root->left->left = newNode(5); 164- root->left->right = newNode(25); 165- 166- printf("Preorder traversal before deletion:\n"); 167- preOrder(root); 168- printf("\n"); 169- 170- // Delete a node 171- root = deleteNode(root, 20); 172- 173- printf("Preorder traversal after deletion:\n"); 174- preOrder(root); 175- printf("\n"); 176- 177- return 0; 178- }</pre>	<div>Run</div>	<pre>/tmp/t3ophXwTZn.o Preorder traversal before deletion: 10 20 5 25 30 Preorder traversal after deletion: 10 20 5 25 30 === Code Execution Successful ===</pre>

2. Heap Sort Implementation

main.c	Run	Output
<pre>1 #include <stdio.h> 2 3 void heapify(int arr[], int n, int i) { 4 int largest = i; 5 int left = 2 * i + 1; 6 int right = 2 * i + 2; 7 8 if (left < n && arr[left] > arr[largest]) 9 largest = left; 10 11 if (right < n && arr[right] > arr[largest]) 12 largest = right; 13 14 if (largest != i) { 15 int temp = arr[i]; 16 arr[i] = arr[largest]; 17 arr[largest] = temp; 18 heapify(arr, n, largest); 19 } 20 } 21 22 void heapSort(int arr[], int n) { 23 for (int i = n / 2 - 1; i >= 0; i--) 24 heapify(arr, n, i); 25 26 for (int i = n - 1; i >= 0; i--) { 27 int temp = arr[0]; 28 arr[0] = arr[i]; 29 arr[i] = temp; 30 heapify(arr, i, 0); 31 } 32 } 33 34 void printArray(int arr[], int size) { 35 for (int i = 0; i < size; i++) 36 printf("%d ", arr[i]); 37 printf("\n"); 38 } 39 40 int main() { 41 int arr[] = {12, 11, 13, 5, 6, 7}; 42 int n = sizeof(arr) / sizeof(arr[0]); 43 44 printf("Unsorted array: ");</pre>	Run	<pre>/tmp/KS9upjXAYh.o Unsorted array: 12 11 13 5 6 7 Sorted array: 5 6 7 11 12 13 --- Code Execution Successful ---</pre>
<pre>12 largest = right; 13 14 if (largest != i) { 15 int temp = arr[i]; 16 arr[i] = arr[largest]; 17 arr[largest] = temp; 18 heapify(arr, n, largest); 19 } 20 } 21 22 void heapSort(int arr[], int n) { 23 for (int i = n / 2 - 1; i >= 0; i--) 24 heapify(arr, n, i); 25 26 for (int i = n - 1; i >= 0; i--) { 27 int temp = arr[0]; 28 arr[0] = arr[i]; 29 arr[i] = temp; 30 heapify(arr, i, 0); 31 } 32 } 33 34 void printArray(int arr[], int size) { 35 for (int i = 0; i < size; i++) 36 printf("%d ", arr[i]); 37 printf("\n"); 38 } 39 40 int main() { 41 int arr[] = {12, 11, 13, 5, 6, 7}; 42 int n = sizeof(arr) / sizeof(arr[0]); 43 44 printf("Unsorted array: "); 45 printArray(arr, n); 46 47 heapSort(arr, n); 48 49 printf("Sorted array: "); 50 printArray(arr, n); 51 52 return 0; 53 }</pre>	Run	<pre>/tmp/KS9upjXAYh.o Unsorted array: 12 11 13 5 6 7 Sorted array: 5 6 7 11 12 13 --- Code Execution Successful ---</pre>

3. Priority Queue using Heap

main.c	Run	Output
<pre>1 #include <stdio.h> 2 #include <stdlib.h> 3 4 struct PriorityQueue { 5 int* heap; 6 int size; 7 int capacity; 8 }; 9 10 void swap(int* a, int* b) { 11 int temp = *a; 12 *a = *b; 13 *b = temp; 14 } 15 16 void heapify(struct PriorityQueue* pq, int i) { 17 int largest = i; 18 int left = 2 * i + 1; 19 int right = 2 * i + 2; 20 21 if (left < pq->size && pq->heap[left] > pq->heap[largest]) 22 largest = left; 23 24 if (right < pq->size && pq->heap[right] > pq->heap[largest]) 25 largest = right; 26 27 if (largest != i) { 28 swap(&pq->heap[i], &pq->heap[largest]); 29 heapify(pq, largest); 30 } 31 } 32 33 void insert(struct PriorityQueue* pq, int value) { 34 if (pq->size == pq->capacity) { 35 printf("Priority Queue is full\n"); 36 return; 37 } 38 39 pq->heap[pq->size] = value; 40 int i = pq->size; 41 pq->size++; 42 43 // Fix the max heap property if it is violated 44 while (i != 0 && pq->heap[(i - 1) / 2] < pq->heap[i]) {</pre>		<pre>/tmp/qC670CCeZR.o Priority Queue (Max-Heap): 30 20 5 10 15 Extracted max: 30 Priority Queue after extraction: 20 15 5 10 === Code Execution Successful ===</pre>
<pre>44 while (i != 0 && pq->heap[(i - 1) / 2] < pq->heap[i]) { 45 swap(&pq->heap[i], &pq->heap[(i - 1) / 2]); 46 i = (i - 1) / 2; 47 } 48 } 49 50 int extractMax(struct PriorityQueue* pq) { 51 if (pq->size <= 0) { 52 printf("Priority Queue is empty\n"); 53 return -1; 54 } 55 if (pq->size == 1) { 56 pq->size--; 57 return pq->heap[0]; 58 } 59 60 // Store the maximum value (root), and remove it from the heap 61 int root = pq->heap[0]; 62 pq->heap[0] = pq->heap[pq->size - 1]; 63 pq->size--; 64 65 // Call heapify to restore heap property 66 heapify(pq, 0); 67 68 return root; 69 } 70 71 void printPriorityQueue(struct PriorityQueue* pq) { 72 for (int i = 0; i < pq->size; i++) { 73 printf("%d ", pq->heap[i]); 74 } 75 printf("\n"); 76 } 77 78 int main() { 79 struct PriorityQueue pq; 80 pq.size = 0; 81 pq.capacity = 10; 82 pq.heap = (int*)malloc(pq.capacity * sizeof(int)); 83 84 insert(&pq, 10); 85 insert(&pq, 20); 86 insert(&pq, 5); 87 insert(&pq, 30);</pre>		<pre>/tmp/qC670CCeZR.o Priority Queue (Max-Heap): 30 20 5 10 15 Extracted max: 30 Priority Queue after extraction: 20 15 5 10 === Code Execution Successful ===</pre>

main.c	Output
<pre>61 int root = pq->heap[0]; 62 pq->heap[0] = pq->heap[pq->size - 1]; 63 pq->size--; 64 65 // Call heapify to restore heap property 66 heapify(pq, 0); 67 68 return root; 69 } 70 71 void printPriorityQueue(struct PriorityQueue* pq) { 72 for (int i = 0; i < pq->size; i++) { 73 printf("%d ", pq->heap[i]); 74 } 75 printf("\n"); 76 } 77 78 int main() { 79 struct PriorityQueue pq; 80 pq.size = 0; 81 pq.capacity = 10; 82 pq.heap = (int*)malloc(pq.capacity * sizeof(int)); 83 84 insert(&pq, 10); 85 insert(&pq, 20); 86 insert(&pq, 5); 87 insert(&pq, 30); 88 insert(&pq, 15); 89 90 printf("Priority Queue (Max-Heap): "); 91 printPriorityQueue(&pq); 92 93 printf("Extracted max: %d\n", extractMax(&pq)); 94 95 printf("Priority Queue after extraction: "); 96 printPriorityQueue(&pq); 97 98 free(pq.heap); 99 100 return 0; 101 }</pre>	<pre>/tmp/qC670CCeZR.o Priority Queue (Max-Heap): 30 20 5 10 15 Extracted max: 30 Priority Queue after extraction: 20 15 5 10 --- Code Execution Successful ---</pre>