Index

|  |  |  |  |
| --- | --- | --- | --- |
| S/NO | Question | Page | Signature |
| **Module: Iterative & recursive method** | | | |
|  | Write a C/C++ program to print Fibonacci series upto nth term  using iteration also compute time complexity |  |  |
|  | Write a C/C++ program to print Fibonacci series up to nth term using recursion also compute the time complexity in terms of input size. |  |  |
| **Module: Searching** | | | |
|  | Write a C/C++ program using linear search to search an element  in an array also compute time complexity for an input of size N. |  |  |
|  | Recursive Write a C/C++ program to perform binary search on  an array of size N and compute time complexity for size N. |  |  |
| **Module: Sorting** | | | |
|  | Write a C/C++ program to perform bubble sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform insertion sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform selection sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform merge sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform quick sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform count sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
|  | Write a C/C++ program to perform radix sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
| **Module: Heap** | | | |
|  | Write a C/C++ program to insert an element into heap, also  compute time complexity for an input of size N. |  |  |
|  | Write a C/C++ program to delete the N element from a heap,  also compute time complexity for those N elements. |  |  |
|  | Write a C/C++ program to build a heap using heapify and use it to perform heap sort, also compute the time complexity for an input of size N |  |  |
|  | Write a C/C++ program to perform heap sort on an integer  array to sort it in ascending order and compute the time  complexity for an input of size N. |  |  |
| **Module: Amortized Analysis** | | | |
|  | Write a C/C++ program to implement dynamic array. First take maximum length of array from user input. Then start by creating array of size 1 and start taking input. Every time the array is full, double its capacity. Use amortizes analysis (aggregate) to calculate time complexity of the program. |  |  |
|  | Write C/C++ program to implement stack with the use of array. Make a new function Multi Pop which pops k times. Take k as user input. Uses amortize analysis (accounting) to calculate time complexity of the program. |  |  |
| **Module: String Matching** | | | |
|  | Write C/C++ program to implement KMP string matching  method to find the pattern string in a text string both given by the user. Compute the complexity of the method for a text string of length N and pattern string of length M, where N>M. |  |  |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to Fibonacci series up to nth term using iteration also compute time complexity | |
| **Input**  #include <stdio.h>  static int operations = 0;  void printFib(int n) {  if (n < 1) {  printf("Invalid Number of terms\n");  operations++;  return;  }  int prev1 = 1;  int prev2 = 0;  operations += 2;  printf("%d ", prev2);  operations++;  if (n == 1) {  operations++;  return;  }  printf("%d ", prev1);  for (int i = 3; i <= n; i++) {  int curr = prev1 + prev2;  prev2 = prev1;  prev1 = curr;  printf("%d ", curr);  operations++;  }  }  int main() {  int n = 9;  operations++;  printFib(n);  printf("\nTotal operations performed: %d\n", operations);  operations++;  return 0;  } | **Output**  Your Output  0 1 1 2 3 5 8 13 21  Total operations performed: 11 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to print Fibonacci series up to nth term using recursive also compute time complexity | |
| **Input**  #include <stdio.h>  static int count = 0;  int fibbonacci(int n) {  if(n == 0){  return 0;  count++;  } else if(n == 1) {  return 1;  count++;  } else {  return (fibbonacci(n-1) + fibbonacci(n-2));  count++;  }  }  int main() {  int n = 5;  int i;  printf("Fibbonacci of %d: " , n);    for(i = 0;i<n;i++) {  printf("%d ",fibbonacci(i));  count++;  }  printf("\nTime complexity : %d\n", count);  } | **Output**  Fibbonacci of 5: 0 1 1 2 3  Time complexity : 5 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program using linear search to search an element   in an array also compute time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  int main(){  int n;  int key = 10;  printf("Enter the size of the array: ");  scanf("%d", &n);  int \*arr = (int\*)malloc(sizeof(int)\*n);  printf("Enter the elements of the array:\n");  for(int i=0; i<n; i++){  scanf("%d", &arr[i]);  if(arr[i] == key){  printf("Element %d found at index %d\n", key, i);  break;  }else if(i == n-1){  printf("Element %d not found in the array\n", key);  }  }} | **Output**  Enter the size of the array: 5  Enter the elements of the array:  10 20 40 1 100  Element 10 found at index 0 |

|  |  |
| --- | --- |
| 1. Recursive Write a C/C++ program to perform binary search on   an array of size N and compute time complexity for size N. | |
| **Input**  #include <stdio.h>  static int operations = 0;  int binary\_search(int arr[], int low, int high, int x) {  operations++; // Counting function call  if (high >= low) {  int mid = (high + low) / 2;  operations++; // Counting mid calculation  if (arr[mid] == x) {  operations++;  return mid;  }  if (arr[mid] > x) {  operations++;  return binary\_search(arr, low, mid - 1, x);  }  operations++;  return binary\_search(arr, mid + 1, high, x);  }  return -1; // Return -1 if element is not found  }  int main() {  int arr[] = {2, 3, 4, 10, 40};  int n = sizeof(arr) / sizeof(arr[0]);  int x = 10;  operations++;  int result = binary\_search(arr, 0, n - 1, x);  operations++;  if (result == -1) {  printf("Element is not present in array\n");  } else {  printf("Element found at index %d\n", result);  }  printf("Number of operations performed: %d\n", operations);  return 0;  } | **Output**  Element found at index 3  Number of operations performed: 8  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform bubble sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  int main(){  int n;  int count = 0;  printf("Enter the size of the array: \n");  scanf("%d", &n);  int \*arr = (int\*)malloc(sizeof(int) \* n);  printf("Enter the elements of the array: ");  for(int i = 0; i < n; i++){  scanf("%d", &arr[i]);  count ++;  }  for(int i = 0; i < n-1; i++){  count++;  for(int j = 0; j < n-i-1; j++){  if(arr[j] > arr[j+1]){  int temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  count++;  }  }  }  printf("Sorted array in ascending order:\n ");  for(int i = 0; i < n; i++){  printf("%d \t ", arr[i]);  count++;  }  printf("\nTotal number of comparisons: %d\n", count);  free(arr);  } | **Output**  Enter the size of the array:  5  Enter the elements of the array: 1 4 0 30 90  Sorted array in ascending order:  0 1 4 30 90  Total number of comparisons: 16 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform insertion sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N | |
| **Input**  #include <stdio.h>  void insertionSort(int arr[], int n) {  for (int i = 1; i < n; i++) {  int key = arr[i];  int j = i - 1;    while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;    printf("After iteration %d: ", i);  for (int k = 0; k < n; k++) {  printf("%d ", arr[k]);  }  printf("\n");  }  }  int main() {  int n;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array: ");  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  insertionSort(arr, n);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } | **Output**  Enter the size of the array: 5  Enter the elements of the array: 1 100 0 4 900  After iteration 1: 1 100 0 4 900  After iteration 2: 0 1 100 4 900  After iteration 3: 0 1 4 100 900  After iteration 4: 0 1 4 100 900  Sorted array: 0 1 4 100 900  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform selection sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  int main(){  int n;  int count = 0;  printf("Enter the size of the array: ");  scanf("%d", &n);  int \*arr = (int\*)malloc(sizeof(int) \* n);  printf("Enter the elements of the array: ");  for(int i = 0; i < n; i++){  scanf("%d", &arr[i]);  count++;  }  for (int i = 0; i < n; i++){  int min\_idx = i;  for (int j = i+1; j < n; j++){  if (arr[j] < arr[min\_idx])  min\_idx = j;  count++;  }  int temp = arr[i];  arr[i] = arr[min\_idx];  arr[min\_idx] = temp;  count++;  }  printf("Sorted array in ascending order:\n");  for(int i = 0; i < n; i++){  printf("%d ", arr[i]);  count++;  }  printf("\nTotal number of comparisons: %d\n", count);  } | **Output**  Enter the size of the array: 5  Enter the elements of the array: 100 4 10 50 700  Sorted array in ascending order:  4 10 50 100 700  Total number of comparisons: 25  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform merge sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  static int operations = 0;  void mergeSort(int arr[], int left, int right);  void merge(int arr[], int left, int mid, int right);  void merge(int arr[], int left, int mid, int right) {  int n1 = mid - left + 1;  int n2 = right - mid;  int L[n1], R[n2];  for (int i = 0; i < n1; i++)  L[i] = arr[left + i];  for (int j = 0; j < n2; j++)  R[j] = arr[mid + 1 + j];  int i = 0, j = 0, k = left;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  } else {  arr[k] = R[j];  j++;  }  k++;  }  while (i < n1) {  arr[k] = L[i];  i++;  k++;  }  while (j < n2) {  arr[k] = R[j];  j++;  k++;  }  printf("After iteration %d: ", mid - left + 1);  for (int i = left; i <= right; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return;  }  void mergeSort(int arr[], int left, int right) {  if (left < right) {  int mid = left + (right - left) / 2;  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }  return;  }  int main() {  int n;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array: ");  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  printf("Before sorting: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  mergeSort(arr, 0, n - 1);  printf("After sorting: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } | **Output**  Enter the size of the array: 5  Enter the elements of the array: 10 1 0 100 70  Before sorting: 10 1 0 100 70  After iteration 1: 1 10  After iteration 2: 0 1 10  After iteration 1: 70 100  After iteration 3: 0 1 10 70 100  After sorting: 0 1 10 70 100  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform quick sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  int comparisonCount = 0; // Global variable to count comparisons  void swap(int\* a, int\* b) {  int t = \*a;  \*a = \*b;  \*b = t;  }  int partition(int arr[], int low, int high) {  int pivot = arr[high];  int i = (low - 1);  for (int j = low; j <= high - 1; j++) {  comparisonCount++; // Increment comparison count  if (arr[j] <= pivot) {  i++;  swap(&arr[i], &arr[j]);  }  }  swap(&arr[i + 1], &arr[high]);  return (i + 1);  }  void quickSort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  void printArray(int arr[], int size) {  for (int i = 0; i < size; i++)  printf("%d ", arr[i]);  printf("\n");  }  int main() {  int arr[] = {10, 7, 8, 9, 1, 5};  int n = sizeof(arr) / sizeof(arr[0]);    printf("Original array: ");  printArray(arr, n);    quickSort(arr, 0, n - 1);    printf("Sorted array: ");  printArray(arr, n);  printf("Number of comparisons: %d\n", comparisonCount);    return 0;  } | **Output**  Original array: 10 7 8 9 1 5  Sorted array: 1 5 7 8 9 10  Number of comparisons: 11 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform count sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int comparisonCount = 0; // Global variable to count comparisons  void countSort(int arr[], int n) {  int max = arr[0], min = arr[0];    // Find the range of the array  for (int i = 1; i < n; i++) {  comparisonCount++; // Increment comparison count  if (arr[i] > max) max = arr[i];  if (arr[i] < min) min = arr[i];  }    int range = max - min + 1;  int\* count = (int\*)calloc(range, sizeof(int)); // Initialize count array  int\* output = (int\*)malloc(n \* sizeof(int)); // Output array    // Store count of occurrences  for (int i = 0; i < n; i++) {  count[arr[i] - min]++;  }    // Change count[i] to store the position of this element in output array  for (int i = 1; i < range; i++) {  comparisonCount++; // Increment comparison count  count[i] += count[i - 1];  }    // Build the output array  for (int i = n - 1; i >= 0; i--) {  output[count[arr[i] - min] - 1] = arr[i];  count[arr[i] - min]--;  }    // Copy the sorted elements back to original array  for (int i = 0; i < n; i++) {  arr[i] = output[i];  }    // Free allocated memory  free(count);  free(output);  }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int arr[] = {4, 2, 2, 8, 3, 3, 1};  int n = sizeof(arr) / sizeof(arr[0]);    printf("Original array: ");  printArray(arr, n);    countSort(arr, n);    printf("Sorted array: ");  printArray(arr, n);    // Time complexity of Count Sort is O(N + K), where K is the range of numbers.  printf("Time Complexity: O(N + K)\n");  printf("Number of comparisons: %d\n", comparisonCount);    return 0;  } | **Output**  Original array: 4 2 2 8 3 3 1  Sorted array: 1 2 2 3 3 4 8  Time Complexity: O(N + K)  Number of comparisons: 13 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform radix sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  int comparisonCount = 0; // Global variable to count comparisons  int getMax(int arr[], int n) {  int max = arr[0];  for (int i = 1; i < n; i++) {  comparisonCount++; // Increment comparison count  if (arr[i] > max)  max = arr[i];  }  return max;  }  void countSort(int arr[], int n, int exp) {  int\* output = (int\*)malloc(n \* sizeof(int));  int count[10] = {0};    for (int i = 0; i < n; i++) {  count[(arr[i] / exp) % 10]++;  }    for (int i = 1; i < 10; i++) {  comparisonCount++; // Increment comparison count  count[i] += count[i - 1];  }    for (int i = n - 1; i >= 0; i--) {  output[count[(arr[i] / exp) % 10] - 1] = arr[i];  count[(arr[i] / exp) % 10]--;  }    for (int i = 0; i < n; i++) {  arr[i] = output[i];  }    free(output);  }  void radixSort(int arr[], int n) {  int max = getMax(arr, n);  for (int exp = 1; max / exp > 0; exp \*= 10) {  comparisonCount++; // Increment comparison count  countSort(arr, n, exp);  }  }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  }  int main() {  int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};  int n = sizeof(arr) / sizeof(arr[0]);    printf("Original array: ");  printArray(arr, n);    radixSort(arr, n);    printf("Sorted array: ");  printArray(arr, n);    // Time complexity of Radix Sort is O(N \* K), where K is the number of digits in the largest number.  printf("Time Complexity: O(N \* K)\n");  printf("Number of comparisons: %d\n", comparisonCount);    return 0;  } | **Output**  Original array: 170 45 75 90 802 24 2 66  Sorted array: 2 24 45 66 75 90 170 802  Time Complexity: O(N \* K)  Number of comparisons: 37  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to insert an element into heap, also compute time complexity for an input of size N. | |
| **Input**  #include <iostream>  #include <vector>  #include <ctime>  using namespace std;  // Global variable to count comparisons  int comparisonCount = 0;  // Function to heapify a subtree rooted at index i  void heapify(vector<int>& heap, int n, int i) {  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;    if (left < n) {  comparisonCount++;  if (heap[left] > heap[largest])  largest = left;  }  if (right < n) {  comparisonCount++;  if (heap[right] > heap[largest])  largest = right;  }    if (largest != i) {  swap(heap[i], heap[largest]);  heapify(heap, n, largest);  }  }  // Function to insert an element into the heap  void insertElement(vector<int>& heap, int element) {  clock\_t start = clock();  heap.push\_back(element);  int i = heap.size() - 1;  while (i > 0) {  comparisonCount++;  if (heap[(i - 1) / 2] < heap[i]) {  swap(heap[i], heap[(i - 1) / 2]);  i = (i - 1) / 2;  } else {  break;  }  }  clock\_t end = clock();  cout << "Time taken for insertion: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;  }  // Function to delete the root element (max element)  void deleteElement(vector<int>& heap) {  if (heap.size() == 0) return;  clock\_t start = clock();  heap[0] = heap.back();  heap.pop\_back();  heapify(heap, heap.size(), 0);  clock\_t end = clock();  cout << "Time taken for deletion: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;  }  // Function to build a max heap using heapify  void buildHeap(vector<int>& heap) {  int n = heap.size();  for (int i = n / 2 - 1; i >= 0; i--) {  heapify(heap, n, i);  }  }  // Heap sort function  void heapSort(vector<int>& arr) {  clock\_t start = clock();  buildHeap(arr);  for (int i = arr.size() - 1; i > 0; i--) {  swap(arr[0], arr[i]);  heapify(arr, i, 0);  }  clock\_t end = clock();  cout << "Time taken for Heap Sort: " << double(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;  }  // Function to display heap  void displayHeap(const vector<int>& heap) {  for (int val : heap) {  cout << val << " ";  }  cout << endl;  }  int main() {  vector<int> heap = {10, 20, 15, 30, 40};  buildHeap(heap);  cout << "Initial Heap: ";  displayHeap(heap);  // Insert an element  insertElement(heap, 50);  cout << "Heap after insertion: ";  displayHeap(heap);  // Delete N elements  int N = 2;  for (int i = 0; i < N; i++) {  deleteElement(heap);  }  cout << "Heap after deleting " << N << " elements: ";  displayHeap(heap);  // Heap Sort  vector<int> arr = {10, 20, 15, 30, 40, 50};  heapSort(arr);  cout << "Sorted Array: ";  displayHeap(arr);  cout << "Total number of comparisons: " << comparisonCount << endl;  return 0;  } | **Output**  Initial Heap: 40 30 15 10 20  Time taken for insertion: 0.000001 seconds  Heap after insertion: 50 40 15 10 20 30  Time taken for deletion: 0.000002 seconds  Time taken for deletion: 0.000002 seconds  Heap after deleting 2 elements: 30 20 15 10  Time taken for Heap Sort: 0.000003 seconds  Sorted Array: 10 15 20 30 40 50  Total number of comparisons: 12 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform quick sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N. | |
| **Input**  #include <iostream>  #include <vector>  #include <chrono>  class MaxHeap {  private:  std::vector<int> heap;  int comparisons; // Counter for comparisons  void heapifyUp(int index) {  while (index > 0) {  int parent = (index - 1) / 2;  comparisons++; // Count the comparison  if (heap[index] > heap[parent]) {  std::swap(heap[index], heap[parent]);  index = parent;  } else {  break;  }  }  }  public:  MaxHeap() : comparisons(0) {}  void insert(int value) {  heap.push\_back(value);  heapifyUp(heap.size() - 1);  }  void display() {  for (int val : heap) {  std::cout << val << " ";  }  std::cout << std::endl;  }  int getComparisons() const {  return comparisons;  }  void resetComparisons() {  comparisons = 0;  }  };  int main() {  MaxHeap heap;  int n, value;  std::cout << "Enter number of elements to insert into heap: ";  std::cin >> n;  auto start = std::chrono::high\_resolution\_clock::now();  for (int i = 0; i < n; i++) {  std::cout << "Enter element " << i + 1 << ": ";  std::cin >> value;  heap.insert(value);  }  auto end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<double> duration = end - start;  std::cout << "\nHeap after insertion: ";  heap.display();  std::cout << "\nTotal comparisons made: " << heap.getComparisons() << std::endl;  std::cout << "Time taken to insert " << n << " elements: "  << duration.count() << " seconds" << std::endl;  // Time complexity analysis  std::cout << "\nTime Complexity Analysis:" << std::endl;  std::cout << "- Single insertion operation: O(log N) comparisons" << std::endl;  std::cout << "- Inserting N elements: O(N log N) comparisons" << std::endl;  std::cout << "- Actual comparisons for " << n << " elements: " << heap.getComparisons() << std::endl;  // Theoretical vs actual comparison  std::cout << "\nTheoretical vs Actual Comparisons:" << std::endl;  int theoretical\_max = n \* log2(n); // Upper bound  std::cout << "- Theoretical maximum (N log N): " << theoretical\_max << std::endl;  std::cout << "- Actual comparisons: " << heap.getComparisons() << std::endl;  std::cout << "- Ratio (Actual/Theoretical): "  << (double)heap.getComparisons()/theoretical\_max << std::endl;  return 0;  } | **Output**  Enter number of elements to insert into heap: 5  Enter element 1: 10  Enter element 2: 20  Enter element 3: 15  Enter element 4: 30  Enter element 5: 5  Heap after insertion: 30 20 15 10 5  Total comparisons made: 7  Time taken to insert 5 elements: 2.8e-05 seconds |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to delete the N element from a heap, also compute time complexity for those N elements. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <math.h>  int comparisons = 0; // Global comparison counter  typedef struct {  int \*array;  int capacity;  int size;  } MaxHeap;  MaxHeap\* createHeap(int capacity) {  MaxHeap\* heap = (MaxHeap\*)malloc(sizeof(MaxHeap));  heap->array = (int\*)malloc(capacity \* sizeof(int));  heap->capacity = capacity;  heap->size = 0;  return heap;  }  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  void heapifyDown(MaxHeap\* heap, int index) {  int largest = index;  int left = 2 \* index + 1;  int right = 2 \* index + 2;  comparisons++;  if (left < heap->size && heap->array[left] > heap->array[largest]) {  largest = left;  }  comparisons++;  if (right < heap->size && heap->array[right] > heap->array[largest]) {  largest = right;  }  if (largest != index) {  swap(&heap->array[index], &heap->array[largest]);  heapifyDown(heap, largest);  }  }  void heapifyUp(MaxHeap\* heap, int index) {  while (index > 0) {  int parent = (index - 1) / 2;  comparisons++;  if (heap->array[index] > heap->array[parent]) {  swap(&heap->array[index], &heap->array[parent]);  index = parent;  } else {  break;  }  }  }  void insert(MaxHeap\* heap, int value) {  if (heap->size == heap->capacity) {  printf("Heap is full!\n");  return;  }  heap->array[heap->size] = value;  heapifyUp(heap, heap->size);  heap->size++;  }  int deleteMax(MaxHeap\* heap) {  if (heap->size == 0) {  printf("Heap is empty!\n");  return -1;  }  int max = heap->array[0];  heap->array[0] = heap->array[heap->size - 1];  heap->size--;  heapifyDown(heap, 0);  return max;  }  void printHeap(MaxHeap\* heap) {  for (int i = 0; i < heap->size; i++) {  printf("%d ", heap->array[i]);  }  printf("\n");  }  void deleteNElements(MaxHeap\* heap, int n) {  if (n > heap->size) {  printf("Cannot delete %d elements, heap only has %d elements.\n", n, heap->size);  return;  }  clock\_t start = clock();  comparisons = 0; // Reset comparison counter  printf("Deleting %d elements: ", n);  for (int i = 0; i < n; i++) {  printf("%d ", deleteMax(heap));  }  printf("\n");  clock\_t end = clock();  double time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nTime taken to delete %d elements: %f seconds\n", n, time\_spent);  printf("Total comparisons made: %d\n", comparisons);  // Time complexity analysis  printf("\nTime Complexity Analysis:\n");  printf("- Single deletion operation: O(log N) comparisons\n");  printf("- Deleting N elements: O(N log M) comparisons (where M is original heap size)\n");    // Theoretical vs actual comparison  int original\_size = heap->size + n;  int theoretical\_max = n \* log2(original\_size); // Upper bound  printf("\nTheoretical vs Actual Comparisons:\n");  printf("- Theoretical maximum (N log M): %d\n", theoretical\_max);  printf("- Actual comparisons: %d\n", comparisons);  printf("- Ratio (Actual/Theoretical): %f\n", (double)comparisons/theoretical\_max);  }  int main() {  int capacity, num\_elements, value, delete\_n;  printf("Enter heap capacity: ");  scanf("%d", &capacity);  MaxHeap\* heap = createHeap(capacity);  printf("Enter number of elements to insert into heap: ");  scanf("%d", &num\_elements);  if (num\_elements > capacity) {  printf("Number of elements exceeds heap capacity!\n");  return 1;  }  printf("Enter %d elements:\n", num\_elements);  for (int i = 0; i < num\_elements; i++) {  scanf("%d", &value);  insert(heap, value);  }  printf("\nHeap before deletion: ");  printHeap(heap);  printf("\nEnter number of elements to delete from heap: ");  scanf("%d", &delete\_n);  deleteNElements(heap, delete\_n);  printf("\nHeap after deletion: ");  printHeap(heap);  free(heap->array);  free(heap);  return 0;  } | **Output**  Enter heap capacity: 10  Enter number of elements to insert into heap: 6  Enter 6 elements:  10 20 15 30 25 5  Heap before deletion: 30 25 15 10 20 5  Enter number of elements to delete from heap: 3  Deleting 3 elements: 30 25 20  Time taken to delete 3 elements: 0.000003 seconds  Total comparisons made: 10 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to build a heap using heapify and use it to perform heap sort, also compute the time complexity for an input of size N. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <math.h>  // Global counter for comparisons  unsigned long long comparisons = 0;  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  // Heapify a subtree rooted at index i  void heapify(int arr[], int n, int i) {  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;  // Compare with left child  comparisons++;  if (left < n && arr[left] > arr[largest]) {  largest = left;  }  // Compare with right child  comparisons++;  if (right < n && arr[right] > arr[largest]) {  largest = right;  }  // If largest is not root  if (largest != i) {  swap(&arr[i], &arr[largest]);  heapify(arr, n, largest); // Recursively heapify the affected subtree  }  }  // Build a max heap from array using heapify  void buildHeap(int arr[], int n) {  // Start from last non-leaf node and heapify each node  for (int i = n / 2 - 1; i >= 0; i--) {  heapify(arr, n, i);  }  }  // Perform heap sort  void heapSort(int arr[], int n) {  // Build initial max heap (O(n) time)  buildHeap(arr, n);  // Extract elements one by one (O(n log n) time)  for (int i = n - 1; i > 0; i--) {  // Move current root to end  swap(&arr[0], &arr[i]);  // Heapify the reduced heap  heapify(arr, i, 0);  }  }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  void analyzeTimeComplexity(int n) {  printf("\nTime Complexity Analysis for N = %d:\n", n);  printf("1. Build Heap Operation:\n");  printf(" - Theoretical: O(N)\n");  printf(" - Explanation: Although heapify is O(log N), building heap from bottom-up results in O(N) total operations\n");    printf("\n2. Heap Sort Operation:\n");  printf(" - Theoretical: O(N log N) for all cases\n");  printf(" - Breakdown:\n");  printf(" \* Building heap: O(N)\n");  printf(" \* N heapify operations during extraction: O(N log N)\n");  printf(" \* Total dominated by O(N log N)\n");    printf("\n3. Space Complexity:\n");  printf(" - O(1) auxiliary space (in-place sorting)\n");    printf("\n4. Comparisons Analysis:\n");  double nlogn = n \* log2(n);  printf(" - Theoretical upper bound (N log N): %.2f\n", nlogn);  printf(" - Actual comparisons: %llu\n", comparisons);  printf(" - Ratio (Actual/Theory): %.2f\n", comparisons/nlogn);  }  int main() {  int n;  printf("Enter number of elements: ");  scanf("%d", &n);  int \*arr = (int\*)malloc(n \* sizeof(int));    // Generate random numbers  srand(time(0));  for (int i = 0; i < n; i++) {  arr[i] = rand() % 1000; // Random numbers between 0-999  }  printf("\nOriginal array (first 20 elements): ");  printArray(arr, n > 20 ? 20 : n);  clock\_t start = clock();  comparisons = 0; // Reset comparison counter  heapSort(arr, n);  clock\_t end = clock();  double time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;  printf("\nSorted array (first 20 elements): ");  printArray(arr, n > 20 ? 20 : n);  printf("\nExecution Time: %.6f seconds\n", time\_spent);  printf("Total Comparisons: %llu\n", comparisons);  // Detailed time complexity analysis  analyzeTimeComplexity(n);  free(arr);  return 0;  } | **Output**  Enter number of elements: 10000  Original array (first 20 elements): 383 886 777 915 793 ...  Sorted array (first 20 elements): 0 0 1 1 2 ...  Execution Time: 0.002345 seconds  Total Comparisons: 235618 |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to perform heap sort on an integer array to sort it in ascending order and compute the time complexity for an input of size N | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  int comparisonCount = 0; // Global variable to count comparisons  void heapify(int arr[], int n, int i) {  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;  if (left < n) {  comparisonCount++;  if (arr[left] > arr[largest])  largest = left;  }    if (right < n) {  comparisonCount++;  if (arr[right] > arr[largest])  largest = right;  }    if (largest != i) {  int temp = arr[i];  arr[i] = arr[largest];  arr[largest] = temp;  heapify(arr, n, largest);  }  }  void insertHeap(int arr[], int \*n, int value) {  (\*n)++;  arr[\*n - 1] = value;    int i = \*n - 1;  while (i > 0 && arr[(i - 1) / 2] < arr[i]) {  comparisonCount++;  int temp = arr[i];  arr[i] = arr[(i - 1) / 2];  arr[(i - 1) / 2] = temp;  i = (i - 1) / 2;  }  }  void printArray(int arr[], int n) {  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  }  int main() {  int arr[100] = {10, 20, 30, 25, 5, 40, 35};  int n = 7;    printf("Original Heap: ");  printArray(arr, n);    int value = 50;  insertHeap(arr, &n, value);    printf("Heap after insertion: ");  printArray(arr, n);    // Time complexity of heap insertion is O(log N)  printf("Time Complexity: O(log N)\n");  printf("Number of comparisons: %d\n", comparisonCount);    return 0;  } | **Output**  Original Heap: 10 20 30 25 5 40 35  Heap after insertion: 50 10 30 20 5 40 35 25  Time Complexity: O(log N)  Number of comparisons: 3  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write a C/C++ program to implement dynamic array. First take maximum length of array from user input. Then start by creating array of size 1, and start taking input. Every time the array is full, double its capacity. Use amortize analysis (aggregate) to calculate time complexity of the program. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  void printArray(int\* arr, int size) {  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int maxSize;  printf("Enter the maximum length of the array: ");  scanf("%d", &maxSize);  int currentSize = 0;  int capacity = 1;  int\* arr = (int\*)malloc(capacity \* sizeof(int));  if (arr == NULL) {  printf("Memory allocation failed!\n");  return 1;  }  printf("Start entering numbers (up to %d):\n", maxSize);  for (int i = 0; i < maxSize; i++) {  int value;  scanf("%d", &value);    if (currentSize == capacity) {  capacity \*= 2;  arr = (int\*)realloc(arr, capacity \* sizeof(int));  if (arr == NULL) {  printf("Memory reallocation failed!\n");  return 1;  }  }    arr[currentSize++] = value;  }  printf("Final Dynamic Array: ");  printArray(arr, currentSize);    // Amortized time complexity: O(1) for insertions, O(N) for total insertions due to doubling  printf("Amortized Time Complexity: O(1) per insertion, O(N) total\n");    free(arr);  return 0;  } | **Output**  Enter the maximum length of the array: 5  Start entering numbers (up to 5):  10 3 4 1 2 100  Final Dynamic Array: 10 3 4 1 2  Amortized Time Complexity: O(1) per insertion, O(N) total  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write C/C++ program to implement stack with the use of array. Make a new function Multi Pop which pops k times. Take k as user input. Uses amortize analysis (accounting) to calculate time complexity of the program. | |
| **Input**  #include <stdio.h>  #include <stdlib.h>  #define MAX 100  int stack[MAX];  int top = -1;  void push(int value) {  if (top < MAX - 1) {  stack[++top] = value;  printf("Pushed %d onto the stack\n", value);  } else {  printf("Stack Overflow!\n");  }  }  int pop() {  if (top >= 0) {  printf("Popped %d from the stack\n", stack[top]);  return stack[top--];  } else {  printf("Stack Underflow!\n");  return -1;  }  }  void multiPop(int k) {  for (int i = 0; i < k; i++) {  pop();  }  }  void printStack() {  if (top == -1) {  printf("Stack is empty!\n");  } else {  printf("Current stack: ");  for (int i = 0; i <= top; i++) {  printf("%d ", stack[i]);  }  printf("\n");  }  }  int main() {  int k;  int value;  printf("Enter number of elements to push onto the stack: ");  int n;  scanf("%d", &n);  for (int i = 0; i < n; i++) {  printf("Enter value to push: ");  scanf("%d", &value);  push(value);  }  printf("Enter the number of elements to pop: ");  scanf("%d", &k);  multiPop(k);    printStack();  // Amortized time complexity for push and pop: O(1) for each operation  printf("Amortized Time Complexity: O(1) per operation for push and pop\n");  return 0;  } | **Output**  Enter number of elements to push onto the stack: 3  Enter value to push: 19  Pushed 19 onto the stack  Enter value to push: 12  Pushed 12 onto the stack  Enter value to push: 100  Pushed 100 onto the stack  Enter the number of elements to pop: 19  Popped 100 from the stack  Popped 12 from the stack  Popped 19 from the stack  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack Underflow!  Stack is empty!  Amortized Time Complexity: O(1) per operation for push and pop  === Code Execution Successful === |

|  |  |
| --- | --- |
| 1. Write C/C++ program to implement KMP string matching method to find the pattern string in a text string both given by the user. Compute the complexity of the method for a text string of length N and pattern string of length M, where N>M | |
| **Input**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  // Function to compute the longest prefix suffix (LPS) array  void computeLPSArray(char\* pattern, int M, int\* lps) {  int len = 0; // Length of the previous longest prefix suffix  lps[0] = 0; // lps[0] is always 0  int i = 1;  while (i < M) {  if (pattern[i] == pattern[len]) {  len++;  lps[i] = len;  i++;  } else {  if (len != 0) {  len = lps[len - 1];  } else {  lps[i] = 0;  i++;  }  }  }  }  // KMP string matching algorithm  void KMPSearch(char\* pattern, char\* text) {  int M = strlen(pattern);  int N = strlen(text);  // Create LPS array  int\* lps = (int\*)malloc(M \* sizeof(int));  computeLPSArray(pattern, M, lps);  int i = 0; // Index for text[]  int j = 0; // Index for pattern[]  int comparisons = 0;  int patternOccurrences = 0;  while (i < N) {  comparisons++;  if (pattern[j] == text[i]) {  j++;  i++;  }  if (j == M) {  printf("Pattern found at index %d\n", i - j);  patternOccurrences++;  j = lps[j - 1];  } else if (i < N && pattern[j] != text[i]) {  if (j != 0) {  j = lps[j - 1];  } else {  i++;  }  }  }  printf("\nTotal comparisons made: %d\n", comparisons);  printf("Total pattern occurrences: %d\n", patternOccurrences);  free(lps);  }  int main() {  char text[1000];  char pattern[100];  printf("Enter the text string: ");  fgets(text, sizeof(text), stdin);  text[strcspn(text, "\n")] = '\0'; // Remove newline character  printf("Enter the pattern string to search: ");  fgets(pattern, sizeof(pattern), stdin);  pattern[strcspn(pattern, "\n")] = '\0'; // Remove newline character  printf("\nSearching for pattern '%s' in text...\n", pattern);  KMPSearch(pattern, text);  // Time complexity analysis  printf("\nTime Complexity Analysis:\n");  printf("1. Preprocessing (LPS array construction): O(M)\n");  printf("2. Searching phase: O(N)\n");  printf("3. Total time complexity: O(N + M)\n");  printf(" where N = text length (%lu), M = pattern length (%lu)\n", strlen(text), strlen(pattern));  printf("\nSpace Complexity: O(M) (for LPS array)\n");  return 0;  } | **Output**  Enter the text string: ABABDABACDABABCABAB  Enter the pattern string to search: ABABCABAB  Searching for pattern 'ABABCABAB' in text...  Pattern found at index 10  Total comparisons made: 26  Total pattern occurrences: 1 |