**Data Structures**

Data structures are arrangements of data in computer memory that allow for efficient storage, organization, and access. They enable various operations to be performed easily.

**1. Primitive Data Structures (PDS)**

These are the basic data types provided by programming languages. They are not composed of other data types.

* **Integer (int)**: Represents whole numbers.
* **Floating-point (float)**: Represents numbers with decimal points.
* **Character (char)**: Represents single characters.
* **Double (double)**: Represents double-precision floating-point numbers.
* **Boolean (bool)**: Represents true or false values.

**2. Non-Primitive Data Structures (NPDS)**

These are more complex data structures that are derived from primitive data structures. They can be categorized as linear or non-linear.

* **Linear Data Structures**: Elements are arranged sequentially.
  + **Array**: A collection of elements identified by index or key.
  + **Stack**: A collection of elements that follows the Last In First Out (LIFO) principle.
  + **Queue**: A collection of elements that follows the First In First Out (FIFO) principle.
  + **Linked List**: A collection of nodes where each node contains data and a reference to the next node.
* **Non-Linear Data Structures**: Elements are not arranged in a sequential manner.
  + **Graph**: A collection of nodes (vertices) connected by edges, which can represent relationships.
  + **Tree**: A hierarchical structure with a root value and subtrees of children, representing parent-child relationships.

These data structures help in efficiently organizing data and performing various operations, making them essential for programming and software development. If you have any specific questions about any of these structures, feel free to ask!

### Sorting Techniques

Sorting techniques depend on the situation and are influenced by two main parameters:

1. **Time Complexity**: This refers to the execution time of a sorting algorithm, indicating how the time taken to sort data grows with the size of the input.
2. **Space Complexity**: This refers to the amount of memory space required by the algorithm to perform the sorting.

Sorting can be performed using several techniques or methods. Some common sorting algorithms include:

**Sorting Techniques**

1. **Bubble Sort**
   * **Time Complexity**: O(n²) (average and worst case)
   * **Space Complexity**: O(1) (in-place)
   * **Description**: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
2. **Selection Sort**
   * **Time Complexity**: O(n²)
   * **Space Complexity**: O(1) (in-place)
   * **Description**: Divides the list into a sorted and an unsorted region. Repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region.
3. **Insertion Sort**
   * **Time Complexity**: O(n²) (average and worst case)
   * **Space Complexity**: O(1) (in-place)
   * **Description**: Builds the sorted array one element at a time by repeatedly picking the next element and inserting it into the correct position.
4. **Merge Sort**
   * **Time Complexity**: O(n log n)
   * **Space Complexity**: O(n) (not in-place)
   * **Description**: Divides the array into two halves, sorts each half, and then merges the sorted halves back together.
5. **Quick Sort**
   * **Time Complexity**: O(n log n) (average), O(n²) (worst case)
   * **Space Complexity**: O(log n) (in-place)
   * **Description**: Selects a "pivot" element and partitions the array into elements less than and greater than the pivot, recursively sorting the sub-arrays.
6. **Heap Sort**
   * **Time Complexity**: O(n log n)
   * **Space Complexity**: O(1) (in-place)
   * **Description**: Builds a binary heap from the input data and repeatedly extracts the maximum (or minimum) element from the heap to build a sorted array.
7. **Counting Sort**
   * **Time Complexity**: O(n + k) (where k is the range of the input)
   * **Space Complexity**: O(k)
   * **Description**: Counts the number of occurrences of each unique value in the input and calculates the positions of each element in the sorted output.
8. **Radix Sort**
   * **Time Complexity**: O(nk) (where k is the number of digits in the largest number)
   * **Space Complexity**: O(n + k)
   * **Description**: Sorts numbers by processing individual digits. It performs counting sort on each digit.

**Summary**

* **Bubble Sort Techniques**
* **Data Size**: For small datasets, simpler algorithms like insertion sort may be effective.
* **Data Characteristics**: If the data is nearly sorted, insertion sort performs well.
* **Memory Constraints**: In-place algorithms like quick sort or heap sort may be preferred when memory usage is a concern.
* **Stability Requirements**: Some algorithms (like merge sort) maintain the relative order of equal elements, which may be important in certain applications.

|  |
| --- |
| **Bubble Sort Techniques** |
| No. of passing(n-1) ,total swaping (depend on data) |
| Dictionary order is follow bubble sort |
| It is depend on value and structure |
| Asked by company   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | Best Case  **Omega Notation (Ω)**: | Worst Case  **Big O Notation (O)** | Average Case  **Theta Notation (Θ)** | Space complexity | Is it stable or not | approach | In place or not | |  |  |  |  |  |  |  |   O bigo, omega,theta |
| Here's the table rewritten with symbols:   | **Characteristic** | **Best Case** | **Worst Case** | **Average Case** | **Space Complexity** | **Stability** | **Approach** | **In-Place** | | --- | --- | --- | --- | --- | --- | --- | --- | | **Bubble Sort** | O(𝑛) | O(𝑛²) | O(𝑛²) | O(1) | ✔ (Stable) | Comparison-based | ✔ (Yes) | | **Selection Sort** | O(𝑛²) | O(𝑛²) | O(𝑛²) | O(1) | ✘ (Unstable) | Comparison-based | ✔ (Yes) | | **Insertion Sort** | O(𝑛) | O(𝑛²) | O(𝑛²) | O(1) | ✔ (Stable) | Comparison-based | ✔ (Yes) | | **Merge Sort** | O(𝑛 log 𝑛) | O(𝑛 log 𝑛) | O(𝑛 log 𝑛) | O(𝑛) | ✔ (Stable) | Divide and conquer | ✘ (No) | | **Quick Sort** | O(𝑛 log 𝑛) | O(𝑛²) | O(𝑛 log 𝑛) | O(log 𝑛) | ✘ (Unstable) | Divide and conquer | ✔ (Yes) | | **Heap Sort** | O(𝑛 log 𝑛) | O(𝑛 log 𝑛) | O(𝑛 log 𝑛) | O(1) | ✘ (Unstable) | Comparison-based | ✔ (Yes) | | **Counting Sort** | O(𝑛 + 𝑘) | O(𝑛 + 𝑘) | O(𝑛 + 𝑘) | O(𝑘) | ✔ (Stable) | Non-comparison | ✘ (No) | | **Radix Sort** | O(𝑛𝑘) | O(𝑛𝑘) | O(𝑛𝑘) | O(𝑛 + 𝑘) | ✔ (Stable) | Non-comparison | ✘ (No) | | **Bucket Sort** | O(𝑛 + 𝑘) | O(𝑛²) | O(𝑛 + 𝑘) | O(𝑛) | ✔ (Stable) | Non-comparison | ✘ (No) |   **Symbols:**   * **O(𝑛)**: Linear time complexity * **O(𝑛²)**: Quadratic time complexity * **O(𝑛 log 𝑛)**: Linearithmic time complexity * **O(1)**: Constant space complexity * **O(log 𝑛)**: Logarithmic space complexity * **O(𝑘)**: Linear space complexity related to input range * **✔**: Yes/True * **✘**: No/False |

02/10/2024

|  |
| --- |
| Buble sort |
| 1. **Instance Swapping**: If you're talking about swapping instances (such as elements or nodes) in a data structure, you might be referring to swapping values or objects between positions in an array or linked list. 2. **Issue with Value Check Not Lasting**: If the swapping doesn't persist after checking the values, it could mean that either the swap is not being executed correctly, or the data structure is being modified in a way that undoes the swap. 3. **Adjacent Node in Bubble Sort**: In bubble sort, the algorithm compares **adjacent elements** and swaps them if they are out of order. If swapping isn't working as expected, make sure:    * You are correctly implementing the **swap logic**.    * The indices or nodes you're working with are correctly identified. 4. **Fix for Swapping Issue**:    * Ensure that after each swap, the updated values are correctly assigned back to the original positions.    * Check if the loop structure (like for or while) is correctly iterating through the elements or nodes to compare adjacent ones. |
| Brute force alogrithme |
| // bubble sort example  #include <iostream>  using namespace std;  int main()  {      int arr[] = {10, 20, 30, 60, 50, 10};      int length = sizeof(arr) / sizeof(arr[0]); // Calculate the length of the array   cout << "Before Swapping" << endl;      for (int i = 0; i < length; i++)      {          cout << arr[i] << " "; // Print each element      }   cout << endl;      cout << "After Swapping" << endl;      for (int i = 0; i < length; i++)      {          for (int j = i + 1; j < length; j++)          {              if (arr[i] > arr[j])              {                  int temp = arr[i];                  arr[i] = arr[j];                  arr[j] = temp;              }          }      }      for (int i = 0; i < length; i++)      {          cout << arr[i] << " ";      }      cout << endl;      return 0;  } |
|  |

03/10/2024

|  |
| --- |
| What is a Data Structure? Data Structure is a way to organized data in such a way that it can be used efficiently. Following terms are foundation terms of a data structure.   * **Interface** − Each data strucure has an interface. Interface represents the set of operations that a datastructure supports.An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations. * **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the defination of the alogrithms used in the opreations of the data structure.  Characteristics of a Data Structure  * **Correctness** − Data Structure implementation should implement its interface correctly. * **Time Complexity** − Running time or execution time of operations of data structure must be as small as possible. * **Space Complexity** − Memory usage of a data structure operation should be as little as possible.  Need for Data Structure As applications are getting complex and data rich, there are three common problems applications face now-a-days.   * **Data Search** − Consider an inventory of 1 million(106) items of a store. If application is to search an item. It has to search item in 1 million(106) items every time slowing down the search. As data grows, search will become slower. * **Processor speed** − Processor speed although being very high, falls limited if data grows to billon records. * **Multiple requests** − As thousands of users can search data simultaneously on a web server,even very fast server fails while searching the data.   To solve above problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be search and required data can be searched almost instantly. Execution Time Cases There are three cases which are usual used to compare various data structure's execution time in relative manner.   * **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If a operation's worst case time is ƒ(n) then this operation will not take time more than ƒ(n) time where ƒ(n) represents function of n. * **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If a operation takes ƒ(n) time in execution then m operations will take mƒ(n) time. * **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If a operation takes ƒ(n) time in execution then actual operation may take time as random number which would be maximum as ƒ(n). |
| Asymptotic Notations Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.   * Ο Notation * Ω Notation * θ Notation |
| Big Oh Notation, Ο  Omega Notation, Ω  Theta Notation, θ |

|  |
| --- |
| Bubble sort |
| Value, structure based  Upperbound 🡪 n2 time complexity  Lowerbound 🡪 n  Why constant is Nagiable 🡪after taken 10, n2 will increase for after live time so constant is negible  Variable is called constant so it is negiable   |  |  | | --- | --- | | N=1,2,3,4,5,6,7,8,9,10 |  | | 2n | n2 | | 2 is not countable |  | |  |  | | Acending order time complexity order of n | Average and worst case time complexity order of n2 |     Space complexity  order of 1(constant) , I use 1 array so  order on n\*n, I use 2 array so many  Approach  Subtract & conqouer |
| 1. **Instance Swapping**: If you're talking about swapping instances (such as elements or nodes) in a data structure, you might be referring to swapping values or objects between positions in an array or linked list. 2. **Issue with Value Check Not Lasting**: If the swapping doesn't persist after checking the values, it could mean that either the swap is not being executed correctly, or the data structure is being modified in a way that undoes the swap. 3. **Adjacent Node in Bubble Sort**: In bubble sort, the algorithm compares **adjacent elements** and swaps them if they are out of order. If swapping isn't working as expected, make sure:    * You are correctly implementing the **swap logic**.    * The indices or nodes you're working with are correctly identified. 4. **Fix for Swapping Issue**:    * Ensure that after each swap, the updated values are correctly assigned back to the original positions.    * Check if the loop structure (like for or while) is correctly iterating through the elements or nodes to compare adjacent ones. |
|  |
| // bubble sort example  #include <iostream>  using namespace std;  int main()  {      // int arr[] = {10, 20, 30, 60, 50, 10};      // int arr[] = {10, 20, 30,40,50, 60};      // case 1 Worst Case --> decending order give data      // int arr[]={60,50,40,30,20,10};      // case 2   Average Case --->      //  int arr[]={30,20,50,10,40,60};      //  case 3 Best case --> accending order      // int arr[] = {10, 20, 30, 40, 50, 60};      //   case 4 Random Case --> Random data      int arr[] = {10, 20, 30, 1000, 40, 50};      int length = sizeof(arr) / sizeof(arr[0]); // Calculate the length of the array      cout << "Before Swapping" << endl;      for (int i = 0; i < length; i++)      {          cout << arr[i] << " "; // Print each element      }      cout << endl;      int count;      cout << "After Swapping" << endl;      for (int i = 0; i < length; i++)      {  count=0;          for (int j = 0; j < length - i - 1; j++)          {              if (arr[j] > arr[j + 1])              {                  int temp = arr[j];                  arr[j] = arr[j + 1];                  arr[j + 1] = temp;                  count++;              }          }          if (count == 0)          {              cout<<"Sorted Array Already"<<endl;              break;          }      }      for (int i = 0; i < length; i++)      {          cout << arr[i] << " ";      }      cout << endl;      return 0;  } |
| Brute Force Alogrithme  Yes, the approach you have implemented to find the maximum element in the array can be considered a **brute force algorithm**. Here's why:   * **Brute force algorithms** typically involve solving a problem by checking all possible solutions or, in this case, iterating through the entire dataset without any optimizations. * In your code, you are iterating through the entire array once using a simple loop, checking each element individually to find the maximum. This straightforward method exhaustively examines every element to determine the answer. * The time complexity of this algorithm is **O(n)**, where n is the number of elements in the array. In this case, for an array of size n, you are visiting each element once to check if it is greater than the current maximum.   In short, this is a brute-force approach because it directly inspects each element without any clever shortcuts or optimizations. |
| #include<iostream>  using namespace std;  int main() {      int arr[] = {10, 20, 50, 60, 40, 30, 5};      int n = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array      int maxElement = arr[0]; // Assume the first element is the maximum      // Iterate through the array to find the maximum element      for (int i = 1; i < n; i++) {          if (arr[i] > maxElement) {              maxElement = arr[i]; // Update maxElement if a larger element is found          }      }      // Print the maximum element found      cout << "The maximum element in the array is: " << maxElement << endl;      return 0;  } |

|  |
| --- |
| Brute force code |
| #include <iostream>  using namespace std;  int main()  {      // Example cases:      // int arr[] = {10, 20, 30, 60, 50, 10};    // Random case      // int arr[] = {60, 50, 40, 30, 20, 10};    // Worst case (descending order)      // int arr[] = {10, 20, 30, 40, 50, 60};    // Best case (already sorted)      int arr[] = {10, 20, 30, 1000, 40, 50};    // Random case (example)      int length = sizeof(arr) / sizeof(arr[0]); // Calculate the length of the array      // Print the array before sorting      cout << "Before Swapping" << endl;      for (int i = 0; i < length; i++)      {          cout << arr[i] << " "; // Print each element      }      cout << endl;      // Brute force bubble sort (no optimizations)      cout << "After Swapping" << endl;      for (int i = 0; i < length; i++)      {          for (int j = 0; j < length ; j++)          {              if (arr[j] > arr[j + 1])              {                  // Swap the elements if they are in the wrong order                  int temp = arr[j];                  arr[j] = arr[j + 1];                  arr[j + 1] = temp;              }          }      }      // Print the sorted array      for (int i = 0; i < length; i++)      {          cout << arr[i] << " ";      }      cout << endl;      return 0;  } |
|  |
|  |
|  |