C++ Data Structure Algorithm Resource

chat.openai.com/chat

write code of all data structre and algoritm in c++ with code and explanation

I'm sorry, but writing code and explanations for all data structures and algorithms in C++ is not possible in this format as it would require a vast amount of text. However, I can provide you with resources where you can learn about them.

Here is a list of popular data structures in C++:

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Heaps
- Graphs
- Hash Tables

For algorithms, here is a list of popular algorithms in C++:

- Sorting algorithms (Bubble sort, Quick sort, Merge sort, etc.)
- Searching algorithms (Linear search, Binary search, etc.)
- Dynamic Programming algorithms
- Greedy algorithms
- Tree algorithms (Traversal, Search, etc.)
- Graph algorithms (BFS, DFS, Shortest path, etc.)

You can find code implementations and explanations for these data structures and algorithms in various online tutorials, books, and forums. Websites such as GeeksForGeeks, LeetCode, and HackerRank have a vast collection of problems and solutions for data structures and algorithms that you can learn from.

write codes of Arrays Linked Lists Stacks Queues Trees Heaps Graphs Hash Tables in c++

Here's a brief code implementation of the popular data structures in C++:

Arrays:

```
c



#include <iostream>
using namespace std;

int main() {
  int arr[5];
  cout << "Enter 5 integer values: ";
  for (int i = 0; i < 5; i++) {
    cin >> arr[i];
  }
  cout << "The values in the array are: ";
  for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
  }
  return 0;
}
```

Linked Lists:

cpp

```cpp
#include <iostream>
using namespace std;

class Node {
public:
  int data;
  Node* next;
};

int main() {
  Node* head = NULL;
  Node* second = NULL;
  Node* third = NULL;

  head = new Node();
  second = new Node();
  third = new Node();

  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;

  Node* temp = head;
  while (temp != NULL) {
    cout << temp->data << " ";
    temp = temp->next;
  }
  return 0;
}
```

Stacks:

c

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
  stack<int> s;
  s.push(1);
  s.push(2);
  s.push(3);

  cout << "Top element: " << s.top() << endl;
  s.pop();
  cout << "Top element after pop: " << s.top() << endl;
  return 0;
}
```

Queues:

c

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
  queue<int> q;
  q.push(1);
  q.push(2);
  q.push(3);

  cout << "First element: " << q.front() << endl;
  q.pop();
  cout << "First element after pop: " << q.front() << endl;
  return 0;
}
```

Trees:

c

```
#include <iostream>
using namespace std;

class Node {
public:
  int data;
  Node* left;
  Node* right;
};

int main() {
  Node* root = NULL;
  root = new Node();
  root->data = 1;
  root->left = new Node();
  root->left->data = 2;
  root->right = new Node();
  root->right->data = 3;
  cout << "Root: " << root->data << endl;
  cout << "Left child: " << root->left->data << endl;
  cout << "Right child: " << root->right->data << endl;
  return 0;
}
```

Heaps:

c

```
#include <iostream
```

write code also for Heaps Graphs Hash Tables in c++

Heaps:

cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void heapify(vector<int>& v, int n, int i) {
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;

  if (l < n && v[l] > v[largest]) {
    largest = l;
  }

  if (r < n && v[r] > v[largest]) {
    largest = r;
  }

  if (largest != i) {
    swap(v[i], v[largest]);
    heapify(v, n, largest);
  }
}

void buildHeap(vector<int>& v, int n) {
  for (int i = n / 2 - 1; i >= 0; i--) {
    heapify(v, n, i);
  }
}

int main() {
  vector<int> v{4, 10, 3, 5, 1};
  int n = v.size();
  buildHeap(v, n);
  cout << "Heap array: ";
  for (int i = 0; i < n; i++) {
    cout << v[i] << " ";
  }
  return 0;
}
```

Graphs:

cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Graph {
  int V;
  vector<int>* adj;
public:
  Graph(int V) {
    this->V = V;
    adj = new vector<int>[V];
  }
  void addEdge(int v, int w) {
    adj[v].push_back(w);
  }
  void printGraph() {
    for (int i = 0; i < V; i++) {
      cout << "Adjacency list of vertex " << i << ": ";
      for (int j = 0; j < adj[i].size(); j++) {
        cout << adj[i][j] << " ";
      }
      cout << endl;
    }
  }
};

int main() {
  Graph g(5);
  g.addEdge(0, 1);
  g.addEdge(0, 4);
  g.addEdge(1, 2);
  g.addEdge(1, 3);
  g.addEdge(1, 4);
  g.addEdge(2, 3);
  g.addEdge(3, 4);
  g.printGraph();
  return 0;
}
```

Hash Tables:

```
c
```

```
#include <iostream>
#include <unordered_map>
using namespace std;

int main() {
  unordered_map<int, int> h;
  h[1] = 100;
  h[2] = 200;
  h[3] = 300;

  for (auto x : h) {
    cout << x.first << " " << x.second << endl;
  }
  return 0;
}
```

now write codes for Sorting algorithms (Bubble sort, Quick sort, Merge sort, etc.) Searching algorithms (Linear search, Binary search, etc.) Dynamic Programming algorithms Greedy algorithms Tree algorithms (Traversal, Search, etc.) Graph algorithms (BFS, DFS, Shortest path, etc.) in c++

Sorting Algorithms:

Bubble sort:

cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int>& arr) {
  int n = arr.size();
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        swap(arr[j], arr[j + 1]);
      }
    }
  }
}

int main() {
  vector<int> arr{64, 34, 25, 12, 22, 11, 90};
  int n = arr.size();
  bubbleSort(arr);
  cout << "Sorted array: ";
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  return 0;
}
```

Quick sort:

cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

int partition(vector<int>& arr, int low, int high) {
  int pivot = arr[high];
  int i = low - 1;
  for (int j = low; j <= high - 1; j++) {
    if (arr[j] <= pivot) {
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[high]);
  return (i + 1);
}

void quickSort(vector<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);
  }
}

int main() {
  vector<int> arr{10, 7, 8, 9, 1, 5};
  int n = arr.size();
  quickSort(arr, 0, n - 1);
  cout << "Sorted array: ";
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  return 0;
}
```

Merge sort:

```
cpp



#include <iostream>
#include <vector>
using namespace std;

void merge(vector<int>& arr, int l, int m, int r) {
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;
  vector<int> L(n1), R(n2);
  for (i = 0; i < n1; i++) {
    L[i] = arr[l + i];
  }
  for (j = 0; j < n2; j++) {
    R[j] = arr[m + 1 + j];
  }
  i = 0;
  j = 0;
  k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
```

also write for Searching algorithms (Linear search, Binary search, etc.) Dynamic Programming algorithms Greedy algorithms Tree algorithms (Traversal, Search, etc.) Graph algorithms (BFS, DFS, Shortest path, etc.) in c++

Searching Algorithms:

Linear search:

c

```
#include <iostream>
#include <vector>
using namespace std;

int linearSearch(vector<int>& arr, int target) {
  int n = arr.size();
  for (int i = 0; i < n; i++) {
    if (arr[i] == target) {
      return i;
    }
  }
  return -1;
}

int main() {
  vector<int> arr{2, 4, 5, 7, 10};
  int target = 5;
  int index = linearSearch(arr, target);
  if (index != -1) {
    cout << "Element found at index " << index << endl;
  } else {
    cout << "Element not found." << endl;
  }
  return 0;
}
```

Binary search:

c

```
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(vector<int>& arr, int target) {
  int low = 0;
  int high = arr.size() - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
      return mid;
    }
    if (arr[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}

int main() {
  vector<int> arr{2, 4, 5, 7, 10};
  int target = 7;
  int index = binarySearch(arr, target);
  if (index != -1) {
    cout << "Element found at index " << index << endl;
  } else {
    cout << "Element not found." << endl;
  }
  return 0;
}
```

Dynamic Programming Algorithms:

Fibonacci sequence (bottom-up approach):

c

```
#include <iostream>
using namespace std;

int fibonacci(int n) {
  int f[n + 1];
  f[0] = 0;
  f[1] = 1;
  for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
  }
  return f[n];
}

int main() {
  int n = 9;
  cout << "Fibonacci number at " << n << " is " << fibonacci(n) << endl;
  return 0;
}
```

Greedy Algorithms:

Fractional Knapsack Problem:

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Item {
  int value;
  int weight;
};

bool compare(Item a, Item b) {
  return a.value / a.weight > b.value / b.weight;
}

double fractionalKnapsack(int W, vector<Item>& items) {
  sort(items.begin(), items.end(), compare
```

ALSO WRITE FOR Tree algorithms (Traversal, Search, etc.) Graph algorithms (BFS, DFS, Shortest path, etc.) in c++

Tree Algorithms:

Binary Tree Traversal (Inorder):

```
cpp



#include <iostream>
using namespace std;

struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
};

void inorderTraversal(TreeNode* root) {
  if (!root) return;
  inorderTraversal(root->left);
  cout << root->data << " ";
  inorderTraversal(root->right);
}

int main() {
  TreeNode* root = new TreeNode({4,
    new TreeNode({2,
      new TreeNode({1, NULL, NULL},
      new TreeNode({3, NULL, NULL}),
    }),
    new TreeNode({6,
      new TreeNode({5, NULL, NULL}),
      new TreeNode({7, NULL, NULL}),
    }),
  });
  cout << "Inorder traversal: ";
  inorderTraversal(root);
  cout << endl;
  return 0;
}
```

Tree Search (Binary Search Tree):

cpp

```
#include <iostream>
using namespace std;

struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
};

bool search(TreeNode* root, int target) {
  if (!root) return false;
  if (root->data == target) return true;
  if (root->data < target) return search(root->right, target);
  return search(root->left, target);
}

int main() {
  TreeNode* root = new TreeNode({4,
    new TreeNode({2,
      new TreeNode({1, NULL, NULL},
      new TreeNode({3, NULL, NULL}),
    }),
    new TreeNode({6,
      new TreeNode({5, NULL, NULL}),
      new TreeNode({7, NULL, NULL}),
    }),
  });
  int target = 5;
  if (search(root, target)) {
    cout << target << " is found in the tree." << endl;
  } else {
    cout << target << " is not found in the tree." << endl;
  }
  return 0;
}
```

Graph Algorithms:

Breadth-First Search:

```
cpp



#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<vector<int>> adjList;
vector<bool> visited;

void BFS(int start) {
  queue<int> q;
  q.push(start);
  visited[start] = true;
  while (!q.empty()) {
    int curr = q.front();
    q.pop();
    cout << curr << " ";
    for (int i = 0; i < adjList[curr].size(); i++) {
      int next = adjList[curr][i];
      if (!visited[next]) {
        q.push(next);
        visited[next] = true;
      }
    }
  }
}

int main() {
```

ALSO FOR Graph algorithms (BFS, DFS, Shortest path, etc.) in c++

Graph Algorithms:

Breadth-First Search:

c

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<vector<int>> adjList;
vector<bool> visited;

void BFS(int start) {
  queue<int> q;
  q.push(start);
  visited[start] = true;
  while (!q.empty()) {
    int curr = q.front();
    q.pop();
    cout << curr << " ";
    for (int i = 0; i < adjList[curr].size(); i++) {
      int next = adjList[curr][i];
      if (!visited[next]) {
        q.push(next);
        visited[next] = true;
      }
    }
  }
}

int main() {
  int n, m;
  cin >> n >> m;
  adjList.resize(n);
  visited.resize(n);
  for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adjList[u].push_back(v);
    adjList[v].push_back(u);
  }
  cout << "BFS traversal: ";
  BFS(0);
  cout << endl;
  return 0;
}
```

Depth-First Search:

c

```
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int>> adjList;
vector<bool> visited;

void DFS(int curr) {
  visited[curr] = true;
  cout << curr << " ";
  for (int i = 0; i < adjList[curr].size(); i++) {
    int next = adjList[curr][i];
    if (!visited[next]) DFS(next);
  }
}

int main() {
  int n, m;
  cin >> n >> m;
  adjList.resize(n);
  visited.resize(n);
  for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adjList[u].push_back(v);
    adjList[v].push_back(u);
  }
  cout << "DFS traversal: ";
  DFS(0);
  cout << endl;
  return 0;
}
```

Shortest Path (Dijkstra's Algorithm):

cpp

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

const int INF = numeric_limits<int>::max();
vector<vector<pair<int, int>>> adjList;
vector<int> dist;
vector<bool> visited;
int n, m;

void Dijkstra(int start) {
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
  pq.push({0, start});
  dist[start]
```

COMPLETE THE CODE OF Shortest Path (Dijkstra's Algorithm):

cpp

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

const int INF = numeric_limits<int>::max();
vector<vector<pair<int, int>>> adjList;
vector<int> dist;
vector<bool> visited;
int n, m;

void Dijkstra(int start) {
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
  pq.push({0, start});
  dist[start] = 0;
  while (!pq.empty()) {
    int curr = pq.top().second;
    pq.pop();
    if (visited[curr]) continue;
    visited[curr] = true;
    for (int i = 0; i < adjList[curr].size(); i++) {
      int next = adjList[curr][i].first, weight = adjList[curr][i].second;
      if (dist[next] > dist[curr] + weight) {
        dist[next] = dist[curr] + weight;
        pq.push({dist[next], next});
      }
    }
  }
}

int main() {
  cin >> n >> m;
  adjList.resize(n);
  dist.resize(n, INF);
  visited.resize(n);
  for (int i = 0; i < m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    adjList[u].push_back({v, w});
    adjList[v].push_back({u, w});
  }
  Dijkstra(0);
  for (int i = 0; i < n; i++) cout << dist[i] << " ";
  cout << endl;
  return 0;
}
```