

# Data\_structure\_note

Thursday, May 30, 2024 2:42 PM

**1. Data Structure** => is a way of organizing data different ways of storing data on your computer

EX: is like preparing for cooking you slice up vegetables

**2. Algorithm** => the process that does something to data to produce the output operations on different data structure

**data structure + algorithm = programming**

---

Steps to learn:

a.. Big O notation => is the language of analyzing algorithms and data structures.

b. Data structures list:

1. Arrays
2. Linked Lists
3. Queues and Stacks
4. Trees
5. Graphs
6. Hash Maps
7. pointers
8. Memory => 1 byte = 8 bits

c. Algorithms:

1. recursion
2. sorting algorithms
3. Graph search algorithms
4. Dynamic programming
5. common problem solving patterns

---

## 3. Memory (RAM) vs. Storage

After turning off the computer, Storage is permanent, and memory will disappear.

int a = 1; => 32 bits for int

4 bytes = 32 bits = 1 integer

### a. Pointer

1. Int x = 4
2. Int \* pX = &x => "English: integer pointer named pX is set to address of x"
3. Int y = \*pX => integer named y is set to the thing is pointed to by pX
4. \* = pointer
5. & = address of

### b. Count

```
std::unordered_map<int, int> myMap = {{1, 10}, {2, 20}, {3, 30}};
```

```
int key = 2;
```

```
int count = myMap.count(key);
```

```
// Returns 1, because the key 2 exists in the map; return 0 is not exist
```

## 4. classes and objects

```
//class
class Robot {
    string name;
    string color;
    int weight;
//constructors
    Robot(String n, String c, int w){
        this.name = n;
        this.color = c;
        this.weight = w;
    }
//function
    void introduceSelf(){
        system.out.println("My name is " + this.name);
    }
}

//to use constructors and functions
Robot r1 = new robot("Leo", "Red", "37");
r1.introduceSelf();
```

---

## 5. Linked List

6 -> 3 -> 4 -> 2 -> 1 -> null

```
class Node{
    int data;
    Node next;
    Node prev;
    //constructor
    Node(int data){
        this.data = data;
    }
}

Node head = new Node(6);
Node nodeB = new Node(3);
Node nodeC = new Node(4);
...

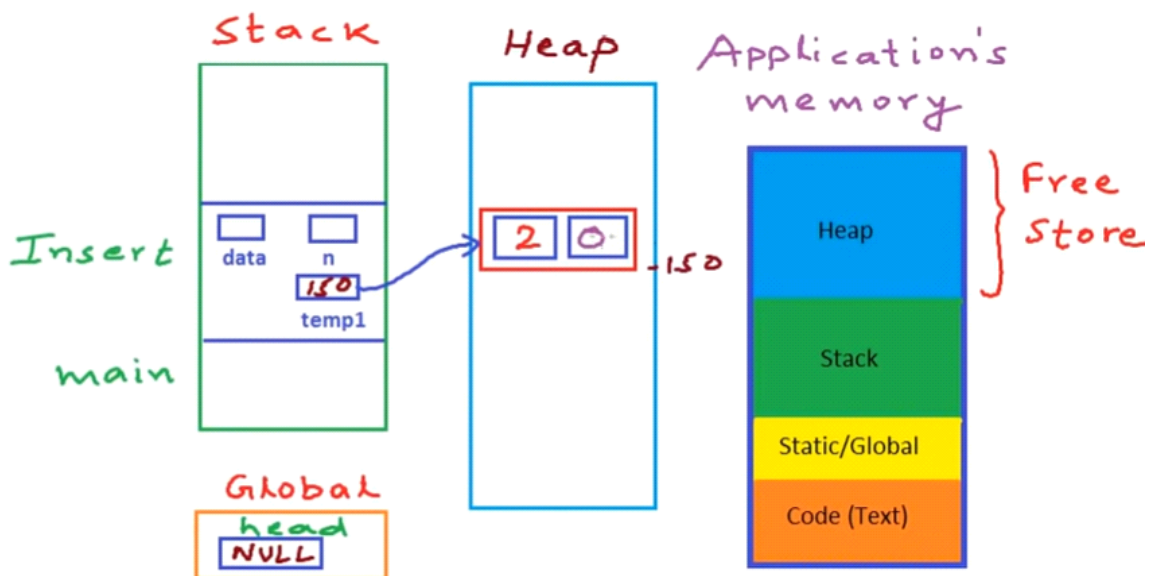
head.next = nodeB
nodeB.next = nodeC
...

//count how many nodes we have
int countNodes(Node head){
    //assuming head != null
    int count = 1;
    Node current = head;
    while(current.next != null){
        current = current.next;
        count += 1;
    }
}
```

```

#include <iostream>
struct Node {
    int data;
    Node* next;
};
Node* head;
void Insert(int x) {
    Node* temp = new Node();
    temp->data = x;
    temp->next = head;
    head = temp;
}
void Print() {
    Node* temp = head;
    std::cout << "List is: ";
    while (temp != NULL) {
        std::cout << " " << temp->data;
        temp = temp->next;
    }
    std::cout << "\n";
}
int main() {
    head = NULL;
    std::cout << "How many numbers?\n";
    int n, x;
    std::cin >> n;
    for (int i = 0; i < n; i++) {
        std::cout << "Enter the number\n";
        std::cin >> x;
        Insert(x);
        Print();
    }
    return 0;
}

```

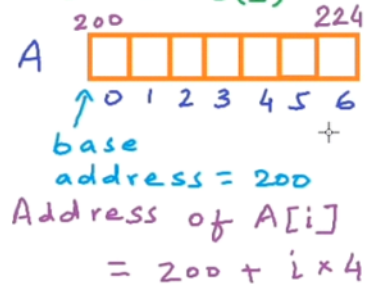


# Array vs Linned List

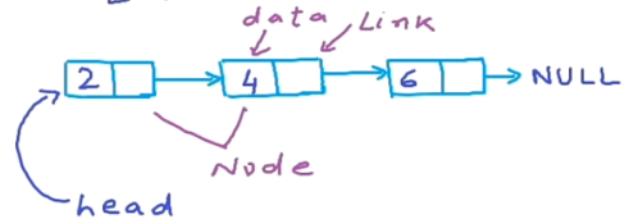
## Array

1) Cost of accessing an element

Constant time -  $O(1)$



## Linned List



Average case:  $O(n)$

Memory requirments

Array

Fixed size

Memory may not be available as one large blocks

Linked list

No unused memory

extra memory for pointer variable

Memory may be available as multiple small blocks

## 6. Recursion

**//factorial**

$n! = n \cdot (n-1)!$  if  $n \geq 1$   
1 otherwise (if  $n = 0$ )

Ex:  $4! = 4 \times 3 \times 2 \times 1 = 24$

```
int fact(int n){
    //assume that n is a positive integer or 0
    if (n >= 1) {
        return n * fact(n-1);
    } else {
        return 1;
    }
}
```

**// Fibonacci sequence**

Ex: 1, 1, 2, 3, 5, 8

```
int fib(int n){
    if (n >= 3){
        return fib(n-1) * fib(n-2);
    } else {
        return 1;
    }
}
```

**google interview problem**

**Recursive staircase problem**

$\text{num\_way}(N) = \text{num\_ways}(N-1) + \text{num\_ways}(N-2)$   
 $\text{num\_way}(0) = 1$

```
num_way(1) = 1
```

```
//python
```

```
//this won't work if n = 2
```

```
def num_ways(n):
```

```
    if n == 0 or n == 1:
```

```
        return 1
```

```
    else:
```

```
        return num_ways(n-1) + num_ways(n-2)
```

```
//fixed
```

```
def num_ways_bottom_up(n):
```

```
    if n == 0 or n == 1:
```

```
        return 1
```

```
    nums = [0] * (n + 1)
```

```
    nums[0] = 1
```

```
    nums[1] = 1
```

```
    for i in range(2, n + 1):
```

```
        nums[i] = nums[i - 1] + nums[i - 2]
```

```
    return nums[n]
```

```
# Example usage
```

```
total_feet = 11
```

```
print(f"Number of ways to jump {total_feet} feet:
```

```
{num_ways_bottom_up(total_feet)}")
```

```
//variation problems
```

```
X = {1,3,5}
```

```
//wrong
```

```
def num_ways_X(n):
```

```
    if n == 0
```

```
        return 1
```

```
    total = 0;
```

```
    for each i in {1,3,5}:
```

```
        if n - i >= 0:
```

```
            total += num_ways_X(n-i)
```

```
    return total
```

```
//fixed
```

```
def num_ways_bottom_up_X(n):
```

```
    if n == 0:
```

```
        return 1
```

```
    nums = [0] * (n + 1)
```

```
    nums[0] = 1
```

```
    for i in range(1, n + 1):
```

```
        total = 0
```

```
        for j in {1, 3, 5}:
```

```
            if i - j >= 0:
```

```
                total += nums[i - j]
```

```
        nums[i] = total
```

```
    return nums[n]
```

---

## 7. Big O notation

**runtime** = time it takes to execute a piece of code

linear time =  $O(n)$   
 constant time =  $O(1)$   
 quadratic time =  $O(n^2)$

Sorting =  $O(n \log n)$   
 Hash Table =  $O(n)$

1. find the fastest growing term
2. take out the coefficient

$T = an + b = O(n)$

$T = cn^2 + dn + e = O(n^2)$

given\_array = [1, 4, 3, 2, ..., 10]

```
def stupid_function(given_array):
    total = 0 -> O(1)
    return total -> O(1)
```

$T = O(1) + O(1) = c_1 + c_2$   
 $= c_3 = c_3 \times 1 = O(1)$   
 $O(1) + O(1) = O(1)$

```
def find_sum(given_array):
    total = 0 -> O(1)
    for each i in given_array:
        total += i -> O(1)
    return total -> O(1)
```

$T_2 = O(1) + n \times O(1) + O(1)$   
 $= c_4 + n \times c_5 = O(n)$

array\_2d = [[1, 4, 3], [3, 1, 9], [0, 5, 2]]

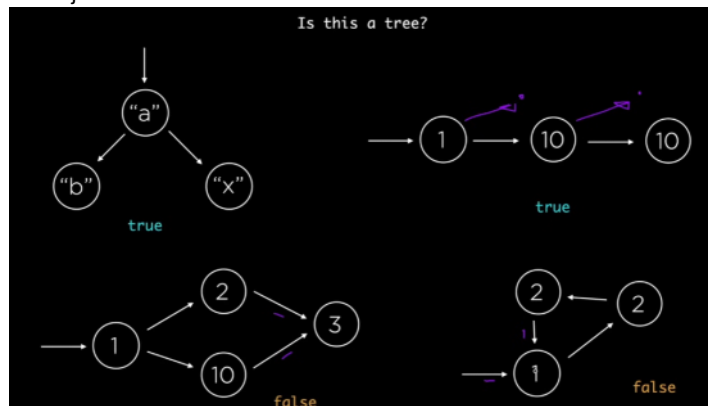
$T_3 = O(1) + n^2 \times O(1) + O(1)$   
 $= c_6 + n^2 \times c_7 = O(n^2)$

```
def find_sum_2d(array_2d):
    total = 0 -> O(1)
    for each row in array_2d:
        for each i in row:
            total += i -> O(1)
    return total -> O(1)
```

$T_4 = O(2n^2) = O(n^2)$   
 $T_4 = 2n^2 \times c + \dots = 2n^2 \times c + c_2n + c_3$   
 $= (2c) \times n^2 + c_2n + c_3 = O(n^2)$

## 8. Tree =>

```
Class Node{
    Int data
    Node left
    Node right
}
```



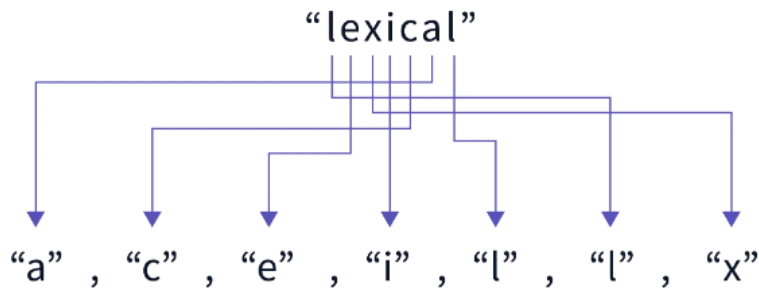
## Binary Tree =>

is a data structure in which each node has at most two children, referred to as the left child and the right child.

---

## 9. Sorting

Sort the string lexicographically



**a. Linear search** =  $O(n)$  =  $O(n/2)$  = It works by checking each element in the list, one by one, until the desired element is found or the end of the list is reached

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i # Return the index of the target
    element
    return -1 # Return -1 if the target is not found

# Example usage
arr = [34, 17, 23, 35, 45, 9, 1]
target = 23
result = linear_search(arr, target)

if result != -1:
    print(f"Element found at index {result}")
else:
    print("Element not found")
```

### b. Binary search:

在一排價格已經排序好的商品中，找到所有小於某價錢的商品數量

From <<https://medium.com/@lindingchi/binary-search-%E9%82%A3%E4%BA%9B%E8%97%8F%E5%9C%A8%E7%B4%B0%E7%AF%80%E8%A3%A1%E7%9A%84%E9%AD%94%E9%AC%BC-%E4%BA%8C-%E6%89%BE%E4%B8%8D%E5%88%B0%E6%80%8E%E9%BA%BC%E8%BE%A6-a308f5b9ce12>>

Arr[-50 ~ 50]

Target = 20

40 -> 20 -> 10 -> ... -> 1

$N \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \dots \rightarrow 1$

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \frac{n}{2^3} \rightarrow \dots \rightarrow \frac{n}{2^x} \approx 1$

$$\frac{n}{2^x} \approx 1 \Rightarrow n = 2^x \Rightarrow \log_2(n) = x = O(\log(n))$$

```
def search(arr, target):
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2 # make sure to round it
        down
        if arr[mid] == target:
            return mid
        elif target < arr[mid]:
            right = mid - 1
        else:
            left = mid + 1
    return -1
arr1 = [-2, 3, 4, 7, 8, 9, 11, 13]
assert search(arr1, 11) == 6
assert search(arr1, 13) == 7
assert search(arr1, -2) == 0
assert search(arr1, 8) == 4
assert search(arr1, 6) == -1
assert search(arr1, 14) == -1
assert search(arr1, -4) == -1
arr2 = [3]
assert search(arr2, 6) == -1
assert search(arr2, 2) == -1
assert search(arr2, 3) == 0
print("If you didn't get an assertion error, this program
has run successfully.")
```

### c. Quicksort:

is typically the fastest for large datasets due to its low overhead and excellent average-case performance.

Implement details:

1. Choosing pivot
  - Random element
  - Median of three
2. Dealing with duplicates
  - 3-way quicksort

```
Def qs(arr, l, r):
    If l >= r:
        Return
    P = partition(arr,l,r)
    Qs(arr, l, p-1)
    Qs(arr, p +1, r)
```





Undo mechanisms in text editors.

Function call management in programming languages (call stack).

**Depth-first search (DFS) algorithms.**

```
# Implementing a stack using a list
stack = []
# Push operation
stack.append(1)
stack.append(2)
stack.append(3)
# Pop operation
print(stack.pop()) # Output: 3
print(stack.pop()) # Output: 2
# Peek operation
print(stack[-1])   # Output: 1

-----

-----

#include <iostream>
#include <vector>
class Stack {
private:
    std::vector<int> items;
public:
    Stack() {}
    bool isEmpty() {
        return items.empty();
    }
    void push(int item) {
        items.push_back(item);
    }
    int pop() {
        if (!isEmpty()) {
            int topItem = items.back();
            items.pop_back();
            return topItem;
        } else {
            throw std::out_of_range("Stack is empty");
        }
    }
    int top() {
        if (!isEmpty()) {
            return items.back();
        } else {
            throw std::out_of_range("Stack is empty");
        }
    }
};

int main() {
    Stack stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    std::cout << "Top element: " << stack.top() << std::endl; //
Output: 3
    std::cout << "Popped element: " << stack.pop() << std::endl;
// Output: 3
    std::cout << "Top element: " << stack.top() << std::endl; //
Output: 2
```

```

std::cout << "Is stack empty: " << std::boolalpha
<< stack.isEmpty() << std::endl; // Output: false
return 0;
}

```

**Queues** = first in, first out

### 1. Operations:

**Enqueue:** Add an element to the end of the queue.

**Dequeue:** Remove the element from the front of the queue.

**Front/Peek:** Look at the front element without removing it

### 2. Usages

Order processing systems.

**Breadth-first search (BFS) algorithms.**

Print job management.

```

from collections import deque
# Implementing a queue using deque from collections
queue = deque()
# Enqueue operation
queue.append(1)
queue.append(2)
queue.append(3)
# Dequeue operation
print(queue.popleft()) # Output: 1
print(queue.popleft()) # Output: 2
# Peek operation
print(queue[0])        # Output: 3

```

## 11. Hash Tables and Dictionaries

is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Fast to compute

Avoid collision

**Double hashing** is an advanced technique used to handle collisions in hash tables

*double-hashing*

Bob 8		Jane 35		Paul 29	Alex 18	Chloe 88	
----------	--	------------	--	------------	------------	-------------	--

$i = h_1(\text{key}) \bmod 8$   
 $(i + c) \bmod 8$   
 $(i + 2c) \bmod 8$

$\gcd(c, m) = 1$   
 $c \leftarrow \{h_2(\text{key}) \bmod (m - 1)\} + 1$   
 $h_2(\text{key}) = h_1(\text{key} + 'd')$   
 $h_2(\text{key}) = h_1(\text{key})$

*search* .

$$\frac{1}{1 - \alpha}$$

*insertion*

$$\alpha = \frac{n}{m}$$

$[0, m-2]$   
 $[1, m-1]$

Suppose we want to insert the keys 10, 22, 31, 44, 59 into the hash table:

1. Insert 10:  $h_1(10) = 10 \bmod 11 = 10$   
No collision, so 10 is placed at index 10.
2. Insert 22:  $h_1(22) = 22 \bmod 11 = 0$   
No collision, so 22 is placed at index 0.
3. Insert 31:  $h_1(31) = 31 \bmod 11 = 9$

- No collision, so 31 is placed at index 9.
4. Insert 44:  $h_1(44) = 44 \bmod 11 = 0$ ,  $h_1(44) = 44 \bmod 11 = 0$   
Collision occurs at index 0.  
Calculate the step size using  $h_2(44) = 1 + (44 \bmod 10) = 5$ ,  $h_2(44) = 1 + (44 \bmod 10) = 5$ .  
Probe sequence:  $(0 + 1 \cdot 5) \bmod 11 = 5$ ,  $(0 + 1 \cdot 5) \bmod 11 = 5$ .  
No collision at index 5, so 44 is placed at index 5.
  5. Insert 59:  $h_1(59) = 59 \bmod 11 = 4$ ,  $h_1(59) = 59 \bmod 11 = 4$   
No collision, so 59 is placed at index 4.

### Advantages of Double Hashing

1. Reduced Clustering: It minimizes both primary and secondary clustering compared to linear and quadratic probing.
2. Efficiency: Provides good performance in terms of average search, insert, and delete operations.

### Dictionary

- Create a dictionary by using empty brackets {}.
- `dictionary_name[key] = value` adds items to a dictionary.
- `len(dictionary_name)` returns the number of keys in a dictionary.
- `print(dictionary_name[key])` prints the value associated with the key.
- Keys in a dictionary are unique. When a key is added multiple times, the old value is overwritten by the new one.

---

## 12. Unordered Map

### NOTES:

0. `std::unordered_map` is an associative container that contains key-value pairs with unique keys.
1. Search, insertion, and removal have average constant-time complexity.
2. Internally, the elements are organized into buckets.
3. It uses hashing to insert elements into buckets.
4. This allows fast access to individual elements, because after computing the hash of the value it refers to the exact bucket the element is placed into.

### WHY UNORDERED MAP

maintain a collection of unique (key:value) pairs with fast insertion and removal.

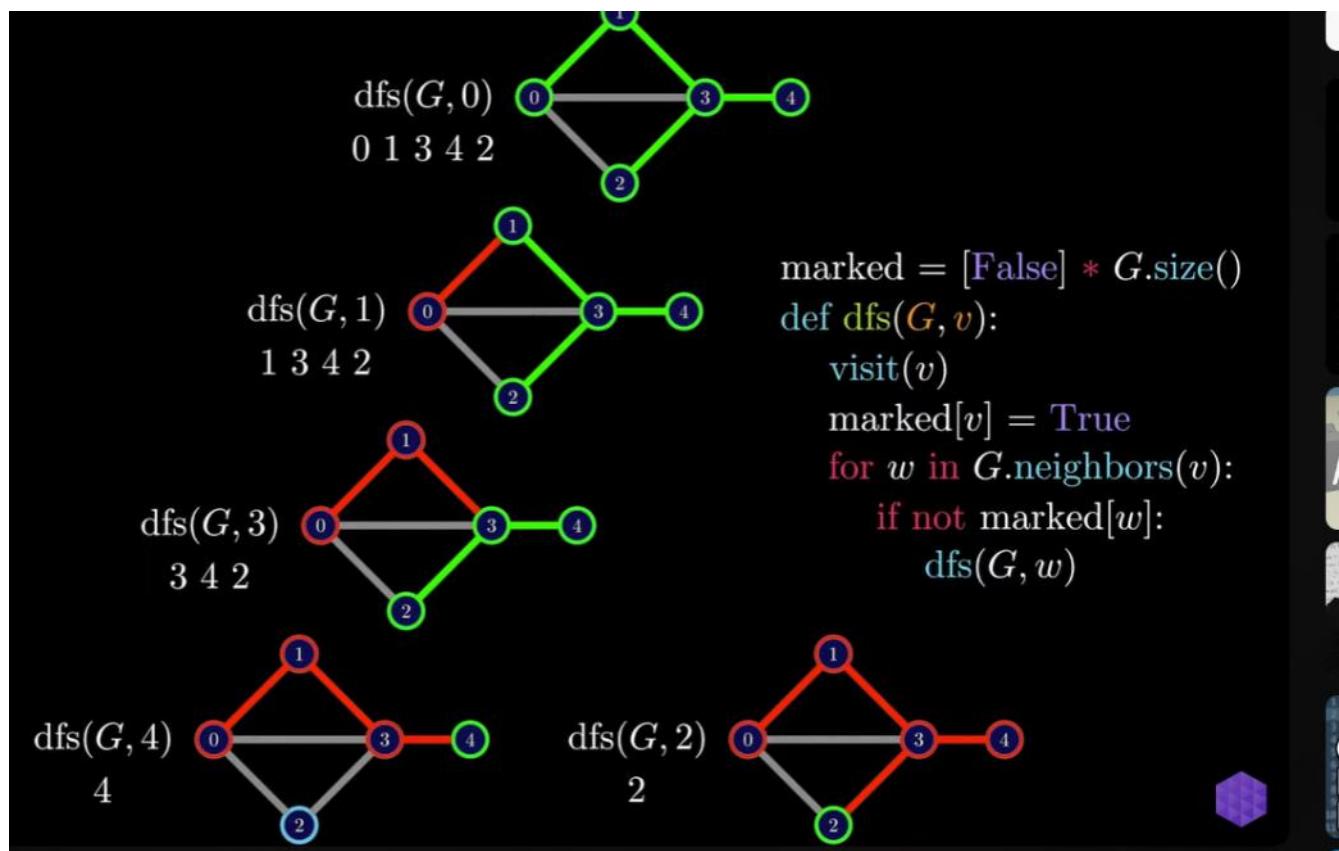
```

14 // PROGRAM:
15 #include <iostream>
16 #include <unordered_map>
17 using namespace std;
18
19 int main()
20 {
21     std::unordered_map<int, char> umap = {{1, 'a'}, {2, 'b'}};
22     // Access
23     cout << umap[1] << endl;
24     cout << umap[2] << endl;
25
26     // Update
27     umap[1] = 'c';
28
29     // Iterate
30     for(auto& elm: umap) {
31         std::cout << elm.first << " " << elm.second << " ";
32     }
33     cout << endl;
34
35     // Find
36     auto result = umap.find(2);
37     if (result != umap.end()) {
38         std::cout << "Found " << result->first << " " << result->second << '\n';
39     } else {
40         std::cout << "Not found\n";
41     }
42
43     return 0;
44 }

```

Handwritten annotations on the code: Yellow arrows point to the initialization of `umap`, the access of `umap[1]` and `umap[2]`, the update of `umap[1]`, the iteration over `umap`, and the `find` operation. To the right of the code, there are handwritten yellow circles and arrows, possibly representing a sequence of operations or a flow diagram.

### 13.Depth First Search( DFS)



# DFS Implementation Comparison

Both run in  $O(V + E)$

Cleaner and easier to read

```
marked = [False] * G.size()
def dfs(G, v):
    visit(v)
    marked[v] = True
    for w in G.neighbors(v):
        if not marked[w]:
            dfs(G, w)
```

More generalizable

```
marked = [False] * G.size()
def dfs_iter(G, v):
    stack = [v]
    while len(stack) > 0:
        v = stack.pop()
        if not marked[v]:
            visit(v)
            marked[v] = True
            for w in G.neighbors(v):
                if not marked[w]:
                    stack.append(w)
```

## Preorder vs Postorder

```
marked = [False] * G.size()
def dfs_pre(G, v):
    visit(v)
    marked[v] = True
    for w in G.neighbors(v):
        if not marked[w]:
            dfs(G, w)
```

```
marked = [False] * G.size()
def dfs_post(G, v):
    marked[v] = True
    for w in G.neighbors(v):
        if not marked[w]:
            dfs(G, w)
    visit(v)
```

### Recursive

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
// Function to visit a node (you can modify this as per your
requirements)
void visit(int node) {
    cout << node << " ";
}
// Function to perform DFS
```

```

void dfs(vector<vector<int>>& graph, int start, vector<bool>&
marked) {
    // Mark the starting node as visited
    marked[start] = true;

    // Visit the starting node
    visit(start);

    // Visit all the unvisited neighbors of the starting node
    for (int neighbor : graph[start]) {
        if (!marked[neighbor]) {
            dfs(graph, neighbor, marked);
        }
    }
}

int main() {
    // Create a sample graph
    vector<vector<int>> graph = {
        {1, 2},
        {0, 2, 3},
        {0, 1, 3},
        {1, 2}
    };

    // Perform DFS starting from node 0
    vector<bool> marked(graph.size(), false);
    dfs(graph, 0, marked);

    return 0;
}

```

---

## 14. Breath First search

BFS starts at arbitrary Node of a graph and explore its neighbor first before moving to its next level of neighbors

```

#include <iostream>
#include <vector>
#include <queue>
using namespace std;
// Function to perform BFS
void bfs(vector<vector<int>>& graph, int start, vector<bool>&
visited) {
    // Create a queue to keep track of the nodes to visit
    queue<int> q;

    // Mark the starting node as visited and push it onto the queue
    visited[start] = true;
    q.push(start);

    // Perform BFS
    while (!q.empty()) {
        // Dequeue the front node from the queue
        int current = q.front();
        q.pop();

        // Print the current node
        cout << current << " ";
    }
}

```

```

        // Visit all the unvisited neighbors of the current node
        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    // Create a larger graph
    vector<vector<int>> graph = {
        {1, 4},          // Vertex 0 is connected to vertices 1 and 4
        {0, 2, 3, 4},    // Vertex 1 is connected to vertices 0, 2, 3,
and 4
        {1, 3, 5},      // Vertex 2 is connected to vertices 1, 3, and
5
        {1, 2, 5, 6},    // Vertex 3 is connected to vertices 1, 2, 5,
and 6
        {0, 1, 5},       // Vertex 4 is connected to vertices 0, 1, and
5
        {2, 3, 4, 6, 7}, // Vertex 5 is connected to vertices 2, 3,
4, 6, and 7
        {3, 5, 7},       // Vertex 6 is connected to vertices 3, 5, and
7
        {5, 6}           // Vertex 7 is connected to vertices 5 and 6
    };

    // Perform BFS starting from node 0
    vector<bool> visited(graph.size(), false);
    bfs(graph, 0, visited);

    return 0;
}
//output = 0 1 4 2 3 5 6 7

```

**Resources:** [Resources for Learning Data Structures and Algorithms \(Data Structures & Algorithms # 8\)](#)