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**

----

4. classes and objects

```
//class
class Robot {
    string name;
    string color;
```

```java
    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 contructors and functions
Robot r1 = new robot("Leo", "Red", "37");
r1.introduceSelf();
```

----

5. Linked List

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

```java
class Node{
    int data;
    Node next;
    Node prev;
    //contructor
    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;
    }
}
```
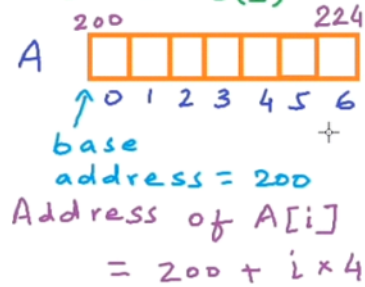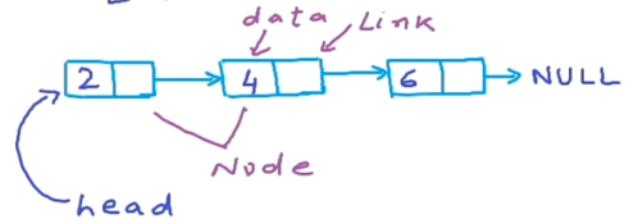
Array vs Linked List

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. (n-1)!   if n >= 1
   1   otherwise (if n = 0)
Ex: 4! = 4 x 3 x 2 x 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
num_way(N) = num_ways(N-1) + num_ways(N-2)
num_way(0) = 1

num_way(1) = 1

```python
//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

```python
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}

```python
//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
```python
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 to takes to execute a piece of code

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

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)
```

$T_2 = O(1) + n \times O(1) + O(1)$
$= c_4 + n \times c_5 = O(n)$
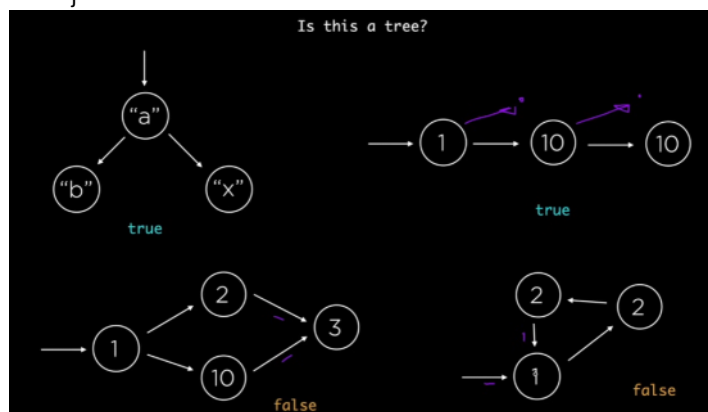
```
array_2d = [[1, 4, 3],       [[1, 4, 3, 1],
            [3, 1, 9],        [3, 1, 9, 4],
            [0, 5, 2]]        [0, 5, 2, 6],
                              [4, 5, 7, 8]]
```

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

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

$T_4 = O(2n^2) = O(n^2)$

$T_4 = 2n^2 \times c + ... = 2n^2 \times c + c_2 n + c_3$

$= (2c) \times n^2 + c_2 n + c_3 = O(n^2)$

```
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)
```

----

8. **Tree** =>
    Class Node{
        Int data
        Node left
        Node right
    }

Is this a tree?

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

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

```python
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:
Arr[-50 ~ 50]
Target = 20

40 -> 20 -> 10 -> … -> 1

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

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

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

```python
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.
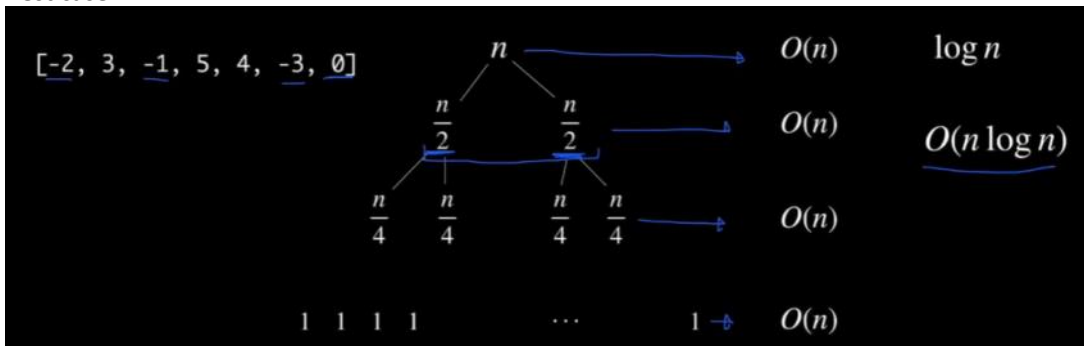
Implement details:
1. Choosing pivot
   Random element
   Median of three
2. Dealing with duplicates
   3-way quicksort

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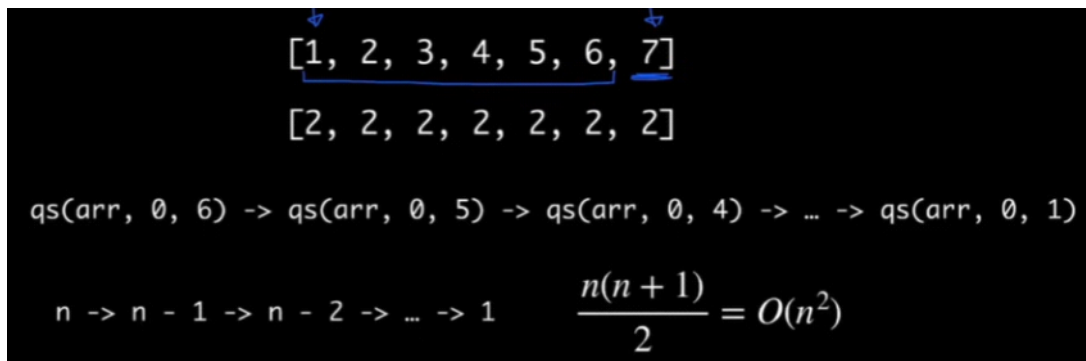


```
def partition(arr, l, r):
    pivot = arr[r]
    i = l - 1
    for j from l upto r - 1:
        if arr[j] < pivot:
            i += 1
            swap arr[i] and arr[j]
    swap arr[i + 1] and arr[r]
    return i + 1
```

Time Complexity:
Best case:

Worst case:

$$[1, 2, 3, 4, 5, 6, 7]$$

$$[2, 2, 2, 2, 2, 2, 2]$$

qs(arr, 0, 6) -> qs(arr, 0, 5) -> qs(arr, 0, 4) -> ... -> qs(arr, 0, 1)

n -> n - 1 -> n - 2 -> ... -> 1 $\frac{n(n+1)}{2} = O(n^2)$

d. other sorting:

    1. **Mergesort:** is a good choice when stability is required and for data that doesn't fit in memory (external sorting).

    2. **Heapsort:** provides reliable O(nlogn) performance with in-place sorting but is generally outperformed by quicksort and mergesort.

    3. **Timsort:** is highly efficient for real-world data and is used in many standard libraries due to its adaptive nature and stability.

----

10. **Stacks and queues**

Stacks = last in, first out.
1. **Operations:**
   Push: Add an element to the top of the stack.
   Pop: Remove the top element from the stack.
   Peek/Top: Look at the top element without removing it.
2. **Usage:**
   Undo mechanisms in text editors.
   Function call management in programming languages (call stack).
   Depth-first search (DFS) algorithms.

```python
# 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
```

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.

```python
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

search $\circ$

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

insertion

Bob 8		Jane 35		Paul 29	Alex 18	Chloe 88	

$i = h_1(\text{key}) \mod 8$

$(i + c) \mod 8$

$(i + 2c) \mod 8$

$\gcd(c, m) = 1$

$c \leftarrow \{h_2(\text{key}) \mod (m - 1)\} + 1$

$h_2(\text{key}) = h_1(\text{key} + \text{'d'})$

$h_2(\text{key}) = h_1(\text{key})$

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

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

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

1. Insert 10: $h1(10)=10\mod11=10$ $h1(10)=10\mod11=10$
   No collision, so 10 is placed at index 10.
2. Insert 22: $h1(22)=22\mod11=0$ $h1(22)=22\mod11=0$
   No collision, so 22 is placed at index 0.
3. Insert 31: $h1(31)=31\mod11=9$ $h1(31)=31\mod11=9$
   No collision, so 31 is placed at index 9.
4. Insert 44: $h1(44)=44\mod11=0$ $h1(44)=44\mod11=0$
   Collision occurs at index 0.
   Calculate the step size using $h2(44)=1+(44\mod10)=5$ $h2(44)=1+(44\mod10)=$
   5.
   Probe sequence: $(0+1\cdot5)\mod11=5$ $(0+1\cdot5)\mod11=5$.
   No collision at index 5, so 44 is placed at index 5.
5. Insert 59: $h1(59)=59\mod11=4$ $h1(59)=59\mod11=4$
   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.

Ditionary
- 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.

```cpp
// PROGRAM:
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    std::unordered_map<int,char> umap = {{1,'a'},{2,'b'}};
    // Access
    cout << umap[1] << endl;
    cout << umap[2] << endl;

    // Update
    umap[1] = 'c';

    // Iterate
    for(auto& elm: umap) {
        std::cout << elm.first << " " << elm.second << " ";
    }
    cout << endl;

    // Find
    auto result = umap.find(2);
    if (result != umap.end()) {
        std::cout << "Found " << result->first << " " << result->second << '\n';
    } else {
        std::cout << "Not found\n";
    }

    return 0;
}
```

Resources: Resources for Learning Data Structures and Algorithms (Data Structures & Algorithms # 8)