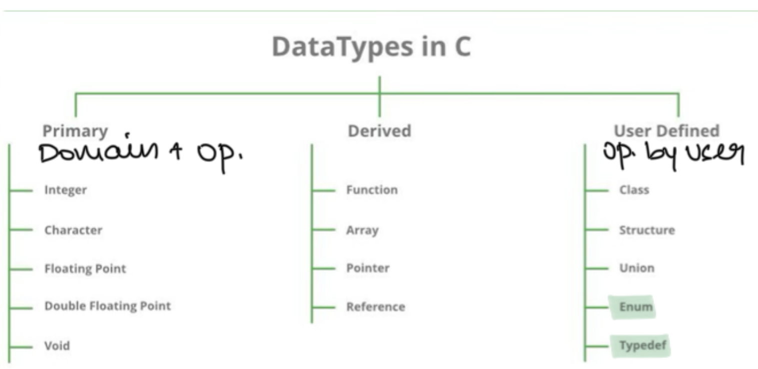
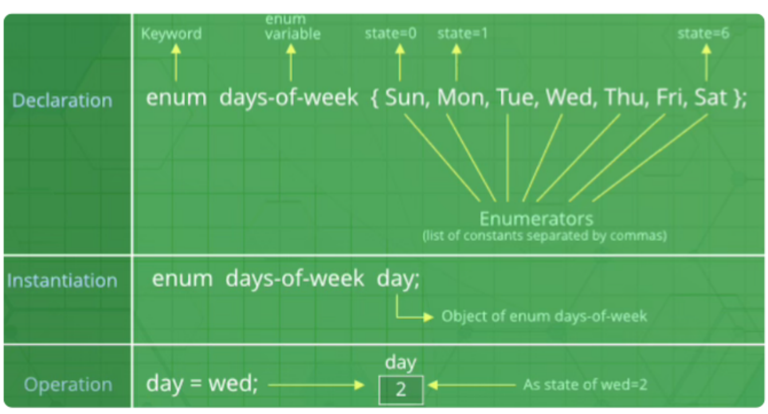
Data structures and algorithms

0. Recap:

1.1	Review of elementary data types and structures in C

Basic introduction to C programming and programs on the following topics:
- Largest of three numbers
- Palindrome
- Array reverse
- Structures
- Call by reference / call by value
- Local, global, static, extern → variable scopes.





Note:
typedef: keyword to provide existing data-type with new name

Syntax:
typedef cur_name new_name

enum (enumeration) assigns names to integral constants in C

Data structures: systematic way to organize data so that it can be used efficiently
↳ have many real life applications

1. Abstract Data types in C:

1.2	Abstract Data Types (ADT)-Basic concept of Data Structures-Performance measures for Data Structures

- User defines a data type along with the ops. it can hold.

- ADTs are like user defined data types that define operations on values using functions without specifying what is there inside the function and how the op are performed. (≈ Black box)

- Uses arrays/linked lists usually

- Advantages:
  - Implementation is done in the back end
  - Different implementations are possible
  - Abstraction and efficiency and reusability

Note: ADT tells is "what" is to be done while data structures explain "how" to do it
↓
There are diff. data structures for 1 ADT.

The program which uses data structure is called a client program

It has access to the ADT i.e. interface.

The program which implements the data structure is known as the implementation.

changes (like linked list to array) in implementation does not affect client.

2. Time and Space complexities: → Efficiency measure

1.3	Time and Space Complexity Asymptotic Measures - Big-Oh, Omega, Theta.

→ Approx.

(1.) Time complexity:

- Priori analysis: "estimation" before prog. execution
- Posteriori analysis: "calculation" after execution

- We are concerned with main memory space (RAM) and CPU computations → instruction exe. by CPU
  ↳ time complexity.

- Ways to calculate:
  1. Examine exact running time ✗
  2. Time complexity ∝ no. of inputs ⇒ f(n) → Growth rate
     ↳ Asymptotic complexity (approximation)

Big O notation is used to measure the performance of any algorithm by providing the order of growth of the function.

It gives the upper bound on a function by which we can make sure that the function will never grow faster than this upper bound.

We want the approximate runtime of the operations performed on data structures.

→ Basically tells how worst an alg. performs

We are not interested in the exact runtime.

Big O notation will help us to achieve the same.

Time complexity of an algorithm = Total no. of CPU comp. = sum of freq. count of each instruction

Frequency count = no. of times an instruction is executed.

(ii) Space complexity:

- Priori analysis: "estimation" before prog. execution
- Posteriori analysis: "calculation" after execution

Space Complexity of an Algorithm =
- Space required to store the source code
- \+ Space required for simple variables
- \+ Space for the data structures used
- \+ Space required for stack for recursive algorithms

constants (eliminate)

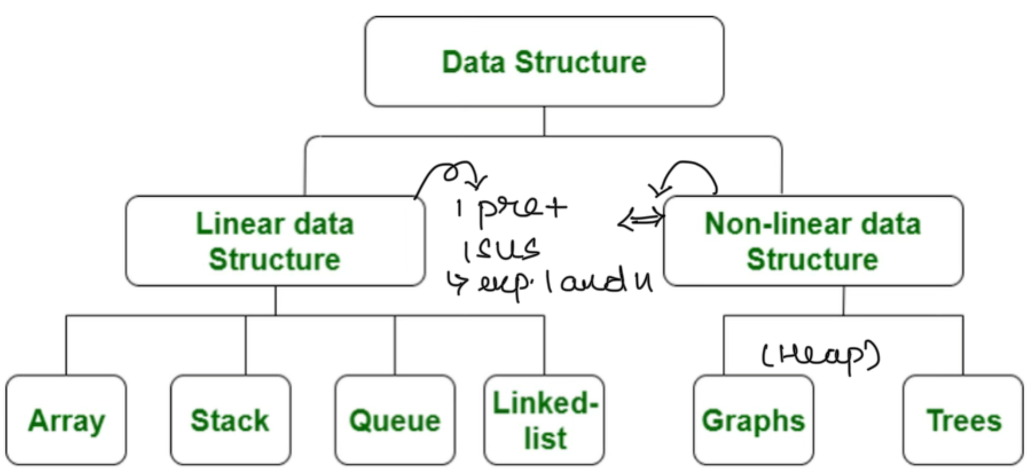
→ No. of ele.

→ No. of times a fn. is called

Note:

Big-Oh: Worst case; Theta: Average; Omega: Best

3. Linear vs. non linear data structures:



4. Static vs. Dynamic data structures:

STATIC DATA STRUCTURES

In these type of data structures, the memory is allocated at compile time. Therefore, maximum size is fixed.

Advantage: Fast access

Disadvantage: Slower insertion and deletion

Eg. Array

DYNAMIC DATA STRUCTURES

In these type of data structures, the memory is allocated at run time. Therefore, maximum size is flexible.

Advantage: Faster insertion and deletion

Disadvantage: Slower access

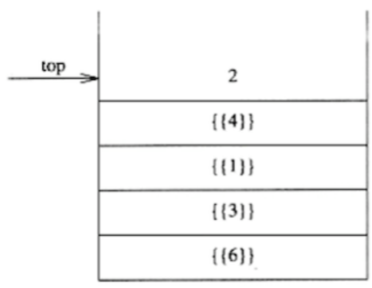
Eg. Linked List

5. Stack ADT:

⇝ the implementation does not matter much

3	STACKS AND QUEUES	
3.1	Stack ADT – Operations → Basic op.	1
3.2	Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls	1
3.3	Queue ADT – Operations – Circular Queue	1
3.4	DeQue – Applications of Queues – Scheduling	1

- Linear data type; uses LIFO algorithm; homogenous
- Example: undo operation, memory, redo operation.
- Note:



Initially, top is set to -1 (then incremented acc.)

- **Algorithms on stack:**

(i) isFull() - to check if stack is full

Algorithm isFull (S, top, size)

/* here, S represents the stack, top represents the location and size denotes the maximum capacity of stack. */

```
begin

if ( top == size-1)
 {
   return 1;
   //stack is full
 }
else
 {
   return 0;
   //stack is not full
 }
```

(ii) isEmpty() - to check if stack is empty

Algorithm isEmpty (S, top)

/* here, S represents the stack and top denotes the location of the topmost entry */

```
if (top == -1)
{
 return 1;
 //stack is empty
}
else
{
 return 0;
 //stack is not empty
}
```

(iii.) push() - to push an element into the stack

Algorithm push (S, x)

/* here S is the stack and x is the number that is to be pushed */

```
if (!(isFull())
{
  top++;
  S[top] = x;
  return 1;
  //pushed successfully
}
else
{
  return 0;
 //stack overflow
}
```

(iv.) pop() - to pop an item from the stack
or last entry

Algorithm pop(S, top)

/* here S denotes the stack and top the topmost entry */

```
if (!(isEmpty())
{
  x = S[top];
  top--;
```

```
    return x;
    // popped out successfully
  }
  else
  {
    return 0;
    // stack underflow
  }
```

(v) peep() - to simply view the topmost entry AKA top()

```
Algorithm peep(s, top)

/* here S denotes the stack and top denotes
   the topmost entry */

if (!(isEmpty())
{
  x = s[top];
  return x;
  // peeped successfully
}
else
  return 0;
  // stack is empty
```

size(): returns size of stack

Note: operator inbetween operands → infix
Eg. a+b

Postfix: ab+ ; prefix: +ab

- Write algorithms for basic stack operations (i) isFull() (ii) isEmpty(), Push(), Pop(), Peek(). Note: use isEmpty and isFull algorithm in last three algorithms.
- Write an algorithm to check palindrome.
- Write an algorithm to check for balancing symbols in the languages: C++ (/* */, (), [], {}). Explain how to print out an error message that is likely to reflect the probable cause.
- Write an algorithm to evaluate a postfix expression.
- Write an algorithm to convert an infix expression that includes (,), +, -, *, and / to postfix.
- Write algorithms to implement two stacks using only one array. Your stack algorithms should not declare an overflow unless every slot in the array is used.
- Show how to implement three stacks in one array.
- Given a string A denoting an expression. It contains the following operators '+', '-', '*', '/'. Check whether A has redundant braces or not. Return 1 if A has redundant braces, else return 0. Note: A will be always a valid expression. (Amazon)

For Example
Input 1: A = "((a + b))"
Output 1: 1
Explanation 1: ((a + b)) has redundant braces so answer will be 1.
Input 2: A = "(a + (a + b))"
Output 2: 0
Explanation 2: (a + (a + b)) doesn't have have any redundant braces so answer will be

0.

Linear Data Structures Assignments for answers

Parameter	Stack	Linear Queue	Circular Queue
Algorithm	LIFO	FIFO	FIFO
Pointer	Top	Front: delete and rear: insert	Front: delete and rear: insert
Initial	Top = -1	Front = 0 and Rear = 0	Front = 0 and Rear = 0
Operations	Push/pop	Enqueue/ dequeue	Enqueue/ dequeue
Full	Top = size - 1	Rear = size -1	(Rear+1)%size = Front OR count = size
Empty	Top = -1	Front = Rear	Front = Rear OR count = 0
Use	Memory, infix to postfix, undo and redo	Job scheduling~ CPU scheduling algorithms	Basically a better version of linear queue

6. Queues:

(i) Linear:

- Algorithm Enqueue (Q, x)

```
begin
if (Rear == size-1)
    print ("Queue is full")
else
    Q[Rear] = x
    Rear++;
end
```

- Algorithm Dequeue (Q)

```
begin
if (Rear == Front)
    print ("Queue is empty");
else
    x = Q[Front]
    Front++;
    return x
end
```

(ii) Circular / Non-linear:

- Algorithm Enqueue (Q, x)

```
begin
count = 0, size;        // or not :)
if (count == size)      // or if ((rear+1)% size) == front)
    print ("Queue is full")
else
    Q[rear] = x;
    rear = (rear+1)% size;
    count ++;
```

end

- Algorithm Dequeue (Q)

begin

count, size;

```
if (count == 0)    // or if (front == rear)
    print ("Queue is empty")

else
    Q[front] = x;
    front = (front+1) % size;
    count --;

end
```

Note: Here, there are two ways to determine Full/Empty

S1:

size-1 → one block is not used

Full: (rear+1) % size == front

Empty: front == rear

S2:

Count variable is used

Full: count == size

Empty: count == 0

space complexity is pretty similar.

→ S1 is better in terms of time complexity.

Applications of Stacks and Queues

1. **Longest Valid Parentheses:** Given a string containing just the characters '(' and ')', return the length of the longest valid (well-formed) parentheses substring

   (:
   isEmpty: push
   else
   pop

) :
   !isEmpty: pop
   if pop == (→ c++

   a. Example 1:

      Input: s = "(()"

      Output: 2

      Explanation: The longest valid parentheses substring is "()".

   b. Example 2:

      Input: s = ")()())"

      Output: 4

      Explanation: The longest valid parentheses substring is "()()".

2. **Sliding Window Maximum:** You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

   a. Example 1:

      Input: nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3

      Output: [3, 3, 5, 5, 6, 7]

      Explanation:

      ```
      Window position                Max
      ---------------               -----
      [1  3  -1] -3  5  3  6  7       3
       1 [3  -1  -3] 5  3  6  7       3
       1  3 [-1  -3  5] 3  6  7       5
       1  3  -1 [-3  5  3] 6  7       5
       1  3  -1  -3 [5  3  6] 7       6
       1  3  -1  -3  5 [3  6  7]      7
      ```

3. **Implement a Queue Using Stacks:** Implement a queue using two stacks to perform enqueue (push) and dequeue (pop) operations. Analyze the time complexity of the operations.
4. **Implement a Stack Using Queues:** Implement a queue using two queues to perform push (enqueuer) and pop (dequeue) operations. Analyze the time complexity of the operations.
5. **Minimum Stack:** Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.Design a stack that supports the usual push, pop, and peek operations but also has a method that returns the minimum element in O(1) time. All operations should be performed in O(1) time complexity. (NOTE: O(N) space complexity)
6. **Sort a Stack:** Given a stack, the task is to sort it such that the top of the stack has the greatest element. (NOTE: O(N) space complexity)

Linear Data Structures Assignments for answers

Basic Operations of Singly Linked List

1. Create a SLL with N nodes
2. Insert x
    a. Beginning
    b. End
    c. At position
    d. Before element y
    e. After element y
3. Delete
    a. Beginning
    b. End
    c. At position
    d. Before element y
    e. After element y
4. Search
    a. By Data
    b. By Position
5. Display
    a. Forward Order
    b. Reverse Order

Linear Data Structures Assignments for answers