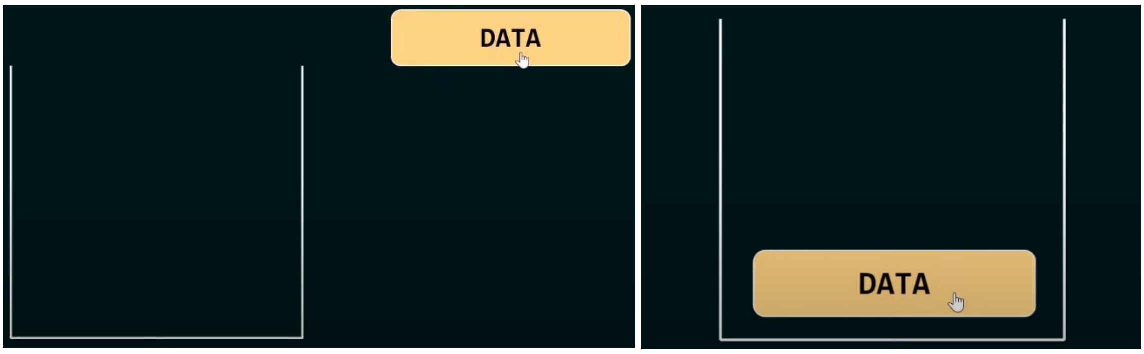
Stack (FILO)

Insertion / **push(data)** dan deletion / **pop()** hanya boleh di akhir suatu list. Hanya 1 pintu.

Di stack ada istilah: **push(data)** , **pop()**, **top / peek()**, **size()**, **isEmpty()**, **isFull()**

a. **push(data)**

.: Insert data into stack

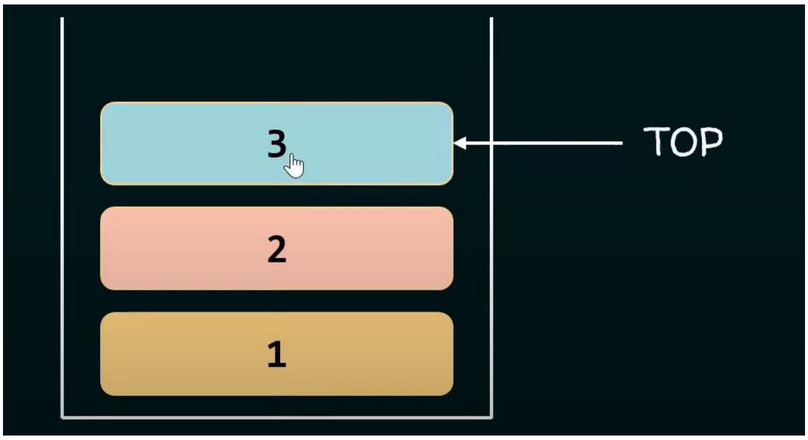


b. **pop()**

.: Delete the last data from the stack

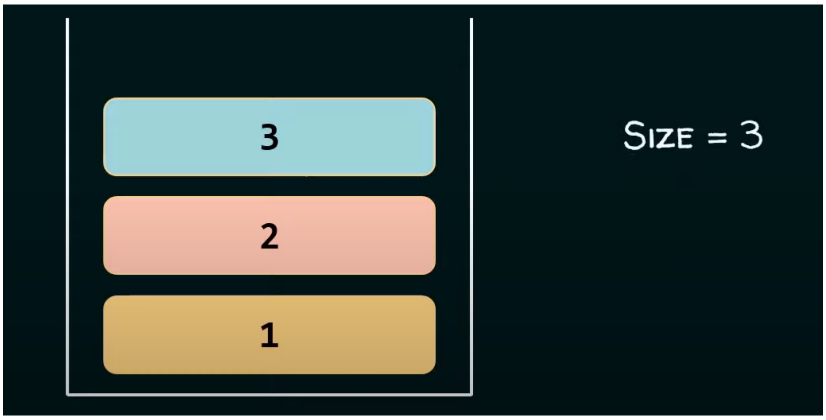
c. **top / peek()**

.: Return the value of the last data from the stack without removing it



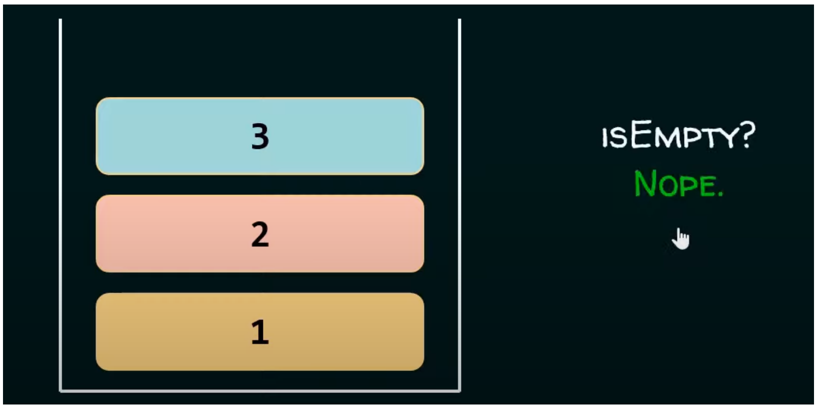
d. **size()**

.: Return the size or the number of how much data in the stack



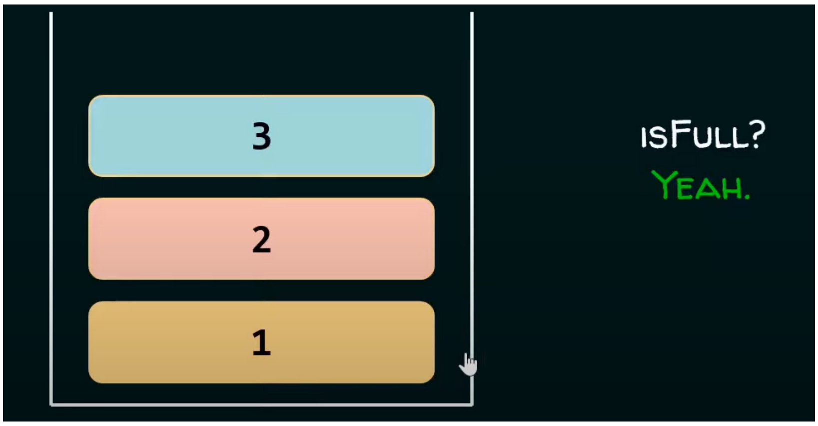
e. **isEmpty()**

.: Return TRUE if the stack is **empty**, else return FALSE

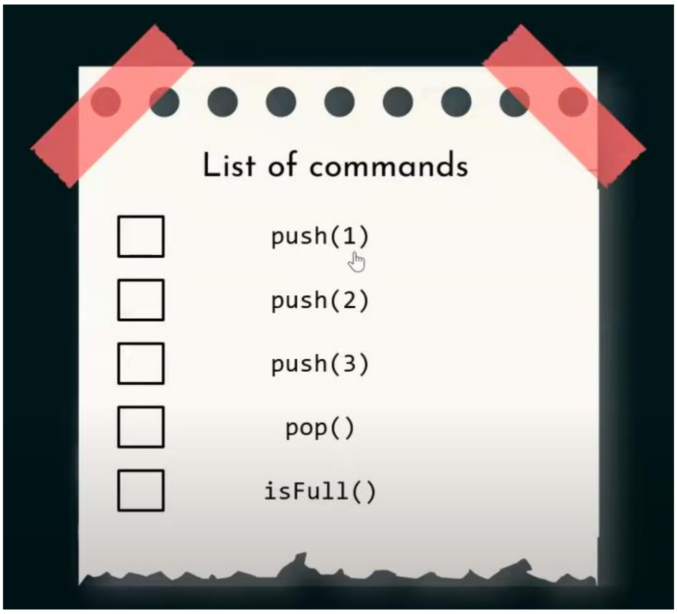


f. **isFull()**

.: Return TRUE if the stack is **full**, else return FALSE



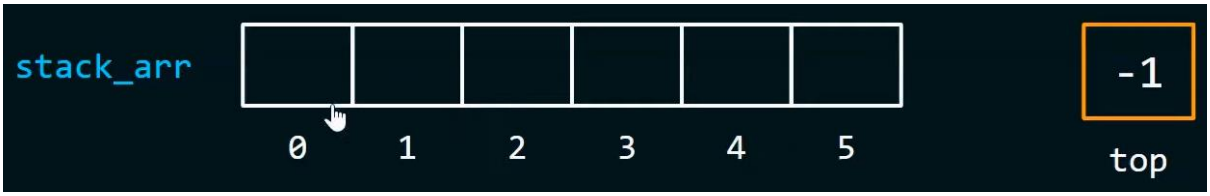
Mini Exercise:



Array Representation

.: Sebelum masuk, **Stack** bisa dilakukan dengan pendekatan **array** dan **linked list**, kita fokus ke array.

.: Di dalam array kita bisa **insert** dan **delete** dari **index manapun**, tapi kita ingin membuat array ini **"behave" seperti stack**, maka kita membutuhkan pointer "**TOP**" untuk **tracking data terakhir**

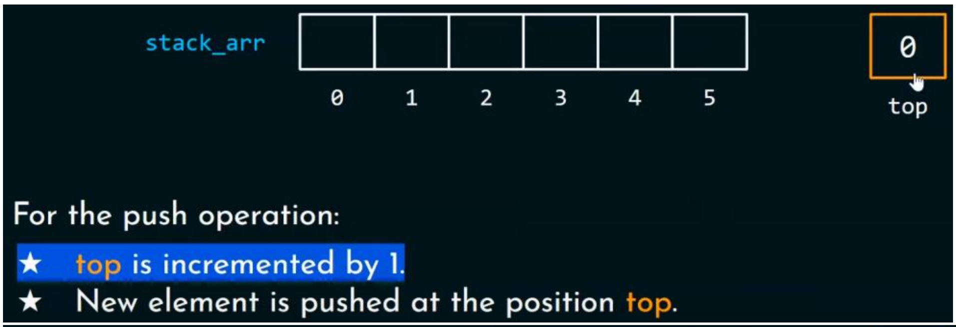


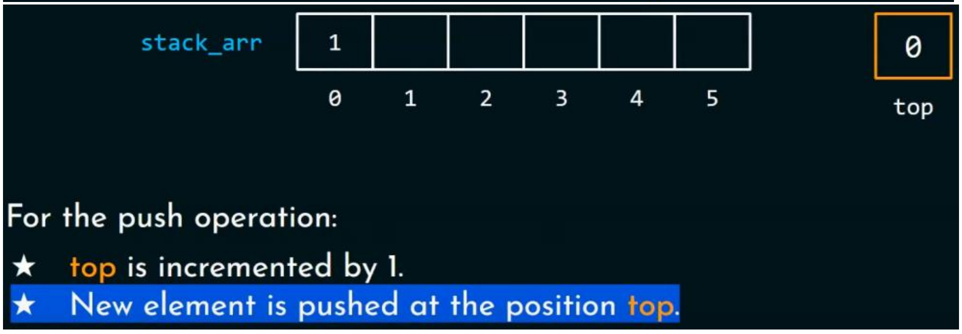
.: TOP = -1 untuk mengindikasikan bahwa list kita kosong dan belum terisi

.: Jika, TOP = 0, artinya TOP berada di index 0 dari stack_arr dan list kita sudah berisi 1 data

.: Dipilih nilai TOP = -1 karena ingin make sure jika list kosong, TOP tidak menunjuk index manapun karena nilai -1 tidak valid pada suatu array

Cara kerja jika, push(1):



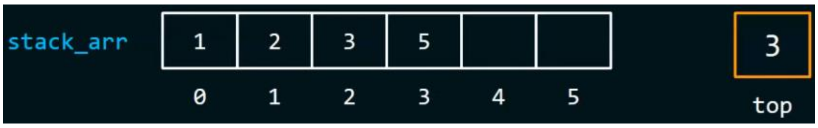


Dengan 2 rules tersebut, lalu kita lanjutkan dengan:

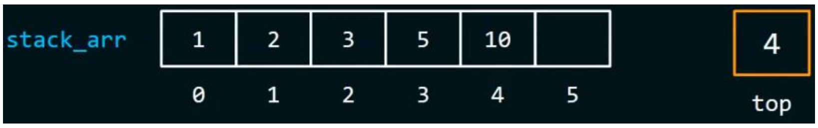
push(2)

push(3)

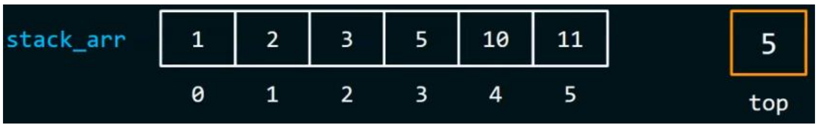
push(5)



push(10)



push(11)

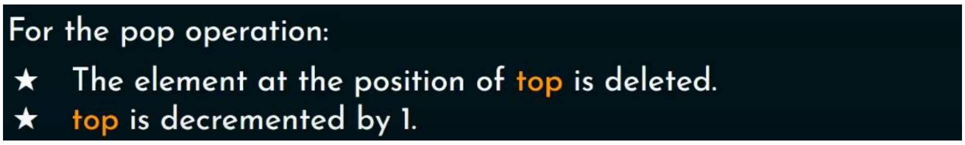


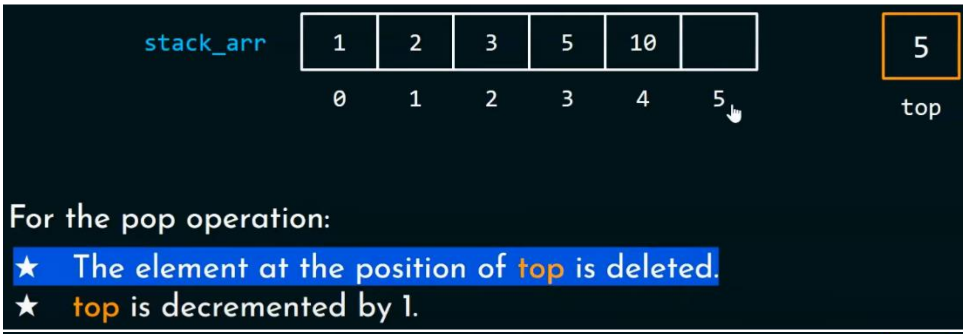
Jika kita **push(12)**, bisa?
Ga bisa, karena space array nya ga cukup, **state** ini namanya “OVERFLOW”

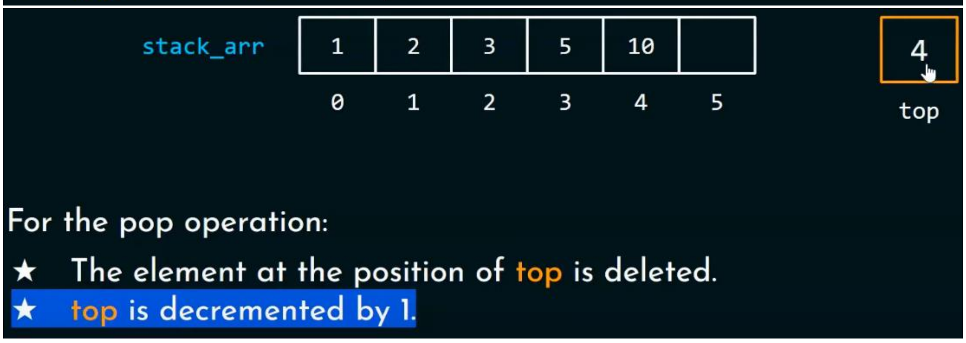
.: Overflow = Array nya udah penuh tapi masih mau di push, jadinya gabisa

Selanjutnya kita akan melakukan **pop()**

Cara kerja jika, pop():







Jika kita **pop()** sampai habis di ujung array, **state** ini namanya “UNDERFLOW”

.: Underflow = List of Array nya udah kosong tapi masih mau di pop, jadinya gabisa

a. Kodingan **push(data)** seperti ini:


```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int stack_arr[MAX];
int top = -1;

void push(int data)
{
    if(top == MAX - 1)
    {
        printf("Stack Overflow\n");
        return; //indicates the end of the function
    }
    top = top+1;
    stack_arr[top] = data;
}

int main()
{
    push(1);
    push(2);
    push(3);
    push(4);
    push(5);
    return 0;
}
```

stack_arr[MAX] dan **top** dijadikan global variable karena akan diakses terus menerus di tiap function, agar tidak selalu di declare. Kita juga bisa menggunakan #define directive.

Untuk handling **overflow**

```c
void print()
{
    int i;
    if(top == -1)
    {
        printf("Stack underflow\n");
        return;
    }
    for(i=top; i>=0; i--)
        printf("%d ", stack_arr[i]);
    printf("\n");
}
```

b. Kodingan **pop()** seperti ini:

.: Kita tidak bisa menghapus data pada array dengan mudah. Dengan begitu, bisa **diakali** dengan **mengurangi (-1) nilai dari TOP**


```c
int pop()
{
    int value;
    if(top == -1)
    {
        printf("Stack underflow\n");
        exit(1);
    }
    value = stack_arr[top];
    top = top - 1;
    return value;
}
```

Variable value dibuat untuk menyimpan nilai dari data yang di delete agar nantinya bisa di proses.

```c
int main()
{
    int data;
    push(1);
    push(2);
    push(3);
    push(4);
    data = pop();
    data = pop();
    print();
    return 0;
}
```

```c
int main()
{
    int data;
    push(1);
    push(2);
    push(3);
    push(4);
    data = pop();
    printf("%d", data);
    return 0;
}
```

Return value dari pop disimpan ke dalam varaible data a.k.a print data yang di delete.

Itu sebabnya kita membuat variable value untuk menyimpan nilai dari data yang di delete yang kemudian akan di return.

c. Kodingan **top / peek()** dan **size()** seperti ini:

```
int peek(){
   if (top == -1)
    {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack_arr[top];
}

int size(){
    if (top == -1)
    {
        printf("Stack underflow\n");
        exit(1);
    }
    return top+1;
}

int main() {
    push(1);
    push(2);
    push(5);
    push(9);
    printf("%d \n", size());
    printf("%d", peek());
```

d. Kodingan **isFull()** dan **isEmpty()** seperti ini:

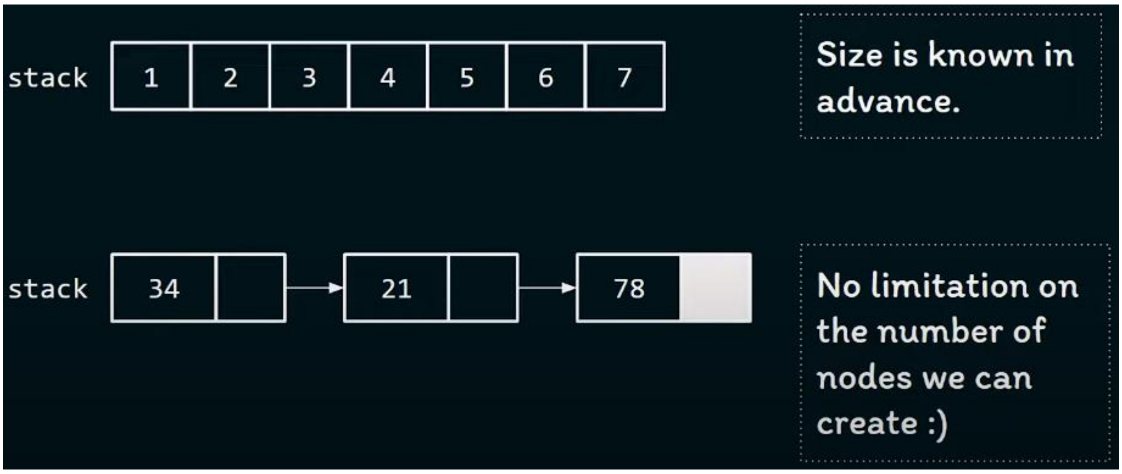
```
int isFull()
{
    if(top == MAX - 1)
        return 1; //indicates the end of the function
    else
        return 0;
}

int isEmpty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}
```

Dengan adanya kedua function tersebut, maka validasi di tiap function lain dapat menggunakan kedua function tersebut.

SOAL ASSIGNMENT 3: [Array Implementation of Stacks (Part 4) (youtube.com)]

Linked List Representation

Kapan kita harus menggunakan linked list dibanding array dalam membuat stack? Ketika ukuran dari stack tidak diketahui.



- TOP pada linked list ditaruh pada head / di depan list. Kenapa? Karena mudah untuk insert dan delete tanpa harus traversing.

a. Fungsi **push(data)** dan **pop()** menggunakan **add_beg()** dan **delete_beg()**:

   Meski menggunakan linked list tidak perlu mengetahui size dari stack, tapi kadang kala **malloc()** function akan **return NULL** ketika memory tidak bisa dialokasikan. Maka kode untuk **stack overflownya** seperti ini:

```
if(newNode == NULL)
{
    printf("Stack Overflow.");
    exit(1);
}
```

malloc() function returns NULL when the requested memory can't be allocated.

b. Fungsi **isEmpty()**:

```
int isEmpty()
{
    if(top == NULL)
        return 1;
    else
        return 0;
}
```

c. Fungsi **peek()**:

```
int peek()
{
    if(isEmpty())
    {
        printf("Stack Underflow.");
        exit(1);
    }
    return top->data;
}
```

d. Fungsi **size()** tinggal melakukan **traversing** dan **count++**

e. Fungsi **print()** tinggal melakukan **traversing** dan **printf**

Linked List Implementation (infix, postfix, prefix)

Prefix	Infix	Postfix
* 4 10	4 * 10	4 10 *
+ 5 * 3 4	5 + 3 * 4	5 3 4 * +
+ 4 / * 6 – 5 2 3	4 + 6 * (5 – 2) / 3	4 6 5 2 – * 3 / +

- Prefix : operator is written before operands
- Infix : operator is written between operands
- Postfix : operator is written after operands

- Why do we need prefix/postfix notation?
- Prefix and postfix notations **don't need brackets** to prioritize operator precedence.
- Prefix and postfix is much **easier for computer to evaluate.**

Operator priority:

Operators	Priority
^ (Exponentiation)	1 (Highest)
* (Multiplication) and / (Division)	2
+ (Addition) and - (Subtraction)	3 (Lowest)

a. Evaluation: Infix notation

- Evaluate a given infix expression: 4 + 6 * (5 – 2) / 3.
- To evaluate infix notation, we should search the highest precedence operator in the string.

4 + 6 * (5 – 2) / 3 search the highest precedence operator, it is ( )
4 + 6 * 3 / 3 search the highest precedence operator, it is *
4 + 18 / 3 search the highest precedence operator, it is /
4 + 6 search the highest precedence operator, it is +
10

b. Evaluation: Postfix notation

Scan from left to right

7 6 5 x 3 2 ^ – +, scan until reach the first operator

7 6 5 x 3 2 ^ – +, calculate 6 x 5

7 30 3 2 ^ – +, scan again until reach next operator

7 30 3 2 ^ – +, calculate 3^2

7 30 9 – +, scan again to search next operator

7 30 9 – +, calculate 30 – 9

7 21 +, scan again

7 21 +, calculate 7 + 24

28 , finish

Using stack:

String	Stack	
4 6 5 2 – * 3 / +		
4 6 5 2 – * 3 / +	4	push(4)
4 **6** 5 2 – * 3 / +	4 6	push(6)
4 6 **5** 2 – * 3 / +	4 6 5	push(5)
4 6 5 **2** – * 3 / +	4 6 5 2	push(2)
4 6 5 2 **–** * 3 / +	4 6 3	A = pop(), B = pop(), push(B – A) → A = 2, B = 5, push(3)
4 6 5 2 – ***** 3 / +	4 18	A = pop(), B = pop(), push(B * A) → A = 3, B = 6, push(18)
4 6 5 2 – * **3** / +	4 18 3	push(3)
4 6 5 2 – * 3 **/** +	4 6	A = pop(), B = pop(), push(B / A) → A = 3, B = 18, push(6)
4 6 5 2 – * 3 / **+**	10	A = pop(), B = pop(), push(B + A) → A = 6, B = 4, push(10)

c. Evaluation: Prefix notation

Manually

Scan from right to left

\+ 7 – x 6 5 ^ 3 2

\+ 7 – x 6 5 ^ 3 2

\+ 7 – x 6 5 9

\+ 7 – x 6 5 9

\+ 7 – 30 9

\+ 7 – 30 9

\+ 7 21

\+ 7 21

28

Using stack:

- Evaluating a prefix notation is similar to postfix notation.
- Hint: the string is scanned from **right to left.**

d. Conversion: Infix to Postfix

- Algorithm:

1. Search for the operator which has the highest precedence
2. Put that operator behind the operands
3. Repeat until finish

Manually	
A + B – C x D ^ E / F	, power has the highest precedence
A + B – C x D E ^ / F	, put ^ behind D and E
A + B – C x D E ^ / F	, x and / have same level precedence
A + B – C D E ^ x / F	, put x at the end
A + B – C D E ^ x / F	, continue with the same algorithm till finish
A + B – C D E ^ x F /	
A + B – C D E ^ x F /	
A B + – C D E ^ x F /	
A B + – C D E ^ x F /	
A B + C D E ^ x F / –	, this is the Postfix notation

Using stack:

Algorithm:

- Scan the string from left to right, for each character in the string:
- If it is an operand, add it to the postfix string.
- If it is an open bracket, push it into stack.
- If it is a close bracket, pop the stack until you found an open bracket. Add each popped element to the postfix string.
- If it is an operator, pop while the stack's top element has higher or equal precedence than the scanned character. Add each popped element to the postfix string. Push the scanned character into stack.
- After you have scanned all character, pop all elements in stack and add them to postfix string.

String	Stack	Postfix String
4 + 6 * (5 – 2) / 3		
4 + 6 * (5 – 2) / 3		4
4 + 6 * (5 – 2) / 3	+	4
4 + 6 * (5 – 2) / 3	+	4 6
4 + 6 * (5 – 2) / 3	+ *	4 6
4 + 6 * (5 – 2) / 3	+ * (	4 6
4 + 6 * (5 – 2) / 3	+ * (	4 6 5
4 + 6 * (5 – 2) / 3	+ * (–	4 6 5
4 + 6 * (5 – 2) / 3	+ *	4 6 5 2
4 + 6 * (5 – 2) / 3	+ * /	4 6 5 2 –
4 + 6 * (5 – 2) / 3	+ /	4 6 5 2 – *
4 + 6 * (5 – 2) / 3	+ /	4 6 5 2 – * 3
4 + 6 * (5 – 2) / 3		4 6 5 2 – * 3 / +

e. Conversion: Infix to Prefix

- Algorithm:

1. Search for the operator which has the highest precedence
2. Put that operator before the operands
3. Repeat until finish

Manually

A + B – C x D ^ E / F
A + B – C x ^ D E / F
A + B – C x ^ D E / F
A + B – x C ^ D E / F
A + B – x C ^ D E / F
A + B – / x C ^ D E F
A + B – / x C ^ D E F
+ A B – / x C ^ D E F
+ A B – / x C ^ D E F
– + A B / x C ^ D E F

Using stack:

Sama seperti Infix to Postfix, tapi harus di reverse dulu, terus hasilnya di reverse lagi agar lebih mudah.

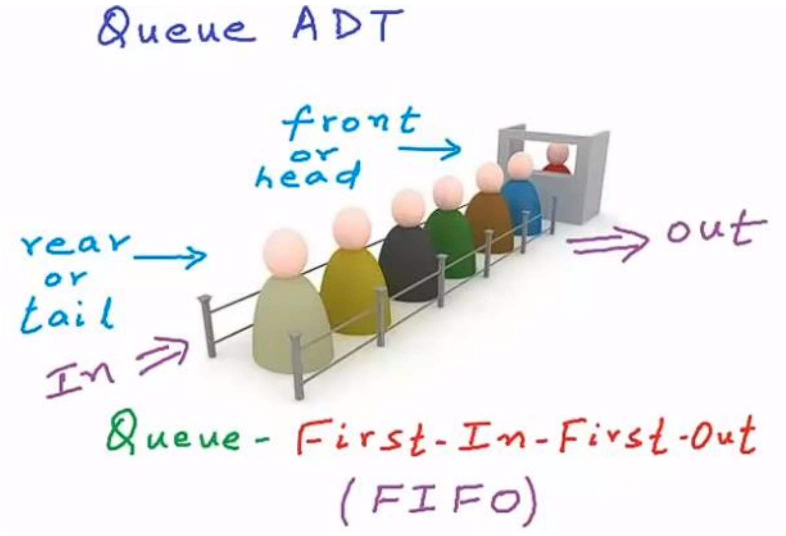
EXERCISE:

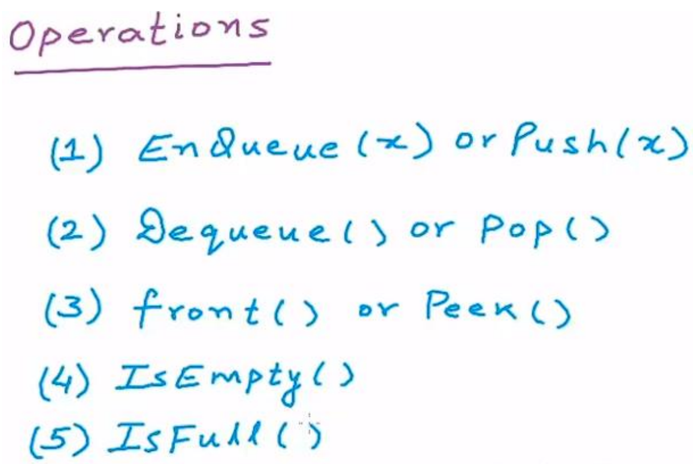
8*(5^4+2)-6^2/(9*3)

Linked List Implementation (Depth First Search (DFS))

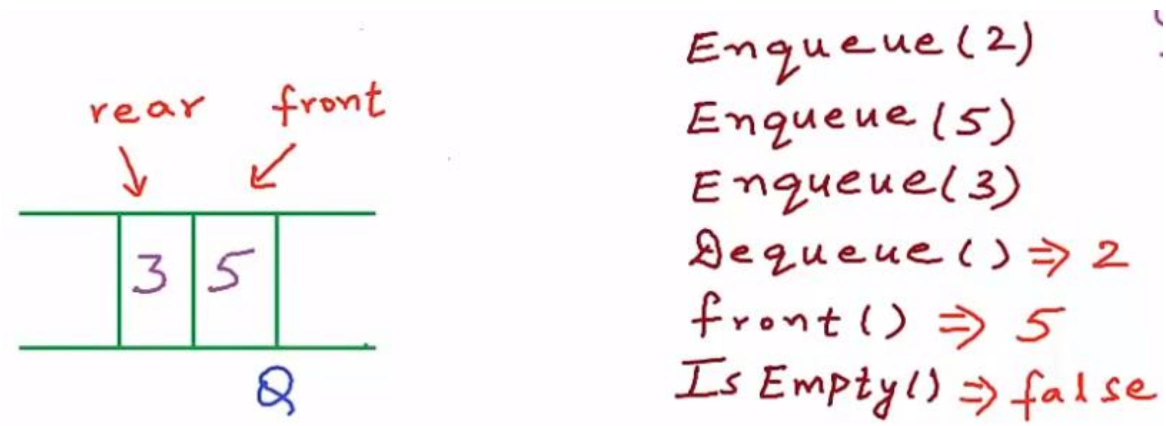
[(32) 6.2 BFS and DFS Graph Traversals| Breadth First Search and Depth First Search | Data structures - YouTube]

Queue (FIFO)





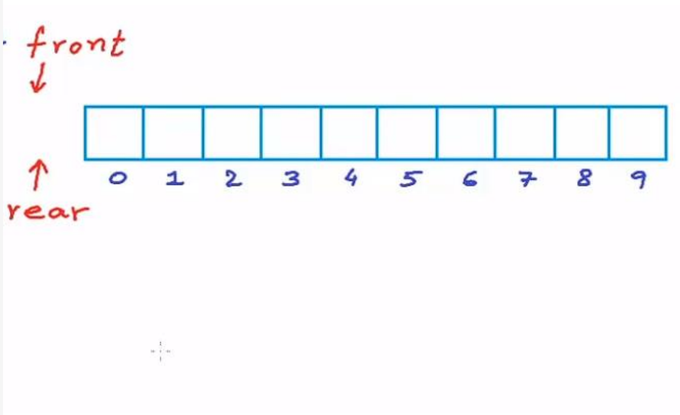
EXAMPLE:



Array Representation

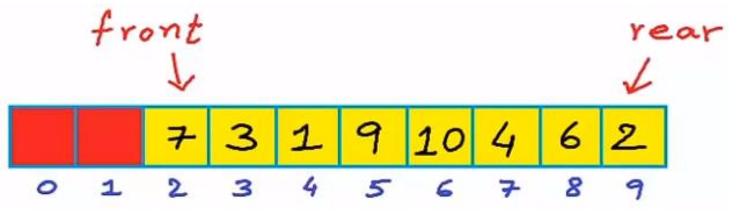
a. **isEmpty()** algorithm:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int queue_arr[MAX];
int front = -1;
int rear = -1;

int isEmpty(){
    if (front == -1 && rear == -1){
        return 1;
    }
    else{
        return 0;
    }
}
```



b. **isFull()** algorithm:

```c
int isFull(){
    if (rear == MAX - 1){
        return 1;
    }
    else{
        return 0;
    }
}
```

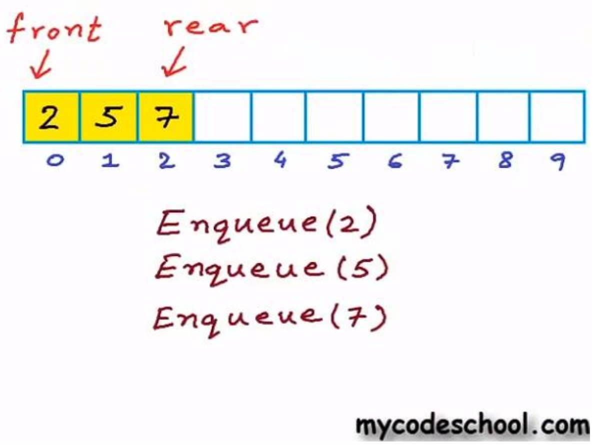


c. **print()**:

```c
void print(){
    int i;
    if(isEmpty()){
        printf("List Empty\n");
        return;
    }
    for(i=front; i<=rear; i++){
        printf("%d ", queue_arr[i]);
    }
    printf("\n");
}
```

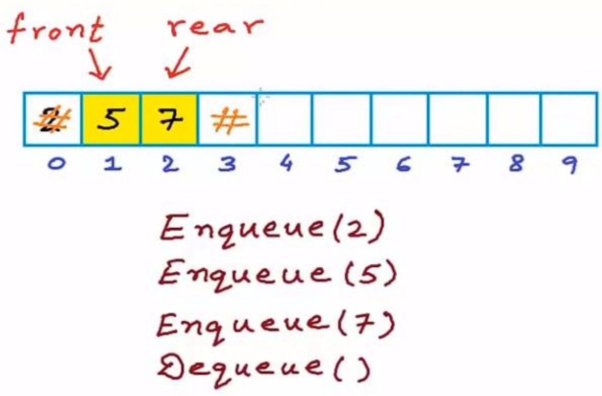
d. **enqueue(x)** algorithm:

```c
void enqueue(int data){
    if(isFull()){
        printf("List Full\n");
        return;
    }
    else if (isEmpty()){
        front = front + 1;
        rear = rear + 1;
    }
    else{
        rear = rear + 1;
    }
    queue_arr[rear] = data;
}
```

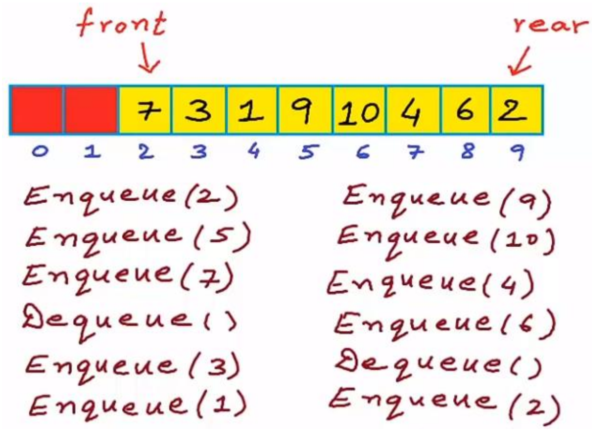


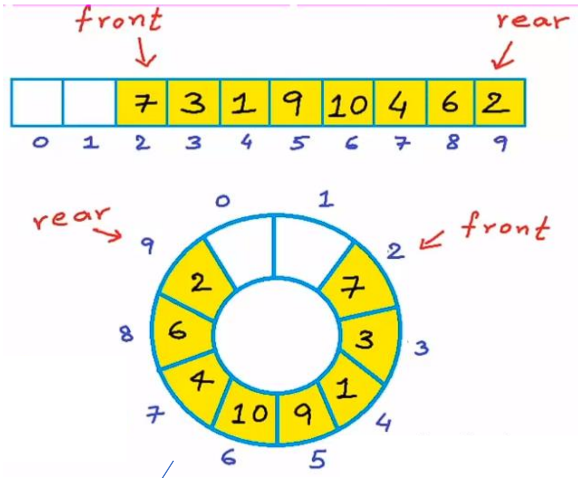
e. **dequeue()** algorithm:

```c
int dequeue(){
    int value;
    if(isEmpty()){
        exit(1);
    }
    else if (front == rear){
        value = queue_arr[front];
        front = -1;
        rear = -1;
    }
    else{
        value = queue_arr[front];
        front = front + 1;
    }
    return value;
}
```



Circular array:





Current position $= i$

Next position $= (i+1) \% N$

Previous position $= (i+N-1) \% N$

```
int next(int data){
    return (data+1)%MAX;
}

int prev(int data){
    return (data+MAX-1)%MAX;
}
```

new_isFull():

```c
int new_isFull(){
    if(rear != front && next(rear) == front){
        return 1;
    }
    else{
        return 0;
    }
}
```

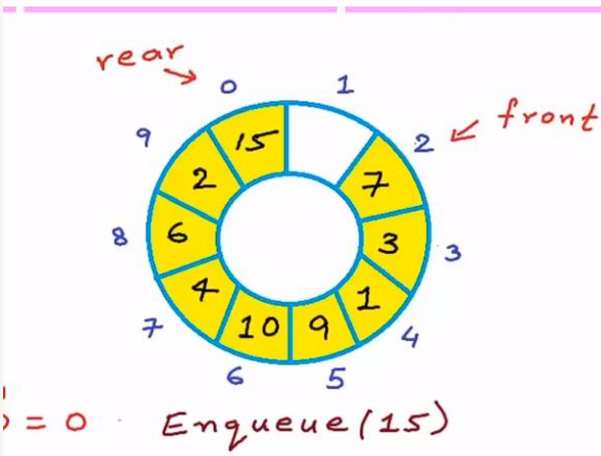
new_print():

```c
void new_print(){
    if(isEmpty()){
        printf("List Empty\n");
        return;
    }

    if (rear < front){
        for(int i=front; i<MAX; i++){
            printf("%d ", queue_arr[i]);
            if (i == MAX-1 && i != rear){
                for(int j=0; j<=rear; j++){
                    printf("%d ", queue_arr[j]);
                }
            }
        }
    }
    else{
        for(int i = front; i<=rear; i++){
            printf("%d ", queue_arr[i]);
        }
    }
    printf("\n");
}
```

Kalau dia udah nyentuh maximum index, reset ke index 0 lagi

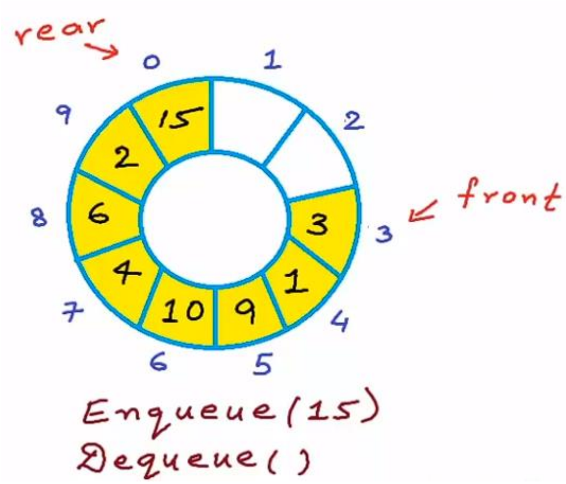
new_enqueue(x):

```c
void new_enqueue(int data){
    if(new_isFull()){
        printf("List Full\n");
        return;
    }
    else if (isEmpty()){
        front = front + 1;
        rear = rear + 1;
    }
    else{
        rear = next(rear);
    }
    queue_arr[rear] = data;
}
```



new_dequeue() algorithm:

```
int new_dequeue(){
    int value;
    if(isEmpty()){
        exit (1);
    }
    else if (front == rear){
        value = queue_arr[front];
        front = -1;
        rear = -1;
    }
    else{
        value = queue_arr[front];
        front = next(front);
    }
    return value;
}
```



Linked List Representation

Pada linked list, **head** jadi **front** dan **tail** jadi **rear**.

Jika datanya baru 1 maka front dan rear menunjuk ke 1 node yang sama.

Setiap **enqueue()**, masuk lewat **rear**.

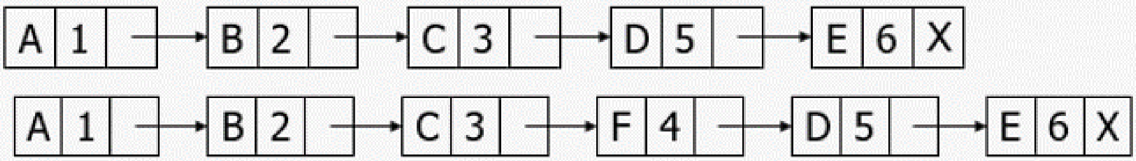
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```

Setiap **dequeue()**, masuk lewat **front**.

```
void Dequeue() {
    struct Node* temp = front;
    if(front == NULL) return;
    if(front == rear) {
        front = rear = NULL;
    }
    else {
        front = front->next;
    }
  free(temp);
}
```

Priority Queue

- Priority queue after **insertion** of a new node:



- Lower priority number means higher priority.

- **Deletion:**
- Deletion is a very simple process in this case.
- The first node of the list will be deleted and the data of that node will be processed first

Breadth First Search (BFS)

[(32) 6.2 BFS and DFS Graph Traversals| Breadth First Search and Depth First Search | Data structures - YouTube]