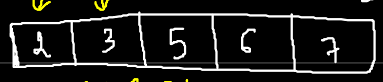
Arrays

scalar → int x = 10;

vector → int x[5];

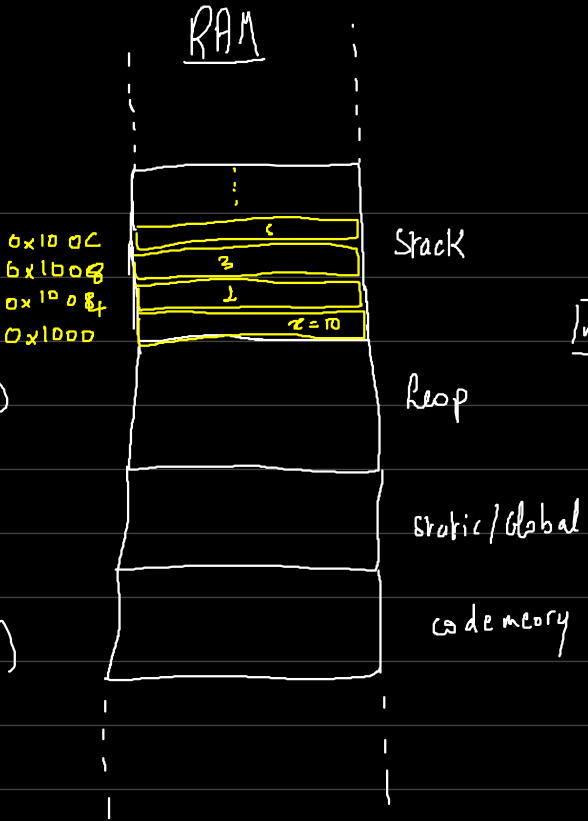
index : 0 1 2 3 4

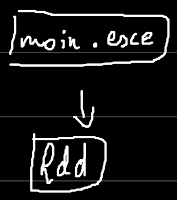


garbage values at the beginning

_ 910135

A=0x1004 0x1008 0x100C 0x1010 0x1014
=A+1 A+2





int A[5] = {2,3,5,6,7}

A[0] → 2

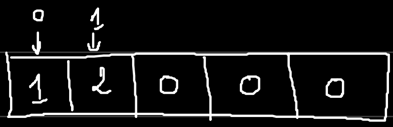
A[3] → 6

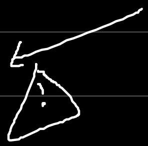
A[4] → 7

printf("%d", A[0])

std::cout << A[0]

int A[5] = {1,3};





A → 0x1004

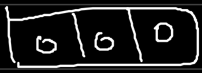
A+1 → 0x1004 + sizeof(int)

→ 0x1008

*(A+1) = *(0x1008)

= 3

int A[3] = {0};



A[2] → 0

2[A] → 0

1[A] → 2

A[0] ≡ 0[A] → 1

*(A+2) ≡ A[2] ≡ 2[A] = 0

Static vs dynamic Arrays

int A[5] = {1,2,4,3,5};

int n;
cin >> n;

int *p;

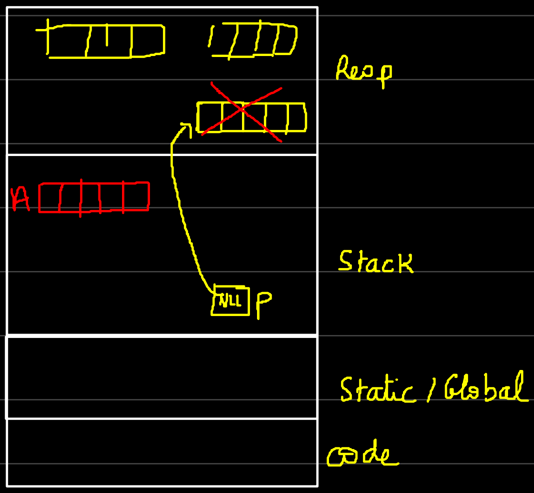
p = new int[n];

delete []p;

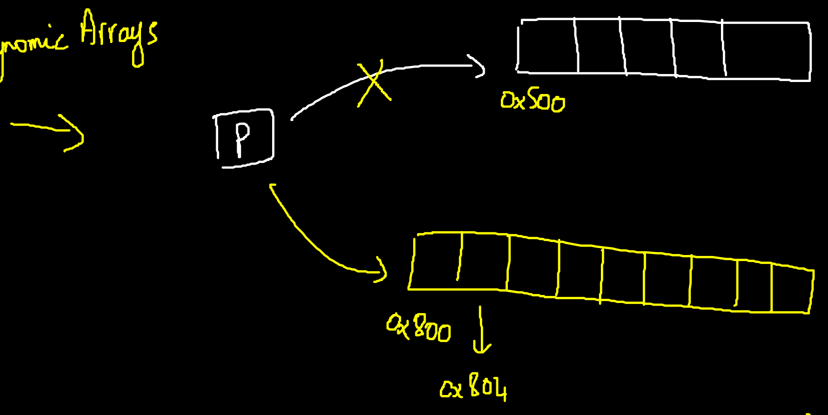
p = NULL;

int* p;

p = new int[5];



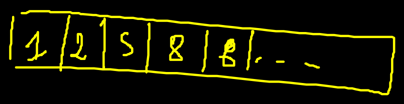
Dynamic Arrays



```
int* p;
p = new int[5];
delete [] p;

p = new int[9];
```

arr → {1, 2, 5, 8, 6}



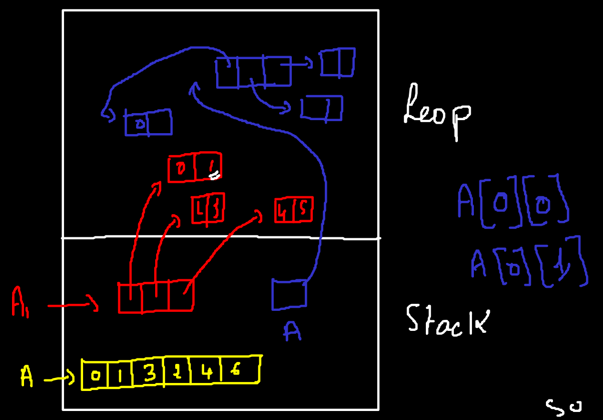
0x600   0x600 + Sizeof(int)
= 0x604

2-D Arrays

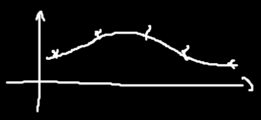
① int A[2][3] = {{0,1,3},{2,4,6}};

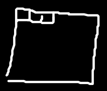
$$\begin{array}{c|ccc} \text{2rows} & 0 & 1 & 3 \\ & 2 & 4 & 6 \\ \hline & \uparrow & \uparrow & \uparrow \\ & \multicolumn{3}{c}{\text{3 columns}} \end{array}$$

~~or~~ new

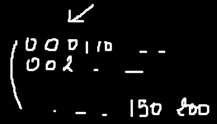


[15, 16, 10, 12]





0 → 255



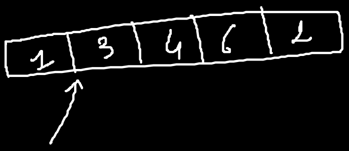
② int* A1[3];
A1[0] = new int[2];   A1[1] = new int[2];   A[2][0]   A[0][1] = 1

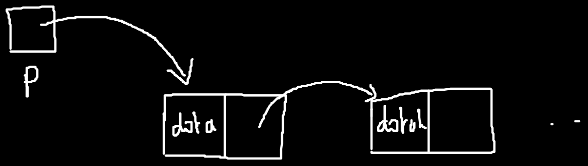
③ int** A;
A = new int* [3];
A[0] = new int[2];

640 ( ) 50 → 720 ( ) 1080 . . .

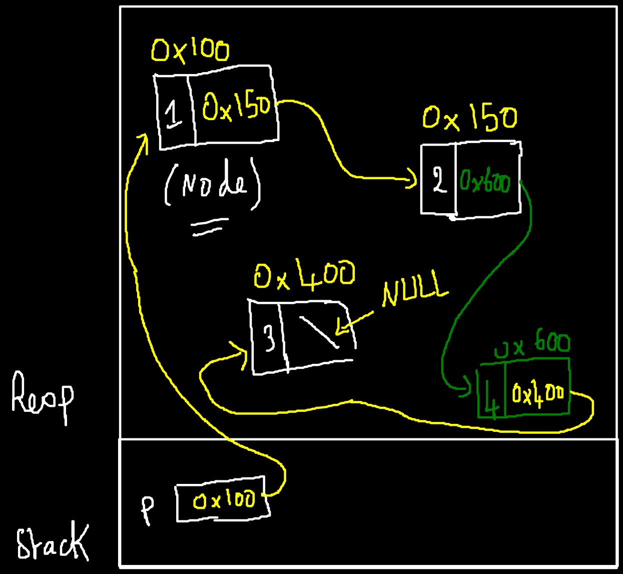
Linked lists

```
int A[5];
```





{1,2,3}



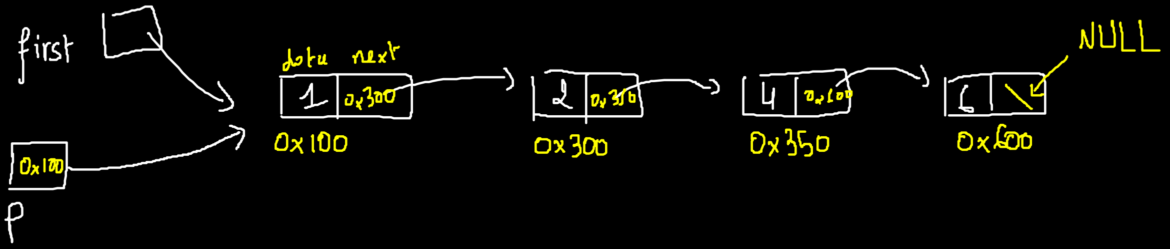
```
Struct Node {
        int data;
        Struct Node* next;
};
```

```
Struct Node * p;
p = new Node;
(*p).data = 1;   ⇔   p -> data =
(*p).next =  ;   ⇔   p -> next =
```

```
p=NULL;
if(p !=NULL){        if(p==NULL)
   ...
```

```
if(p)  ⇔  if(p!=NULL)          if(!p)  ⇔  if(p==NULL)
```

```
while (p1->next != NULL)
{
        cout << p1->data;
        p1 = p1->next;
}
```

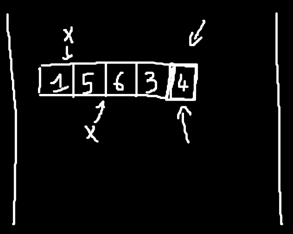


```
void  Display ( Struct  Node * p ) {
                while ( p!=NULL)
                {       cout << p->data;
                   ->  p = p->next;
                 }
          };
                                          Display (first);
```

```
int  Search  ( Struct  Node * p,  int a)
{     int i=0   <------------------  int  sum = 0;
      while ( p != NULL)
      {                   ------>    sum += p->data;
            if ( p->data = a )
            {
                  return i;
            }
            i++;
      }
      return -1;
}
```

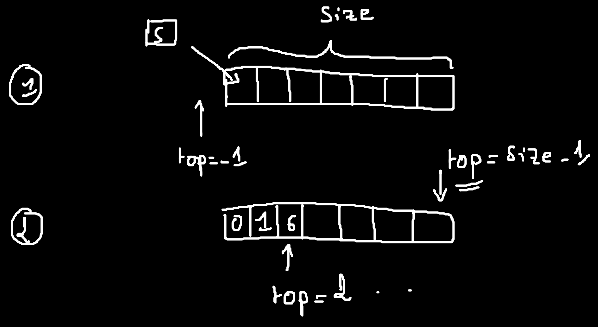
Stack

complexity => O(1)



LIFO

Last In first out



```
Struct Stack{
        int size;
        int top;
        int* s;
};
```

```
void create (Stack* st , int size)
{
        st -> size = size;
        st -> top = _1;
      st->S = new int [st->size]
}
```

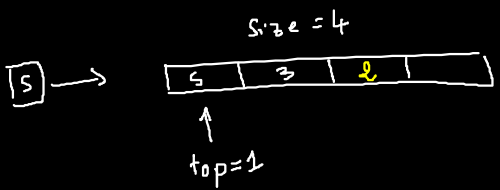
st -> size = (* st). size

top == -1 => Stack is empty

top == size _ 1

```
bool  isFull (Stack st)
{
      return st. top == st. size - 1;
  }
```

```
bool  isEmpty (Stack st]
{
      return st. top == -1;
                      true
 }
```



```
int pop {
  value = -1;
   if (!isEmpty (st))
   {
        value = st->s [st -> top];
        st -> top --;
   }
   return value;
```

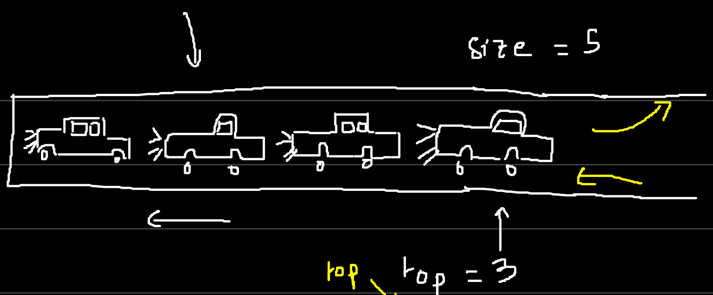
push -> void

```
st -> top ++
st -> s [st -> top] = value;
```

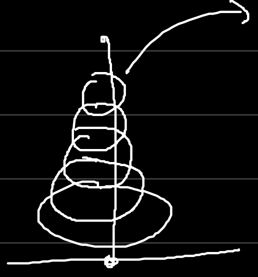
display :

i <- top

i   2 -> 3 -> 5



LIFO



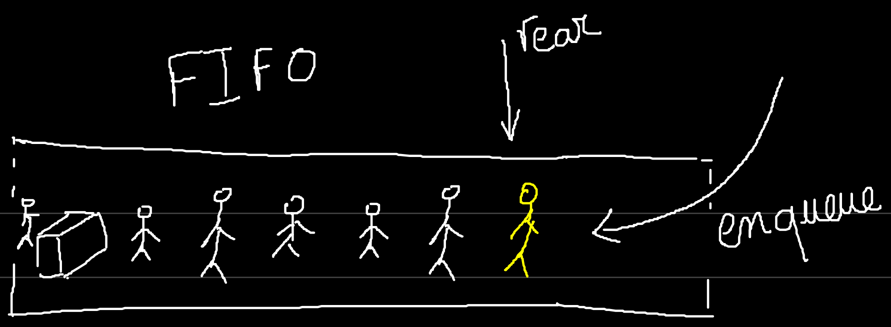
int

top

3	4	2	1	

Size = 5

Queues



dequeue

Size = 9

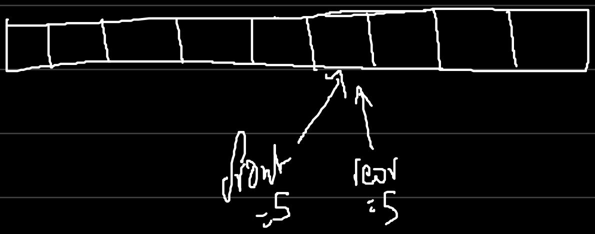
q->rear == q->size-1;

enqueue(1)
enqueue(6)

O(1)

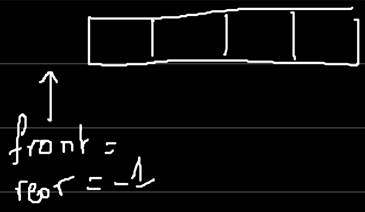
int dequeue()
O(n)

void enqueue(int v) | int rear;

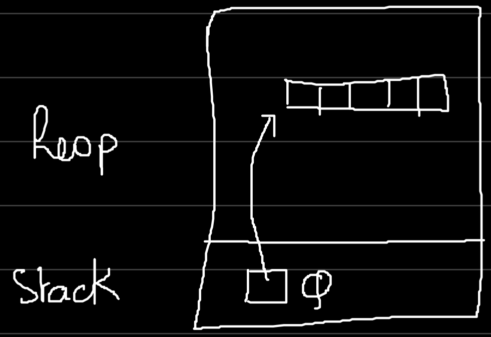


== | =

```
isEmpty(Queue q)
{
 return q.front == q.rear;
```



```
struct Queue {
    int size;
    int front;
    int rear;
    int *Q;
}
```



Q = new int[Size];

if (a=b) <- error

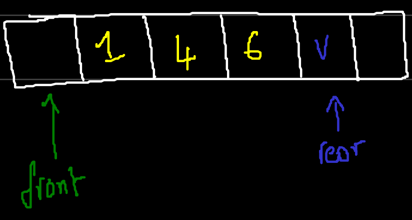
if (a==b)

```
int a = 0;
int b = 1;
if (a=b) <- error
{
    ->
    ->
    ->
}
```

```
void create(Queue *q, int size)

    q->size = size;
    q->front = q->rear = -1;
    q->Q = new int[q->size];
                front++;
                value = Q[front]
}
```

i(front+1) ——> i <= q.rear



delete 3

rear++;
Q[rear] = v