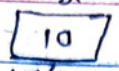
29/03/2022

Array

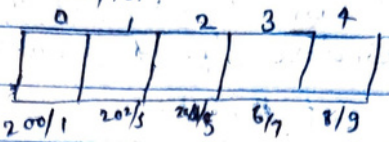
Date
Page

int x = 10;



→ Scalar

int A[5] = A



→ Vector

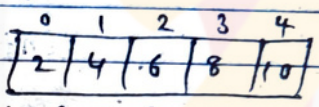
* Arrays are vector variable.

* Declaration and initialization of an array -

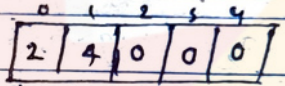
int A[5];

garbage value

int A[5] = {2, 4, 6, 8, 10};

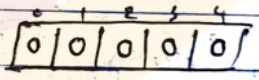


int A[5] = {2, 4};

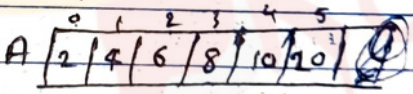


(Remaining are initialize with 0)

int A[5] = {0};

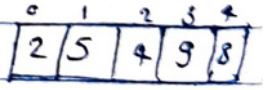


int A[] = {2, 4, 6, 8, 10, 20}



* Accessing of an array -

int A[5] = {2, 5, 4, 9, 8};



① for (i=0; i<5; i++)
{ printf ("%d", A[i]);
}

② printf ("%d", A[2]);

③ " " ", 2[A]);

④ " " ", *(A+2));

30/03/2022

Date
Page

Static vs Dynamic Array

Static Array

* In c language, when you are mentioning the size of an array, it must be a constant value, because size of an array decided at compile time.

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

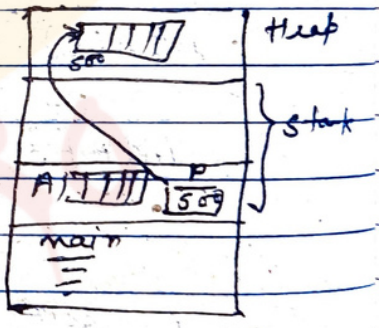
* But in c++, we can creat an array of any size at run time inside the stack whereas in c language the size has to be decided at compile time only.

```
int n;
cin>>n;
int B[n];
```

* In any language, when you declare a variable, the memory for that variable will go inside the stack only.

Dynamic Array

```
void main()
{ int A[5];
  int *P;
  P = new int[5];
P = (int *) malloc (5 * size of (int));
```



* When you declare a normal array then it will be created inside stack, and when you want an array in heap then you must have a pointer and then you should allocate the memory using new operator (c++) or using malloc. malloc fun. (c)

it will be created in a heap and the heap memory can be accessed with the help of pointer.

* Accessing of an static and dynamic array

```
int A[5];
int *P;
P = new int[5];
A[0] = 5;
P[0] = 5;
```

Note - Once the array of some size is created, it cannot be resized.

→ If at all, you want to increase (change) the size of an array, it is possible in another way, but it is possible only in heap. Stack array can not be resized at all.

* How to increase the size of an array -

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int *P, *q;
  int i;
P = (int *)malloc(5 * sizeof(int));
P[0] = 3; P[1] = 5; P[2] = 7; P[3] = 9; P[4] = 11;
q = (int *)malloc(10 * sizeof(int));
    for (i = 0; i < 5; i++)
        q[i] = P[i];
    free(P); // Deallocation of P memory
```

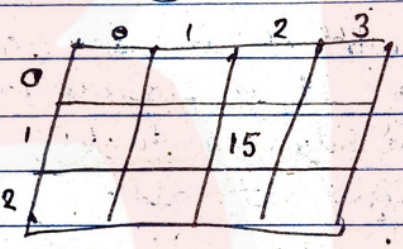
```
P = q;
q = NULL;
for (i = 0; i < 5; i++)
    printf("%d \n", P[i]);
return 0;
}
```

31/03/2022

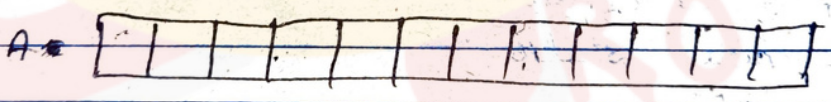
2-D Array - (Initialization methods) -

i) `int A[3][4] = {{1,2,3,4}, {2,4,6,8}, {3,5,7,9}};`

3×4 = (12)



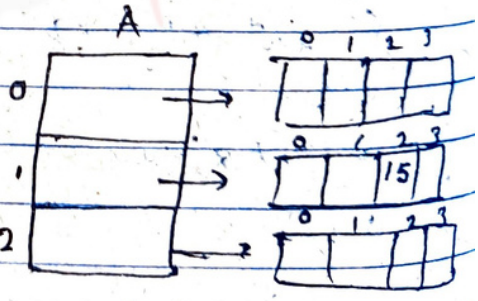
`A[1][2] = 15;`



ii) `int * A[3];` ← in stack

All are created in heap

```
A[0] = new int [4];
A[1] = new int [4];
A[2] = new int [4];
```



`A[1][2] = 15;`

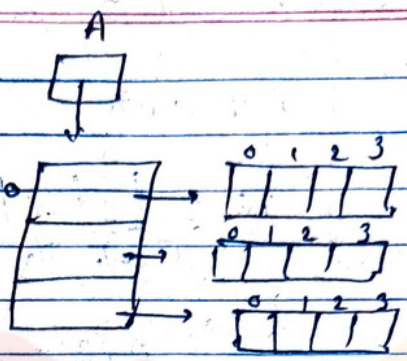
Date
Page

(iii) int **A;

It will creat in stack

All are created in heap memory

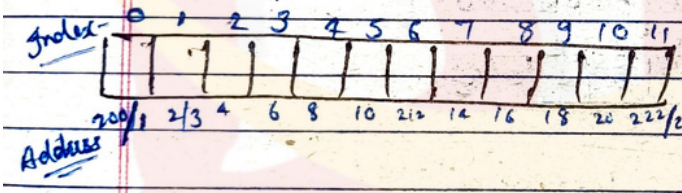
```
A = new int*[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

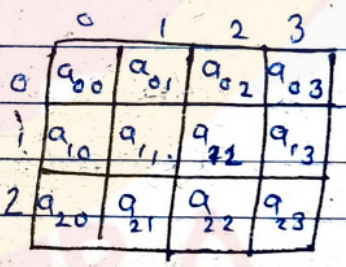



```
for(i=0; i<3; i++)
{ for(j=0; j<4; j++)
    { A[i][j] = ____;
    }
}
```

* Arrays in compilers -

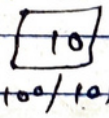
int A[3][4];





Note - When we access any element then it is done by its address not by their names.

Eg:- int x = 10;



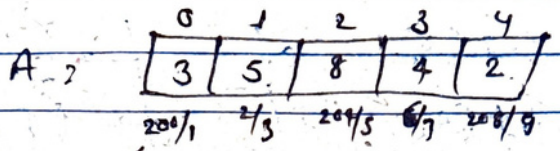
(To access this machine code should prefer its address i.e 100 not its name)

But address can be known at only execution time not compilation time, because memory for the variable is allocated at execution.

To know the address of variable at compilation time, compiler generate a formula -

Date Page

```
int A[5] = {3, 5, 8, 4, 2};
```



L_0

A[3] = 10;

Add(A[3]) = 200 + 3 * 2 = 206

Add(A[3]) = L_0 + 3 * 2

Formula for obtaining address of any variable

$$Add(A[i]) = L_0 + i * w$$

L_0 : Base address, i : index, w : Size of data type

more faster

* If index is starting onwards 1

i.e. A [3 | 5 | 8 | 4 | 2]
1 2 3 4 5

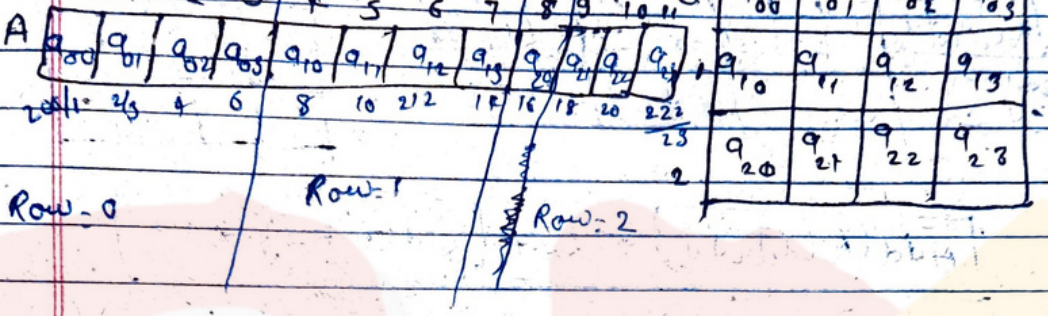
$$Add(A[i]) = L_0 + (i-1) * w$$

(It is not more faster than above)

Date
Page

* Row major mapping (2D-array)

int A[3][4]
m × n



A[i][2] = 10;

Add(A[i][2]) = 200 + [1 * 4 + 2] * 2 = 212

Add(A[2][3]) = 200 + [2 * 4 + 3] * 2 = 222

∴ Add(A[i][j]) = L_0 + [i * n + j] * w

more fast

If index are starting from 1 -

int A[1...3][1...4]

Add(A[i][j]) = L_0 + [(i-1) * n + (j-1)] * w

C++ language follow Row major formula.

A[10][15] A[10][15]

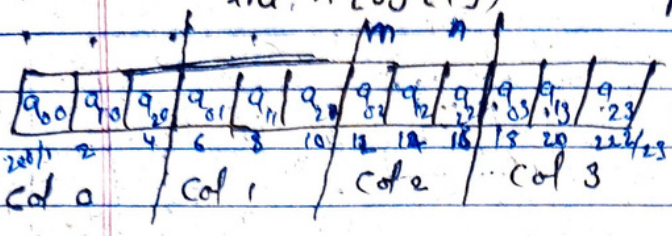
= 100 + [A[i][j] = 100 + [i * 15 + j] * 4

100 + [15 i + j] * 4 = 100 + [15 i + j] * 4 = 100 +

Date
Page

* Column major mapping (2-D)

int A[3][4];



A	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

$$Add(A[i][j]) = L_0 + [j * m + i] * w$$

$Add(A[2][2]) = 200 + [2*3+2]*2 = 214$

$Add(A[1][3]) = 200 + [3*3+1]*2 = 220$

01/04/22

* Row major and Column major formula for 'n' dimension

Type $A[d_1][d_2][d_3][d_4]$;

Row-major (Left to Right)

$$Addr(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4] * w$$

For n-dimension:

$$L_0 + \sum_{p=1}^{n} \left[i_p * \prod_{q=p+1} d_q \right] * w$$

Column major (Right to left)

$$Add(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4 * d_1 * d_2 * d_3 + i_3 * d_1 * d_2 + i_2 * d_1 + i_1] * w$$

$$L_0 + \sum_{p=n}^{1} \left[i_p * \prod_{q=n-1} d_q \right] * w$$

For Row-major formula -

$$L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4] * w$$

(3, 2, 1)

4-D = 3 + 2 + 1 (Multiplication)

5-D = 4 + 3 + 2 + 1

n-D = $n-1 + n-2 + \dots + 3 + 2 + 1 = \frac{n(n-1)}{2}$

time - $\frac{n(n-1)}{2} = O(n^2)$ → ~~it take too~~ much time.

* To Reduce the time we use Horner's ~~rule~~ rule -

Add(A[i1][i2][i3][i4]) =

$L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4] * w$

~~$i_4 + d_4$~~

$i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4$

$i_4 + d_4 [i_3 + i_2 * d_3 + i_1 * d_2 * d_3]$

$= i_4 + d_4 * [i_3 + d_3 * [i_2 + i_1 * d_2]]$

↑ ↑ ↑ → 3

4-D = 3 (multiplication)

5-D = 4

n-D = n-1

time - $(n-1) = O(n)$

Date
Page

Formula for 3-D array -

int A[l][m][n];

Row major - (left to R)

Add(A[i][j][k]) = L_0 + [i * m * n + j * n + k] * w

Column major (R - L)

Add(A[i][j][k]) = L_0 + [k * l * m + j * l + i] * w

04/09/2022

* Creating an array in heap memory -

```
#include <stdio.h>
struct array
  {
    int *A;
    int size;
    int length;
  };
int main()
  {
    struct Array arr;
    printf("Enter size of an array");
    scanf("%d", &arr.size);
    arr.A=(int *) malloc(arr.size * sizeof(int));
    arr.length =0
    int n, i;
    printf("Enter number of numbers");
    scanf("%d",&n);
    printf("Enter all elements\n");
```

```
for(i=0; i<n; i++)
scanf("%d", &arr.A[i]);
arr.length = n;
Display(arr);
}
int i; Void Display(struct Array arr)
{
    int i;
    printf("\n Elements are\n");
    for(i=0; i<arr.length; i++)
        printf("%d", arr.A[i]);
}
```

Note - If you want to create an array of desired size, for that you can take a pointer and later you can create during run time.

```
int *A;
size = 10;
A = new int[size];
```

```
int A[10];
```

↓

If you follow this one then you must mention the size.

Inserting and Appending in a Array

```
#include <stdio.h>
struct Array
{ int A[10];
  int size;
  int length;
};
void display (struct Array arr)
  { int i;
    printf("\n Elements are\n");
    for(i=0; i<arr.length; i++)
      printf ("%d", arr.a[i]);
  }
Insert
void append (struct array *arr, int index, int x)
{ int i;
  if (index >= 0 && index <= arr->length)
    { for (i = arr->length; i > index ; i--)
      arr->A[i] = arr->A[i-1];
    arr-> A[index] = x;
    arr-> length++;
    }
  }
int main()
  { struct array arr1 = {{2,3,4,5,6}, 10, 5};
    append(&arr1, 10);
    insert
```

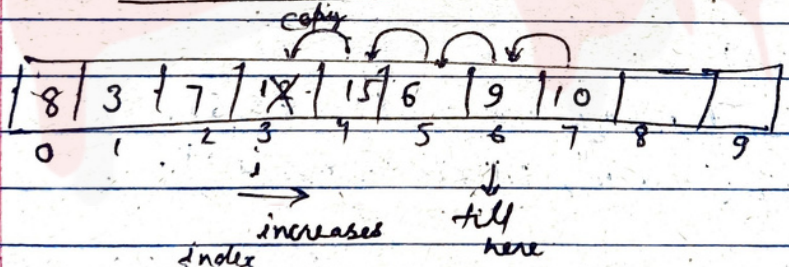
```
Void append (struct Array *arr, int x)
{ if (arr->length < arr->size)
    arr->A[arr->length++] = x;
}
int main()
{ struct Array arr1 = {{2,3,4,5,6}, 10, 5};
  Append (&arr1, 10);
  Insert (&arr1, 0, 12);
  Display (arr1);
  return 0;
}
```

Time taken

min - O(1) (when no shifting is required)

max - O(n)

* Delete Element



Delete (3)

```
x = A[index]          — 1
for (i = index; i < length-1; i++)
    A[i] = A[i+1];    — n
length--;             — 1
```

time - min = 2 → O(1) → Best case

max = 2+n → O(n) → Worst case

* Deleting an element of an array

```
#include<stdio.h>
struct Array
{ int A[0];
  int size;
  int length;
};
int delete(struct Array * arr, int index)
{ int x = 0;   //deleting value
  int i;
if (index >= 0 && index < arr->length)
  {
    x = arr->A[index];
    for (i=index ; i< arr->length-1; i++)
      arr->A[i] = arr->A[i+1];
      arr->length--;
      return x;
  }
  return 0;
}
int main()
{ struct Array arr = {{2,3,4,5,6}, 10, 5};
  printf("%d\n", Delete(&arr, 4));
  Display(arr);   (Its code is in previous question)
return 0;
```

05/08/2021

* **Linear Search** (Element must be unique)

Searching all element one by one from starting.

A	8	9	4	7	6	3	10	5	14	2
	0	1	2	3	4	5	6	7	8	9

size = 10
length = 10

For key = 5 ↓ successful

```
for(i=0; i<length; i++)
{
    if (key == A[i]
        return i;
}
return -1;
```

time -
min - O(1) - Best case (When element is present at first index)
max O(n) - Worst case (When element is present at last index)

$$Avg = \frac{1+2+3+\cdots+n}{n} = \frac{\frac{n(n+1)}{2}}{n}$$

$$= \frac{n+1}{2}$$

$$= O(n)$$

* Worst case and avg case will take the same amount of time.

* **Improving linear Search**

i) Transposition (comes closer to first element ~~one by one~~)
(Move the element that we want to search, to the previous location, just one)

key = 10

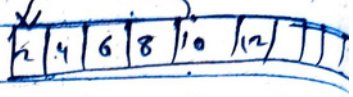
```
for(i=0; i<length; i++)
{   if (key == A[i])
    {   swap(A[i], A[i-1]);
        return (i-1);
    }
}
```

2	4	6	8	10	12

ii) Move to front / Head

(Bring the element to the first index directly)

```
for(i=0; i<length; i++)
{ if(key == A[i]
  { swap(A[i], &A[0]);
    return 0;
  }
}
```



Binary Search (for sorted list)

A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	8	10	15	18	21	24	27	29	33	34	37	39	41	43

size = 15
length = 15

key = 18

lower (L)	higher (H)	mid ($\frac{l+h}{2}$)
0	14	7
0	6	3
4	6	5
4	4	4 (found)

When key > mid then
L = mid + 1
H = same
and key < mid
H = mid - 1
L = same

key = 34

L	H	M
0	14	7
8	14	11
8	10	9
10	10	10 (found)

Algorithm for Binary search -

(tail recursion)

Iterative

```
Binarysearch (l, h, key)
{
  while (l <= h)
    mid = [(l+h)/2];
    if (key = A[mid])
      return mid;
    else if (key < A[mid])
      h = mid - 1;
    else
      l = mid + 1;
  }
  return -1;
}
```

Recursive

```
RBinarysearch (l, h, key)
  if (l <= h)
  { mid = [(l+h)/2];
    if (key == A[mid])
      return mid;
    else if (key < A[mid])
      return RBinarysearch
        (l, mid-1, key);
    else
      return RBinarysearch
        (mid+1, h, key);
  }
  return -1
}
```

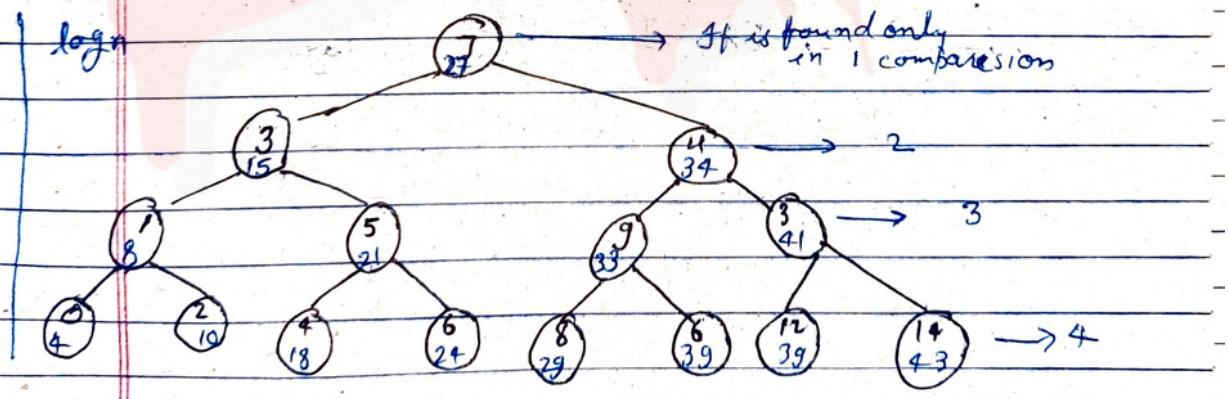
05/04/22

* Time Analysis

A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	8	10	15	18	21	24	27	29	33	34	37	39	41	43

Size = 15
length = 15



Time take will be dependent on the number of comparision and the number of comparision depend upon height of the tree.

So, the time taken for searching in binary searches = logn

Best case- min - $o(1)$ → for unsuccessful = $o(\log n)$

Worst case- max = $o(\log n)$

Date
Page

How binary search work -

let $n = 16$ (elements)

$\frac{16}{2} = 1, \frac{16}{2^4} = 1$

$\frac{\frac{16}{2}}{2}$

$\frac{\frac{\frac{16}{2}}{2}}{2}$

$\Rightarrow 2^4 = 16$

$\Rightarrow 4 = \log_2 16$ = $\boxed{\log_2 n}$

$\boxed{\log_2(n+1)}$ Exact formula

Avg time : $o(\log n)$

$1 + 1\times2 + 2\times4 + 3\times8$

$= 1 + 1\times2^1 + 2\times2^2 + 3\times2^3$

Avg time = $\boxed{\frac{\sum_{i=1}^{\log n} i \times 2^i}{n}}$ = $\frac{\log n \times 2^{\log n}}{n}$ = $\frac{\log n \times n}{n}$

= ~~$\log n \times n$~~ 1

= $\frac{\log n \times n}{n}$

= $\boxed{\log n}$

Avg time for successful searching = ~~$\log n$~~

Unsuccessful = $\log n$

Operation on Array

i) get(index) - It means we want to know element at a particular index i.e reading an element.

Eg:
```
get(index)
    if (index >= 0 && index < length)
        return A[index];
```
time - O(1)

ii) set() - It is used to replace a value at a particular index i.e writing an element.

```
set(index, x)
    if (index >= 0 && index < length)
        A[index] = x;
```
time - O(1)

iii) Max() - It is finding the max^m of the given elements.

```
1 —— max = A[0];
n —— for(i=1; i<length; i++)
     {
n-1 ——   if (A[i] > max)
             max = A[i];
     }
1 —— return max;
```
time - $f(n) = 2n+1$
$= O(n)$

iv) Min() - Finding the min^m value.

```
min = A[0];
for(i=; i<length; i++)
{   if (A[i] < min)
        min = A[i];
}
return min;
```
— O(n)

v) Sum() - finding the total.

```
1   Total = 0;
n+1 for (i=0; i<length; i++)
n   {
      Total = total + A[i];
    }
1   return total;
```

$P(n) = 2n+3$

$\therefore O(n)$

→ finding total by using recursion -

$$sum(A,n) = \begin{cases} 0 & n<0 \\ sum(A,n-1) + A[n] & n>=0 \end{cases}$$

```
int sum(A, n)        // call sum(A, length-1)
{ if (n<0)
    return 0;
  else
    return sum(A, n-1) + A[n];
}
```

time - O(n)

vi) Avg() - Finding the Avg.

```
total = 0;
for (i=0; i<length; i++)
    total = Total + A[i];
}
return Total/n;
```

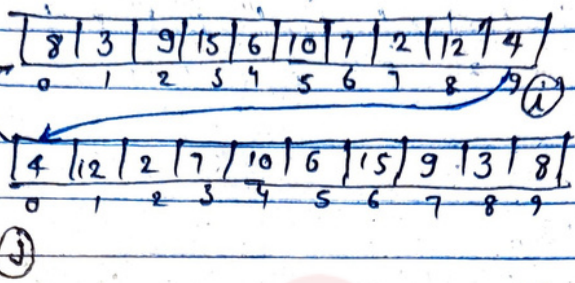
O(n)

* Reverse and shift an Array

Date
Page

i) Reverse -

1st method - using another array



size = 10
length = 10

It will copy A to B in reverse order.

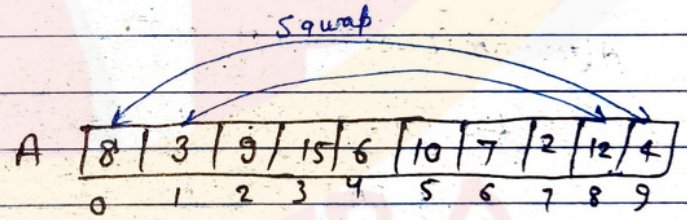
```
for (i = length-1, j=0; i >= 0; i--, j++)
{
    B[j] = A[i];
}
```

time - O(n)

It will copy B to A (not reverse)

```
for (i=0; i<length; i++)
{
    A[i] = B[i];
}
```

2nd method - by using swaping



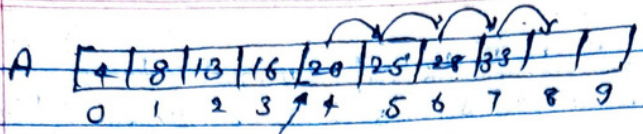

```
for (i=0; j = length-1; i<j; i++, j--)
{
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

07/04/22

Array ADT

Date
Page

i) Inserting in a sorted array -



Insert -18

(We will start from last index and check if key is less than previous one then shift until it found actual position)

```
x = 18;
i = length - 1;
while (A[i] > x)
  { A[i+1] = A[i];
    i--;
  }
A[i+1] = x;
```

ii) Checking if an array is sorted -

```
Algorithm isSorted (A, n)
{ for (i=0; i<n-1; i++)
    if (A[i] > A[i+1]
      return false;
  }
  return true;
}
```

iii) Arranging -ve on left side - (using two Pointers)

'i' will found positive

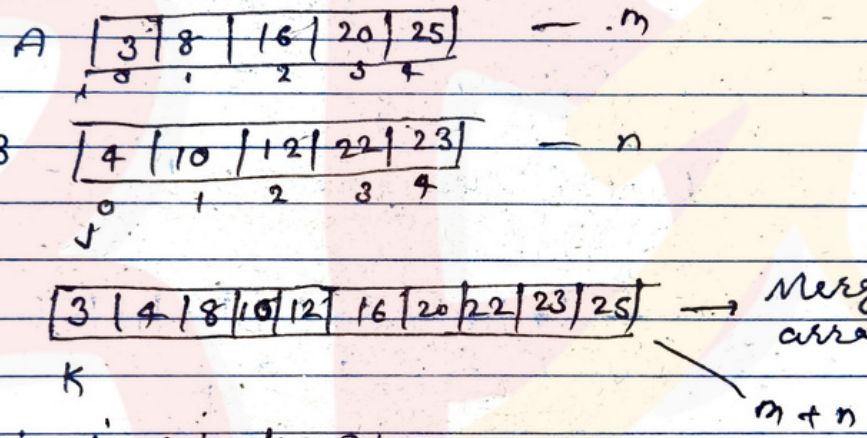
'j' will found negative

```
i = 0;
j = length - 1;
while (i < j);
{ while (A[i] < 0) {i++;}
```

```
while(A[j] >= 0) { j--; }
if (i<j)
{
    swap (A[i], A[j]);
}
}
```

* Merging Arrays -

(Merging can be done only on sorted arrays)




```
i=0; j=0; k=0;
while (i<m && j<n)
{ if (A[i]<B[j])
      c[k++] = A[i++];
  else
      c[k++] = B[j++];
}
for (; i<m; i++)
    c[k++] = A[i];
for (; j<n; j++)
    c[k++]; B[j];
```

time - O(m+n)

08/04/22

Student Challenge

→ Finding single missing element in sorted array

A	1	2	3	4	5	6	8	9	10	11	12
	0	1	2	3	4	5	6	7	8	9	10

Method-1 using n-natural rule-

```
Sum = 0;
for (i=0; i<11; i++)
{
    Sum = Sum + A[i];
}
S = n*(n+1)/2;
```

S - Sum;
78 - 71 = 7

printf("Missing no. is %d", S - Sum);

$S = \frac{n(n+1)}{2}$

$= \frac{12(12+1)}{2}$

$= 78$

method-2 using indices

6	7	8	9	10	11	13	14	15	16	17
0	1	2	3	4	5	6	7	8	9	10

6-0 = 6

6 + 6 = 12
index diff

l = 6;
h = 17;
n = 11;

6 - 0 = 6
7 - 1 = 6
8 - 2 = 6
9 - 3 = 6
10 - 4 = 6
11 - 5 = 6
13 - 6 = 7

```
diff = l - 0;
for (i=0; i<n; i++)
{
    if (A[i] - i != diff)
    {
        printf("Missing element %d", i+diff);
        break;
    }
}
```

Time - O(n)

→ Finding multiple missing elements in a sorted array -

6	7	8	9	11	12	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9	10

diff = 6 6 6 6 7 7 9 9 9 9 9

6+4 = 10

7+6 = 13 8+6 = 14

l = 6
h = 19
n = 11

```
diff = 6-0;
for (i=0; i<n; i++)                    time - O(n)
{ if (A[i] - i != diff)
    while(diff < A[i] - i)
  { printf("%d\n", i + diff);
    diff++;
  }
}
```

09/04/22

→ Find missing element in unsorted array -

A

3	7	4	9	12	6	1	11	2	10
0	1	2	3	4	5	6	7	8	9

n = 10
l = 1

H

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

h = 12

Hash table (Bitset) ↓ it takes constant time

(When have to search, we can apply hashing if possible)

(Whenever we are using hash table and storing the elements then we need an array space equal to the largest element)

Eg:- A - n = 10 (largest element = 12)
h (hash array) - h = 12

```
for (i=0; i<n; i++)
{   H[A[i]]++;          ——— n
}
for (i=l; i<=h; i++)
{                       ——— n
   if (H[i]==0)
   printf("%d", i);
```

time = 2n
O(n)

→* Finding Duplicates in sorted array and count that duplicates:

	0	1	2	3	4	5	6	7	8	9
A	3	6	8	8	10	12	15	15	15	20

It will find the Duplicate elements

```
lastDuplicate = 0;
for (i=0; i<n; i++)
{   if (A[i]==A[i+1] && A[i]!=lastDuplicate)
    {   printf("%d\n", A[i]);
        lastDuplicate = A[i];
    }
}
```

time = O(n)

```
for (i=0; i<n-1; i++)
{   if (A[i]==A[i+1])
    {   j=i+1;
        while (A[j]==A[i]) j++;
        printf("%d is appearing %d times", A[i], j-i);
        i=j-1;
    }
}
```

* Find Duplicates elements in shorted - array using hashing.

Date
Page

A | 3 | 6 | 8 | 8 | 10 | 12 | 15 | 15 | 15 | 20 | n = 10

0 1 2 3 4 5 6 7 8 9

H | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | 1 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 |

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 h: 20

```
for (i = 0; i < n; i++)
    H[A[i]]++;                    } n

for (i = 0; i <= h; i++)          } n
    if (H[i] > 1)
        printf ("%d %d", i, H[i]);
```

time = 2n

O(n)

* Find Duplicates elements in unsorted array and count -

A | 8 | 3 | 6 | 4 | 6 | 5 | 6 | 8 | 2 | 7 | n = 10

```
for (i = 0; i < n-1; i++)                          } n
{   count = 1;
    if (A[i] != -1)
    {   for (j = i+1; j < n; j++)                  } n
        {   if (A[i] == A[j])
            {   count++;
                A[j] = -1;
            }
        }
    if (count > 1) printf ("%d %d", A[i], count);
}
```

time - $O(n^2)$

(loop in loop)

Using hashing

Date
Page