..../..../.....

A program is a set of instructions which performs operations of data. So without data instructions cannot be performed So data is important part of the program. So when a program is dealing with the data will organize the data structures Depending on requrements of a program the type of procedures that is performing we have various data structures

int A[5] Arrays int
int A[3][3] Matrices int
not C we have to create it ← Linked list

} physical data structure → is arranged in the memory (arrangment)

they define how the data (provided every bgyge C, C++, Java).

Stack
Query
Trees
Graph
Hashing

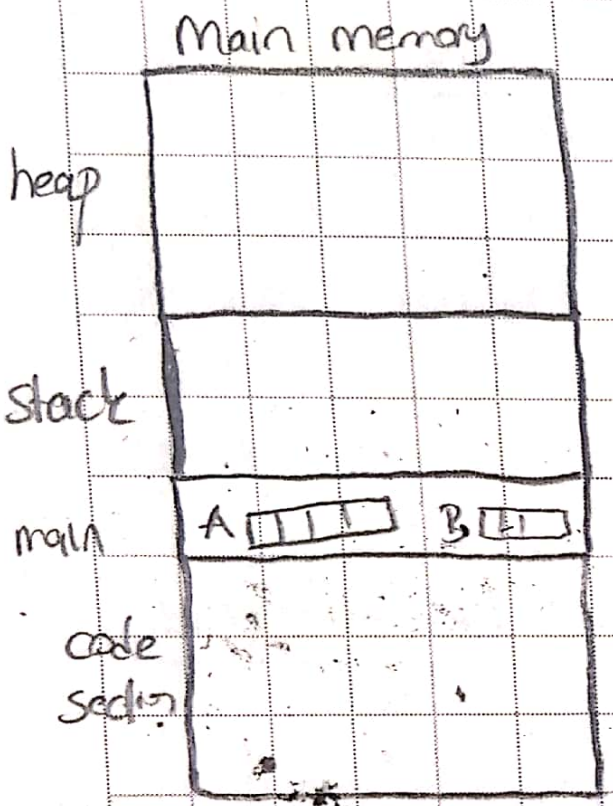
} logical data structures → data can be utilized (utilized)

they define how the

Recursion
Sortings

CamScanner ile tarandı

Arrays: collections of similar data. If you have some set of integers or set of floats you can group them under one name as array



int A[5];

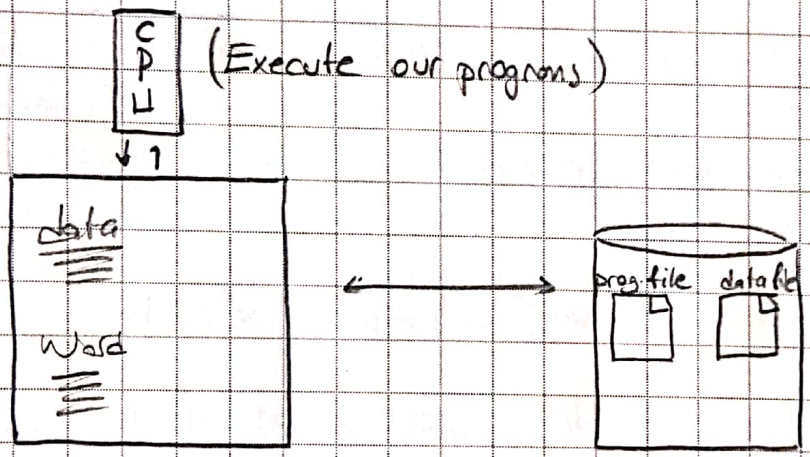
int B[5] = {2, 9, 8, 6, 10};

declaration initialization

CamScanner ile tarandı

→ Data is an integral part of our applications or programs data warehouse, data bases, big data

→ Data Structure arrangment of collection of data items so that can be utilized efficiently, operations on that data can be done efficiently. So during the execution of program, how the program will manage data inside the main memory and perform the operations

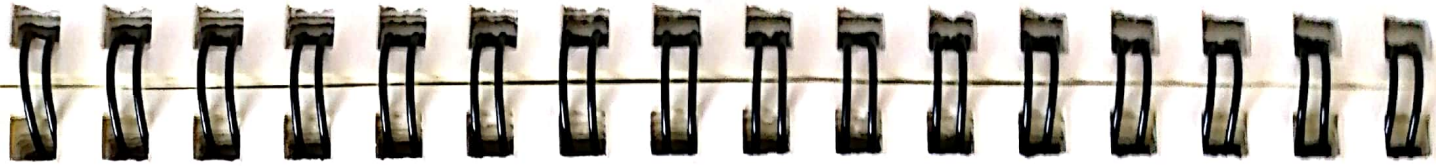


→ Question how it will arrange the data in the main memory for performing its operations. that arrangements is data structure. Data structure is a part of running program

→ Arrangement of data in main memory is <u>data structure</u>

→ How this data organized in form of table on the disk is <u>data base</u>

CamScanner ile tarandı



...../...../.....

Data warehouse is helpful for analyzing the business or making the policies, or starting a new trend or giving offers to customers dealing with the customers So this large size data is data warehouse and the algorithms written for analyzing that data "data mining algorithms".

- Data structure inside the main memory during the execution.
- Data base on the disk(HDD) and large size data
- Data warehouse is huge size data which is inactive now and required it is utilized

Data structue → main memory arrangement of data

Data base → Array of disk arrangement of data

Data warehouse → that is not operational not used day to day

Big data → Interet

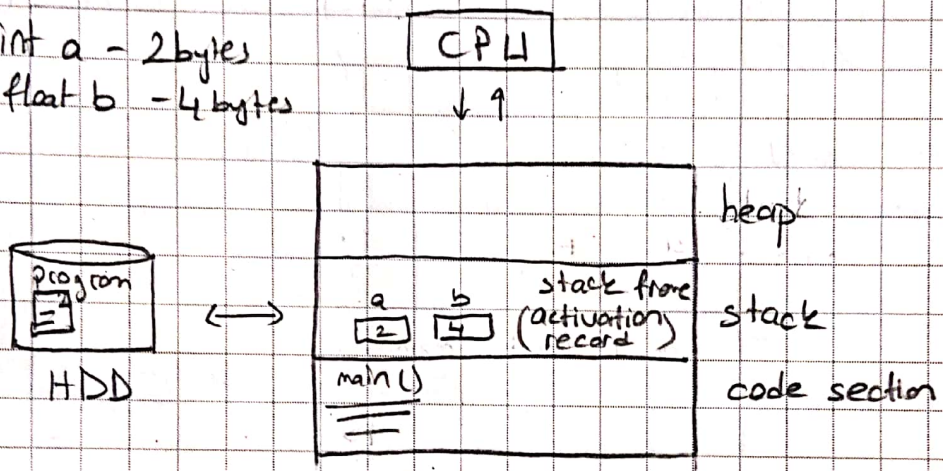
CamScanner ile tarandı

Stack vs Heap Memory

...../...../.....

- Memory is divided into small adreseble units that are called as bytes. Every byte is having adress
- Memory 2 dimensional but adresses are single dimensinan (linear)

int a - 2 bytes
float b - 4 bytes



→ Whetever the variables declere inside your program or function the memory for those variables will be created inside the stack

→ The portion of memory that is given to the function is called as activation record of that function

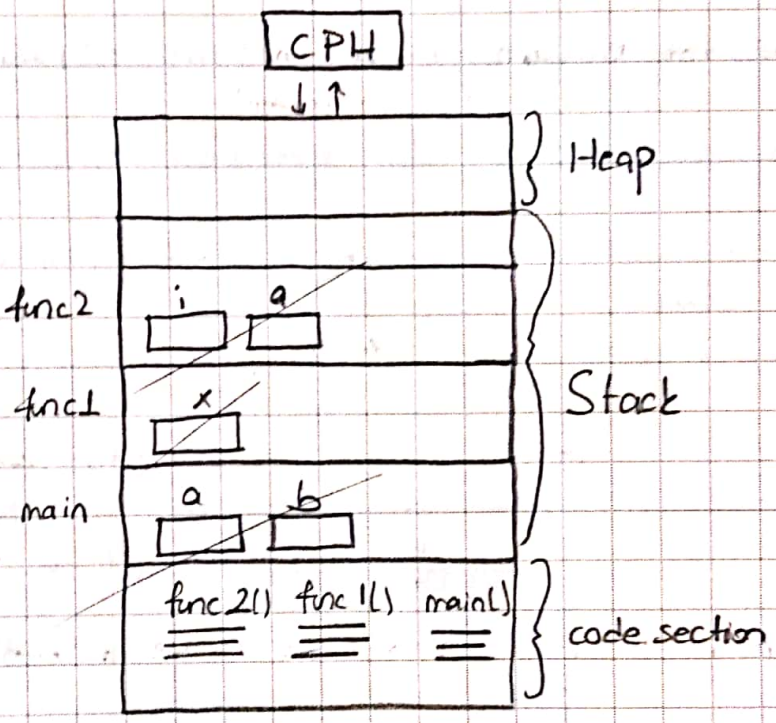
→ How the memory is allocated inside the stack depends the variables inside a function. So size of the memory required by a function was decided at compile time only

Static memory allocation : How may bytes of memory is required by this function was it decided at compile time

CamScanner ile tarandı

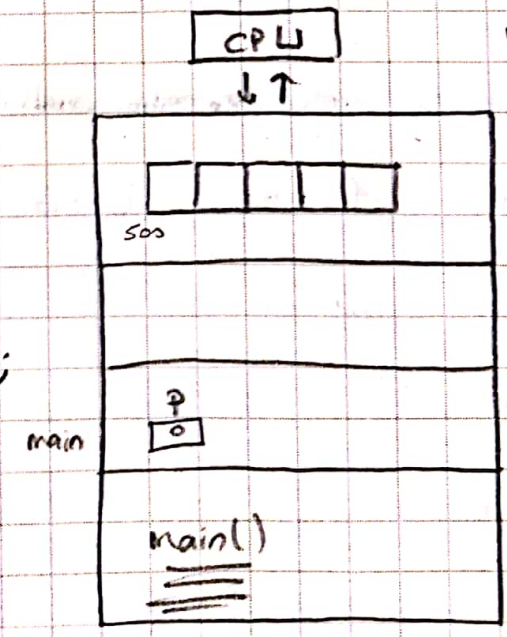
Stack top top top delete delete delete

```
void func2(int i)
  int a
void func1()
  int x;
  fun2(x);
void main()
int a
float b
func1()
}
```



Heap unorganized program cannot directly acses heap memory. How acses ↓ using pointer*

```
void main()
  int *p;        2 bytes
C++ → p=new int[5];
C → p=(int*)malloc(2*5);
  delete []p;
  p=NULL;
```



• Heap memory should be used like a source. When require you take the memory, you don't require you release memo

delete

CamScanner ile tarandı

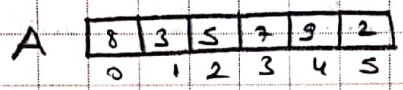
Types of Data Structures

...../...../.....

1- Physical data structures

• These data structures decides or defines how the memory is organized, how the memory allocated.

a) **Array** : collection of contigious memory locations. fixed size (statistic) can be created in stack or heap. If you are sure max lenght use



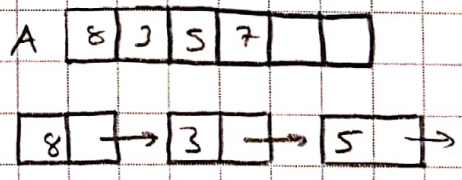
b) **Linked list** : complete dyconik data structure. collection of nodes always created in heap

• The lenght of list can grow and reduce, dyronicly. So its having variable lenght.

2- Logical data structures

• These data structures is dicipline to performing operations like inseting more values or deleting existing values or searching for the values and us.

linear
- a) Stack LIFO (last in first out)
- b) Queues FIFO



non-linear
- c) Trees
- d) Graph

tabular
- e) hash table

• These logical data structures are implemented using any of these physical data structures either array or linked list

CamScanner ile tarandı

Abstract Data Type (ADT)

...../...../.....

1- Representing of Data

2- Operation of Data

Adt defined the data and operations on data together and let it be used as data type, by hiding all the internal details (use OOP)

List → 8, 3, 5, 7, 9, 6, 2, 7

0 1 2 3 4 5 6 7

data : 1- spcace for statiy elements

2- Capasity

3- Size

operations :
- add(x) / append(x) → adding some element end of list
- add(index, x) / insert(index, x)
- remove(index)
- set(index, x) / replace(index, x)
- get(index)
- search(x) / contains(x)
- sort

CamScanner ile tarandı

Recursion

```
void fun2(int n)
{
  if(n>0)
  fun2(n-1)
  print("%d",n)
}
```

```
void main()
{
  int x=3;
  fun2(x);
}
```

fun2(3)
fun2(2) 3
fun2(1) 2
fun2(0) 1
x

output = 1 2 3

1 - Go to next room

2 - Switch on bulb

3, 2, 1

```
void fun1(int n)
{
  if(n>0)
  printf("%d",n);
  fun1(n-1);
}
```

```
void main()
{
  int x=3;
  fun1(x);
}
```

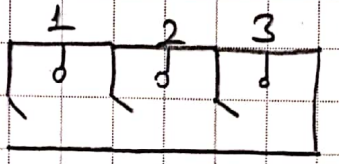
fun1(3)
3 fun1(2)
2 fun1(1)
1 fun1(0)
x

output = 3 2 1

1 - switch on bulb

2 - Go to next room

1, 2, 3




```
void fun(int n)
  if(n>0)
```

Ascending 1 - CALLING

2 - fun(n-1)

Descending 3 - RETURNING

Recursion functions have 2 phases that are ascending and descending

Loops have ascending. Descending only have recursive functi

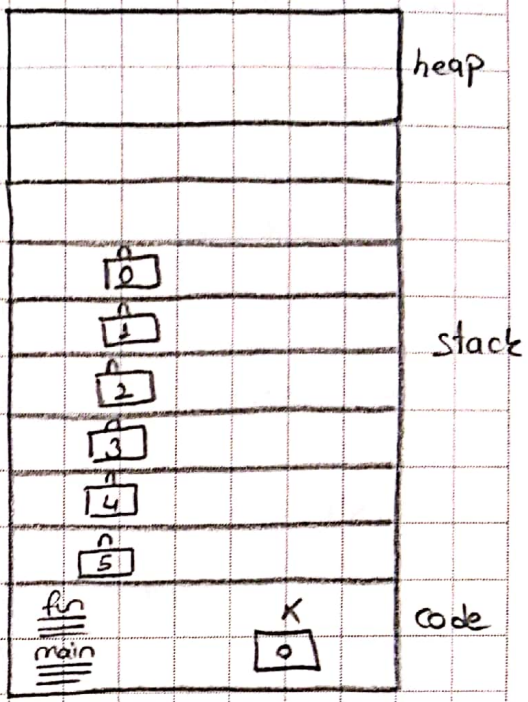
CamScanner ile tarandı

Static Variables in Recursion

...../...../.....

fun(5) = 15
/
fun(4) + 5 = 15
/
fun(3) + 4 = 10
/
fun(2) + 3 = 6
/
fun(1) + 2 = 3
/
fun(0) + 1 = 1
|
0

```
int fun(int n)
{
  if(n>0)
  {
 return fun(n-1)+n;
 }
  return 0;

main()
int a=5;
printf("%d",fun(a));
}
```



fun(5)
/
fun(4) + 5 = 25
/
fun(3) + 5 = 20
/
fun(2) + 5 = 15
/
fun(1) + 5 = 10
/
fun(0) + 5 = 5
|
0

```
int fun(int n)
{
  static int x=0;

    if(n>0)
    {
      x++;
   return fun(n-1)+x;
   {
   return x;
   }

    main
    int a=5;
    printf("%d", fun(a));

    }
```

CamScanner ile tarandı

Types of Recursion

1 - Tail Requsion

2 - Head Reqursion

3 - Tree reqursion

4 - Indirect reqursion

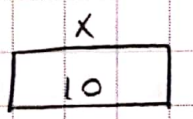
5 - Nested reqursion

CamScanner ile tarandı

Arrays

...../...../.....

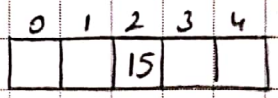
scalar → int x=10;



scalar (having just magnitude)

Array: We can store multiple values that is a list of values or set of values. Shortly collections of elements and all elements are same type.

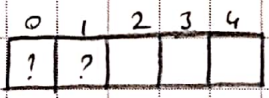
Vector → int A[5];



A[2] = 15;

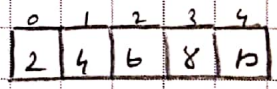
Array decleration

→ int A[5];

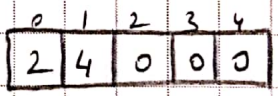


garbage value

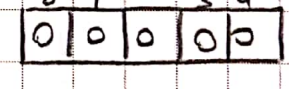
→ int A[5] = {2,4,6,8,10}



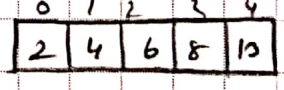
→ int A[5] = {2,4}



→ int A[5] = {0};



→ int A[ ] = {2,4,6,8,10}



Acses of elements

→ for(int i=0; i<5 ; i++):
printf("%d", A[i];

→ printf("%d", A[2];

→ printf("%d", 2[A];

→ printf("%d", *(A+2));

CamScanner ile tarandı

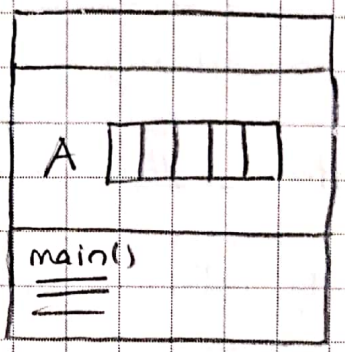
Static vs Dynamic Array

...../...../.....

Static vs dynamic array means size of an array dynamic or static

C

```
void main ()
{
    int A[5];
}
```

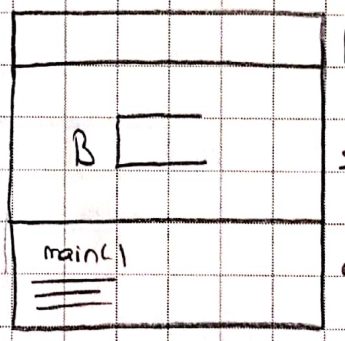


Static array because the size of an array was decided at compile time.

In c langage size of array it must be constant value, it connot be variable

C++

```
void main()
{
    int n;
    cin >> n;
    int B[n];
}
```



In C++ we can create any size. Size of an array will decided at run time

How to creat an array inside the heap?

→ for acsesing anything from heap we must have pointer

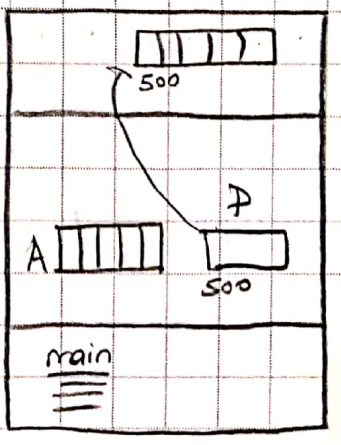
```
void main (){
int A[5];
int *P;
```

C++ → `P = new int[5]`

C → `P = (int x*)malloc (5 * sizeof(int));`

C++ → `delete [] P.`

C → `free (p);`



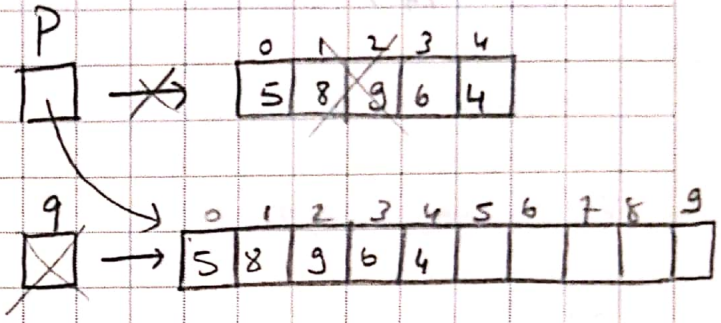
- If you don't delete unused memory, it causes memory leak problem

CamScanner ile tarandı

How to Increase Array size?

..../..../.....

```
int *P = new int[5];
int *q = new int[10]
for(i=0; i<5; i++);
    q[i] = p[i];
    delete [] p;
    p = q;
    q = NULL;
```



→ p have small car. p needs bigger size car.

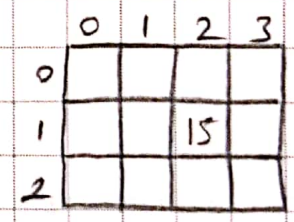
so p purchase a new car. q help in purchasing a new car. and all things transfered new car and old car sold out. P has bigger new car.

2D Array

in stack

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

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



A[1][2] = 15

some are in stack some are in heap

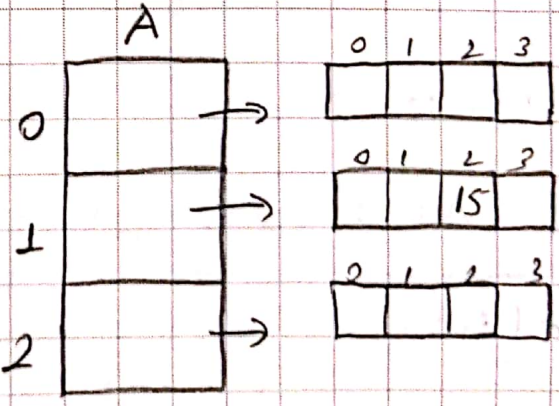
② `int *A[3];`

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

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

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

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

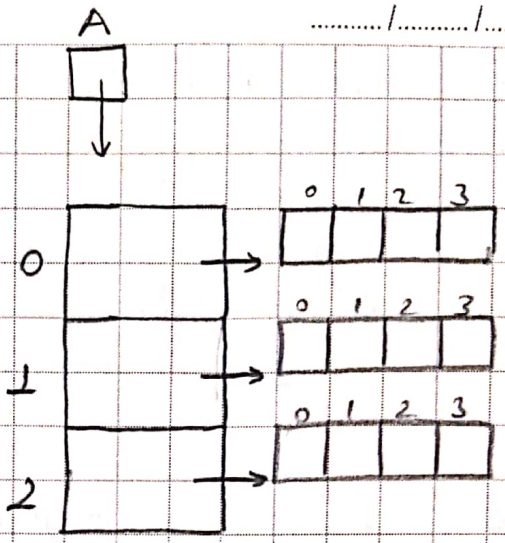


CamScanner ile tarandı

→ completely in heap

③

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



→ We can access 2D arrays with nested loops

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

Arrays in Compiler

```
int A[5] = {3,5,8,4,2};
A[3] = 10;
```

	0	1	2	3	4
A	3	5	8	4	2
	200/1	2/3	4/5	6/7	8/9

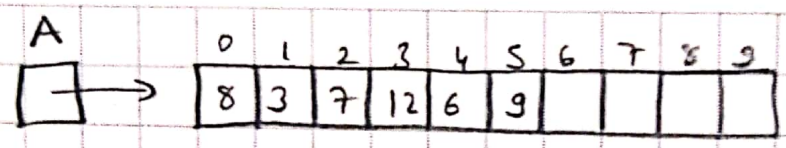
Add(A[3]) = 200 + 3 × 2 = 206

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

L_0: base address, i: index, w: size of data type

CamScanner ile tarandı

Array ADT



size=10
lenght = 6

① int A[10];

② int *A;
A = new int [size];

Display()

```
for(int i=0;i<Lenght;i++){
    print(A[i]); }
```

Insert(index, x)

```
for(i=Lenght; i>index; i--){
A[i] = A[i-1]; }
A[index] = x;
lenght ++;
```

Add(x)/Append(x)

```
A[Lenght] = x;
Lenght ++
```

CamScanner ile tarandı

LINKED LIST

..../..../....

Array vs. Linklist

Array → Dynamic `int *P = new int[5];`

Static `int [5];`

- Fixed size
- insert, delete are inefficient elements are usually shifted
- Memory assign during compile time
- Elements are stored contigiously
- Sequential acces is faster because elements in contigianus memory locations

Link list → Dynamic

- flexible
- Dynamic size in heap
- Efficient no shifting
- Memory assign during execution or runtime
- Elements are stored randomly
- Sequential acces is slow because elements not in contigious in memory locations

Why would you use link list over array?

→ It is easier to store data of different size in a link list. An array assume every element is exactly the same size.

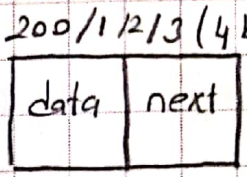
→ It is easier for a link list to grow organicly. An array's size needs to be known before of time or re-created when it needs to grow

→ As long as your iterations all happen in a foreach context, you don't lose any performance in iterations

CamScanner ile tarandı

• Link list is a collection of nodes. Each nodes contain data and pointer to next node.

• Link list created in heap



size of structre = 4byte

```
struct Node
{
    int data;      → 2byte
    struct Node* next;   → 2byte
};
```

C → Structre

C++ → Structre, Class

• self referential structure

difference between class and structre

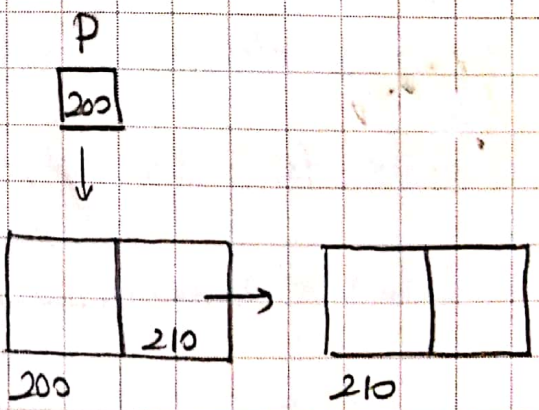
Class → everything by default is private

Structre → everything by default is public

```
struct Node *p
p=new Node;
```

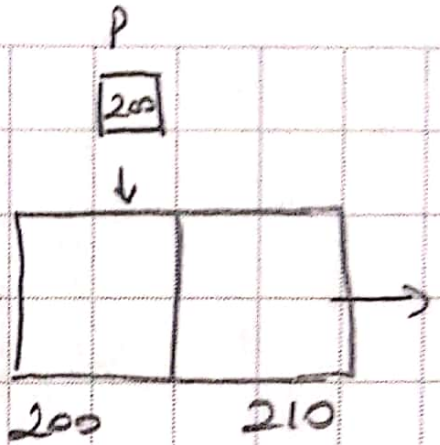
```
struct Node *p = NULL;

if (P==NULL)
if (P==0)
if (!P)
```


```
if (P!=NULL)
if (P!=0)
if (P)
```



CamScanner ile tarandı

if (p → next == NULL)



p is a last node

if (p → next != NULL) → p is continious we have orather nodes

CamScanner ile tarandı

STACK

Stack ADT

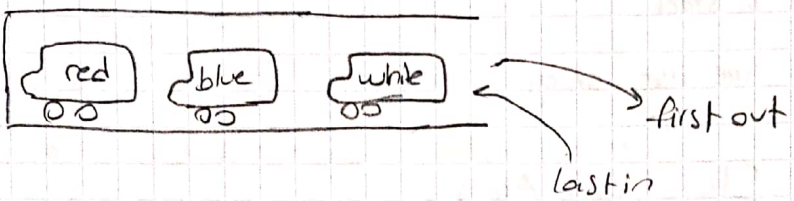
Stack is a collection of elements set of values collection of a elements. and those collection of elements they are insetted, deleted by following this dicipline

En son eklenen ilönce silinir

LIFO: Last in first out

Recursion are the fnctions which call themselves like loop

Recursive use stack



Recursion ← Iteration

Recursion uses stack system

Every recursion can be converted into recursion also every recursion can be converted into iteration then you can work recursion into iteration

→ If you have a known precedure recursion and you want to convert iteration meyle you required stack.

Recursion uses automatic system stack doesn't need you do it compiler take care of it but when you are recursion converting into iteration progremmer should create stack and implement the stack in the program

→ If you have creative precedure and you are using a stack means it can be done better using recursion.

Stack Adt

ADT

→ Abstrac data type contain data representation and the operations on the stack

Adt of Stack

data :

→ Space for storing the elements : Stack is a collection of elements so we need space for storing both collection of elements

→ Top pointer: pointing on the topmost element in the stack

because the mpotant one of is topmost element which one was recently inserted in

CamScanner ile tarandı

Operations

1- push(x) → inserting a value (not all value)
(if you want more than 1 value call push() more than 1 times)

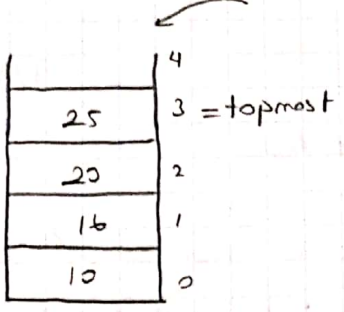
2- pop() → deleting a value

3- peek(index) → looking at the value

4- StackTop() → knowing the topmost value : 25

5- isEmpty() → Stack is empty?

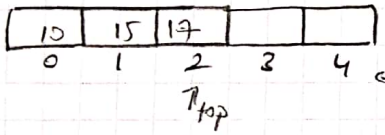
6- isFull() → Stack is full?



→ We have to same type in stack.

Implementing a stack we use array or linkedlist

Stack using Array



size : 5 fixed size

O(1) constant

3 things we need

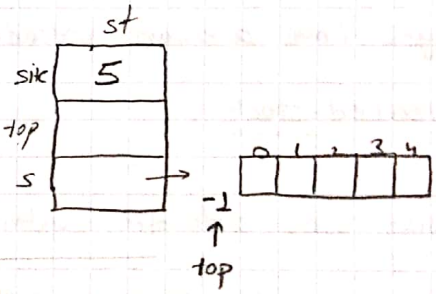
1- array

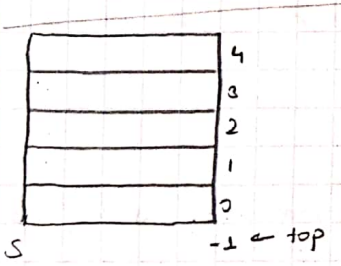
2- size

3- top pointer

```
struct stack
{
  int size;
  int top;
  int *s;
```

```
int main()
{
  struct stack st;
  printf("enter the size of stack");
  scanf("%d", &st.size);
  st.s = new int [st.size];
  st.top = -1;
```





Empty

```
if (top == -1)
```

Full

```
if (top == size-1)
```

→ before insertation we must check the stack is full or not. if it is full it cannot be inset any element. if it is not full we can increament of pointer and insert an element in array.

push

```
void push(stack *st, int x)
  if (st->top == st->size-1)
    printf("stack overflow");
  else
  {  st->top++
     st->s[st->top] = x;
  }
```

time taken constant

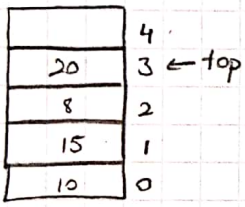
CamScanner ile tarandı

before the delation we should check are there any elements present in the stack or not

```
int pop (stack * st)                    time taken constant
{
   int x = -1;
   if (st -> top == -1)
      print ("stack underflow")
   else
   {  x = st -> s [st -> top];
      st -> top --
   }
   return x;
}
```

pop

→ Time taken by push and pop operation is constant.



position | index = top - position - 1

position	index
1	3 = 3 - 1 + 1
2	2 = 3 - 2 + 1
3	1 = 3 - 3 + 1
4	0 = 3 - 4 + 1

```
peek (stack st, int pos)
{   int x = -1;
   if (top - pos + 1 < 0)              time taken constant
      printf ("invalid position")
   else
      x = st.s [st.top - pos + 1];
   return x;
}
```

peek

```
int stacktop (stack st)
{
 if (st.top == -1)
 return -1;
 else
    return st.s [st.top];
}
```

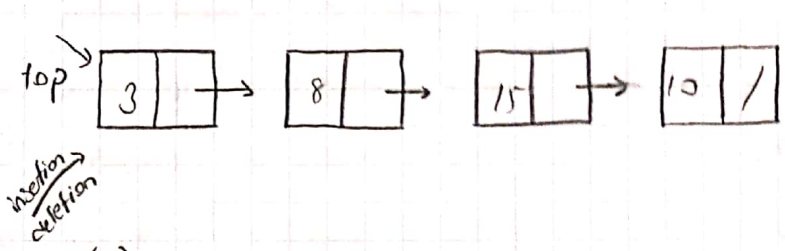
```
int isEmpty (stack st)
{
  if (st.top == -1)
   return 1; (true)
else
   return 0; (false)
}
```

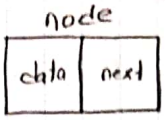
```
int isFull (stack st)
{
   if (st.top = st.size -1)
    return 1; (true)
    else
     return 0; (false)
}
```

CamScanner ile tarandı

Stack using Link List

linked list is a collection of nodes. which have list of elements. Stack is also collection of elements but insertion and delations is done by using LIFO.






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

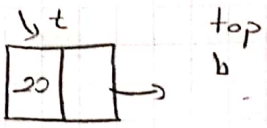
Empty

```
if (top == NULL)
```

Full

```
Node * t = New Node;
if ( t == NULL)
```

```
void push (int x)
  Node *t = new Node;
  if (t == NULL)
      print (" stack overflow");
  else
      t -> data = x;
      t -> next = top;
      top = t;
```




```
int pop( )
{
  Node *p;
  int x = -1;
  if (top == NULL)
    printf ("Stack is empty")
  else
      p = top;
      top = top -> next
      x = p -> data;
      free (p);
  }
      return x;
  }
```

```
int peek(int pos)
{
  int i;
  Node * p = top
  for( i = 0; p != NULL && i < pos-1; i++)
    p = p -> next;
  }
    if (P! = NULL)
      return p -> data;
    else
      return -1;
```

```
int stacktop( )
{
  if(top)
    return top -> data;
    return -1;
```

```
int isEmpty( )
{
   return top ? 0 : 1;
{
```

```
int isFULL( )
{
 Node * t = new Node;
int x = t ? 1 : 0;
free (t);
return x;
}
```

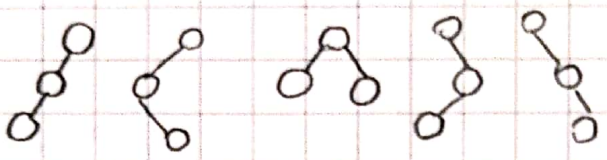
CamScanner ile tarandı

Trees

Number of binary trees

combination formula

$n=3$



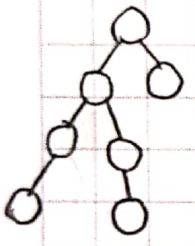
$T(3)=5$

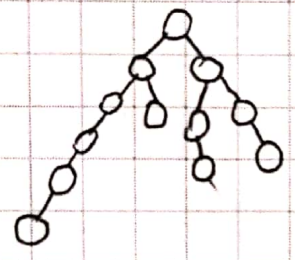
$T(n) = \frac{^{2n}C_n}{n+1}$ = catalan number

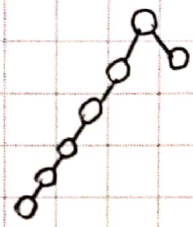
Internal Nodes & External (leaf) nodes

$T(n) = \sum_{i=1}^{n} T(i-1)\,T(n-i)$

recursive formula







deg (2) = 2	deg (2) = 3	deg (2) = 1
deg (1) = 2	deg (1) = 5	deg (1) = 4
deg (0) = 3	deg (0) = 4	deg (0) = 2

→ $\boxed{deg(0) = deg(2) + 1}$

Strict binary tree $\{0, 1, 2\}$ (0 ✓, 1 ✗, 2 ✓)

if height is given h

Min nodes $n = 2h+1$

Max nodes $n = 2^{h+1}-1$

if n nodes are given

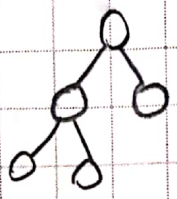
Min height $h = \log_2(n+1)-1$

Max height $h = \frac{n-1}{2}$

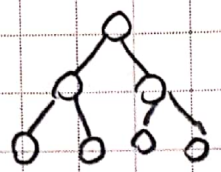
$$\log_2(n+1)-1 \leq h \leq \frac{n-1}{2}$$

CamScanner ile tarandı

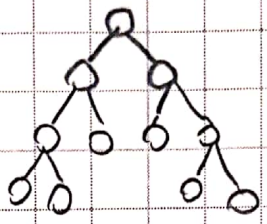
Strict binary tree
Internal x extend nodes



$i=2$
$e=3$



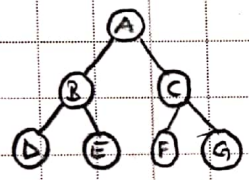
$i=3$
$e=4$



$i=5$
$e=6$

$$e = i + 1$$

Array Representation of Binary Tree



element	index	Leftchild	Rightchild
A	1	2	3
B	2	4	5

A	B	C	D	E	F	6
1	2	3	4	5	6	7

heap-stack → it can be static-dynamic

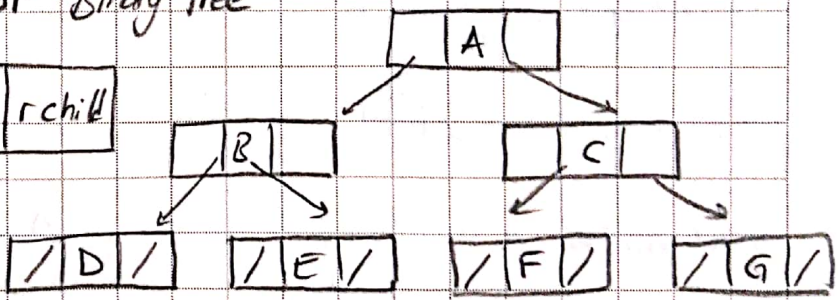
- element i
- leftchild $2i$
- rightchild $2i+1$
- parent $i/2$

Link Representation of Binary tree

- doubly linked list

lchild	data	rchild

definitly heap → dynamic



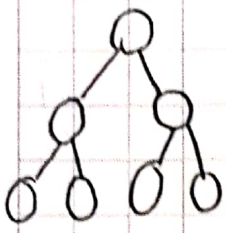
$n=7$

$n+1$ → null pointer sone

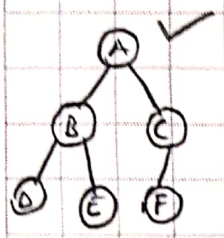
$e = i + 1$

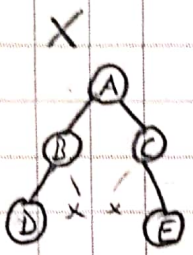
CamScanner ile tarandı

Full vs Complete Binary Tree



Full = each heigh have max numbe of nodes
(herhangi bir yere node eklemeye calistigimizda heigh yükseliyorsa)





Complete = when representing in array this should not blank spaces between elem

Complete ✓

A	B	C	D	E	F

Not complete X

A	B	C	D	-	-	E

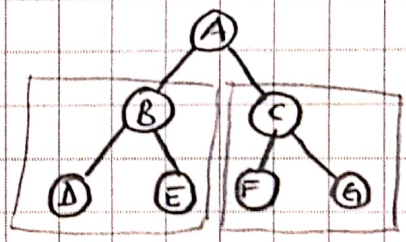
Tree Traversal

Preorder = Root, left, right

Inorder = left, root, right

Postorder = left, right, root

Level order = level by level



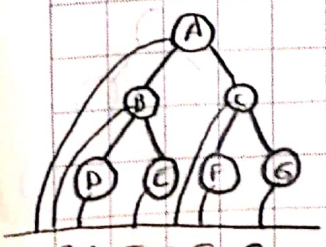
pre = A (BDE) (CFG)
A, B, D, E, C, F, G

inorder = (D, B, E) A (F, C, G)
D B E A F C G

post = (D, E, B) (F, C, G), A
D, E, F, C, G, A

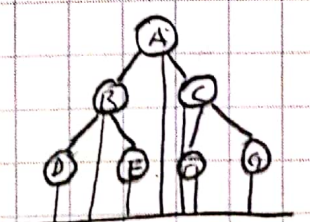
level = A, B, C, D, E, F, G

preorder



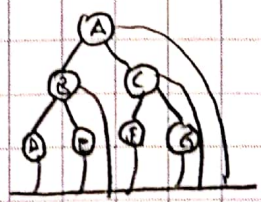
A B D E C F G

inorder



D B E A C F G

postorder

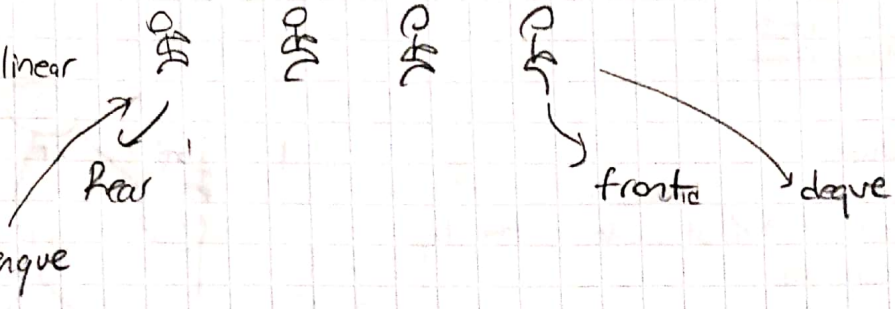


D E B F G C A

CamScanner ile tarandı

Ques

- logical data structures, linear
- FIFO = first in first out



Queue ADT

Data

1- Space for storing elements
2- Front - for deletion
3 - Rear - insertion

Operations

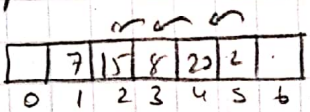
1 - enque
2 - deque
3 - isEmpty
4 - isfull
5 - first()
6 - last()

Queue can be Implemented

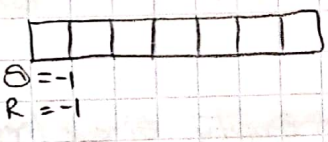
1- Array
2- LinkedList

Ques Using Array

size = 7 fix



Insert - O(1)
delete - O(n)
→ * one pointer



insert O(1)
delet O(1)

** 2 pointers

empty

if(front=rear)

initially

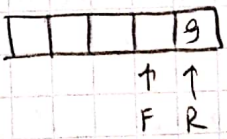
front=rear=-1

full

if (Rear = size-1)

Drawback

1- We can only use the spaces of deleted element
2- Every location can be used only once



Binary tree

n node n-1 actions

Root

CamScanner ile tarandı

SORTING TECHNISHES

Criteria for analysis

- Number of comparison : decide time complexity
- Number of swaps
- Adoptive : if any method is take less time over all the sorted list
- Stable : if sorting algorithm is preseving the order of duplicate elements in sorted list algorithm stable
- Extra memory : some tekniks need extra spaces

Buble Sort, adaptive ✓ stable ✓

A: | 8 | 5 | 7 | 3 | 2 | — max element

0 1 2 3 4

1st pass

8	5	5	5	5
5	8	7	7	7
7	7	8	3	3
3	3	3	8	2
2	2	2	2	8

4 compare 4 swap

2st pass

5	5	5	5
7	7	3	3
3	3	7	2
2	2	2	7
8	8	8	8

3 compare 3 swap

3sth pass

5	3	3
3	5	2
2	2	5
7	7	7
8	8	8

2compare 2 swap

4sth pass

3	2
2	3
5	5
7	7
8	8

1compare 1swap

n-1 passes
Comparison = $O(n^2)$
Swap = $O(n^2)$

- Comparison based sort
  - Buble, Insertion, Selection → $O(n^2)$
  - Heap sort, Merge sort, Quick sort, Tree sort → $O(n \log n)$
  - Shell sort → $O(n^{3/2})$
- Index based sort
  - Count sort, Radix sort → $O(n)$
    - ↳ they are faster but space consumption small

```
void bublesort (int A[], int n) {

  int flag = 0;
  for(i=0; i<n-1; i++) {
  for (j=0 ; j<n-i ; i++) {
    if ( A[j] > A[j+1]){
      swap (&A[j] , &A[j+1]);
     flag=1;
  }
}
   if (flag == 0) {
    return;
}
}
}
```

```
int main() {
int A[] = {3,7,9,10,5,4,8};
int n = sizeof (A) / size of (A[0]
print (A,n)
bublesort (A,n)
print (A,n, "Sorted A");
return 0;
}
```

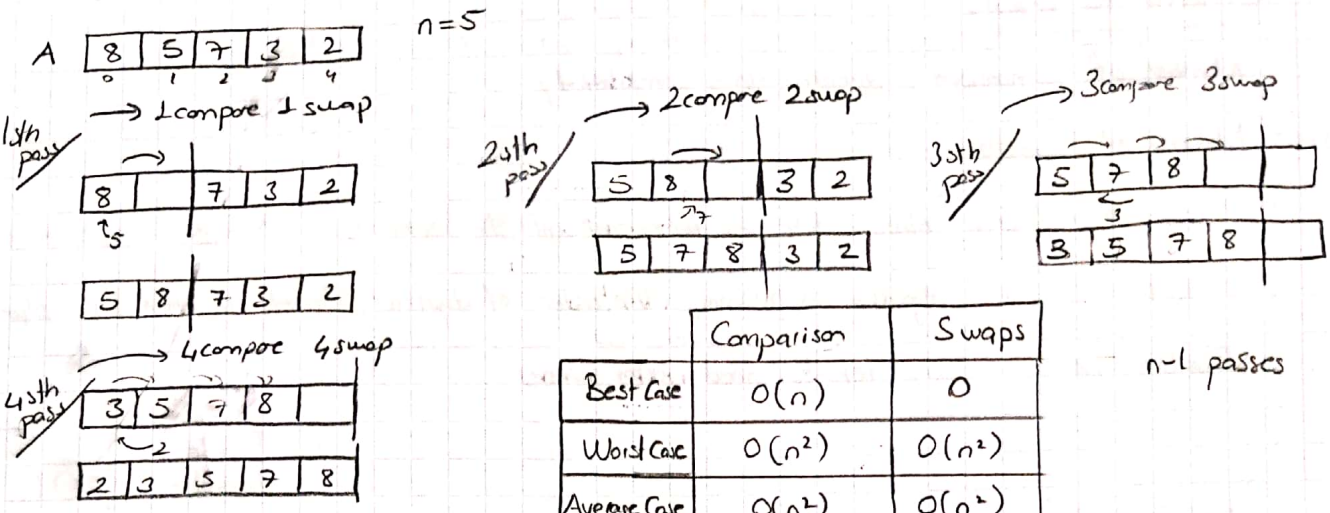
	Comparison	Swaps
Best case	$O(n^2)$	0
Worst case	$O(n^2)$	$O(n^2)$
Avarege case	$O(n^2)$	$O(n^2)$

* Buble ismi suya bas atildiginda yuzu cikan balonculer gibi yavas yavas yer degistiren dizi

CamScanner ile tarandı

Insertion Sorts

Adoptive ✓ stable ✓



	Comparison	Swaps
Best Case	O(n)	0
Worst Case	$O(n^2)$	$O(n^2)$
Average Case	$O(n^2)$	$O(n^2)$

n-1 passes

→ In array we must shift elements but link list we don't have shift anything.

So insertion sort more useful with linklisted than array.

```
void InsertionSort (int A[], int n) {
  for(i=1 ; i<n , i++) {
    int j = i-1
    int x = A[i]
    while( j>-1  && A[j] >x ) {
      A[j+1] = A[j];
      j--;
    }
    A[j+1] = x;
```

```
int main() {
int A[] = {10,12,8,9,5,11,8,3};
print(A, sizeof(A)/sizeof(A[0]);
 InsertionSort (A, sizeof(A)/sizeof(A[0]));
print(A, sizeof(A)/sizeof(A[0]), "Sorted A");
return 0;
```

	Buble Sort	Insertion Sort
Min comparison	O(n)	O(n)
Max comparison	$O(n^2)$	$O(n^2)$
Min swap	O(1)	O(1)
Max swap	$O(n^2)$	$O(n^2)$
Adoptive	✓	✓
Stable	✓	✓
Linked list	×	✓
k passes	✓	×

Already in ascending

" " descending

CamScanner ile tarandı

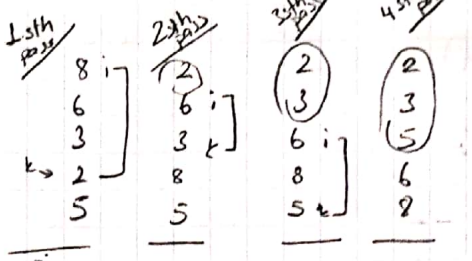
Selection Sort Stable X adaptive X

minimum swap

min element

A

8	6	3	2	5
0	1	2	3	4



comp 4 swap 1 | comp 3 swap 1 | comp 2 swap 1 | comp 1 swap 1

	Comparison	Swap
Best case	$O(n^2)$	$O(n)$
Worst case	$O(n^2)$	$O(n)$
Average Case	$O(n^2)$	$O(n)$

```
Void selectionSort (int A[], int n) {
  for(i=0 ; i<n-1 ; i++) {
  int j;
  int k;
  for( j=k=i ; j<n ; j++) {
    if (A[j] < A[k]) {
        k=j
    }
    } swap(&A[i], &A[k]);
```

(Exchange Sorts)

		Insertion	Buble	Selection
Comparison	Best	$O(n)$	$O(n^2)$	$O(n^2)$
	Average	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Worst	$O(n^2)$	$O(n^2)$	$O(n^2)$
Swap	Best	0	0	$O(n)$
	Average	$O(n^2)$	$O(n^2)$	$O(n)$
	Worst	$O(n^2)$	$O(n^2)$	$O(n)$

Shell Sort

A

3	5	8	11	13	16	18	2	7	9	15
0	1	2	3	4	5	6	7	8	9	10

$gap = \frac{n}{2}$ $gap = \frac{11}{2} = 5$ $gap = \frac{5}{2} = 2$ $gap = \frac{2}{2} = 1$

→ Exploits the best case performace of insetion Sort

→ Strategy = make list mostly seted, final insetion sort can complete soting

→ limited shifting few element shifting

```
//code similar to insetion sort with some modifications
  void ShellSort (int A[], int n) {
   for( int gap = n/2 ; gap >= 1; gap /= 2) {
   for (int j = gap ; j < n; j++ ) {
     int temp = A[j];
     int i = j - gap;
     while( i >=0 ; && A[i] > temp) {
     A[i+gap] = A[i];
       i = i - gap;
   } A[i + gap] = temp ;
```

CamScanner ile tarandı

Merge Sort

stable ✓

```
void Merge(int x[], int y[], int z[], int m, int n)
int i, j, k;
i = j = 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];
```

i →

0	2
1	10
2	18
3	20
4	23

j →

0	4
1	9
2	19
3	25

k →

0	2
1	4
2	9
3	10
4	18
5	19
6	20
7	23
8	25

→ Require extra spaces

→ $n \log n$

→ divide and conquer

→ requires extra space

Quick Sort

pivot

(50) 70 60 90 40 80 10 20 30
i ——— j

50 30 60 90 40 80 10 20 70
i ——— j

50 30 20 90 40 80 10 60 70
i ——— j

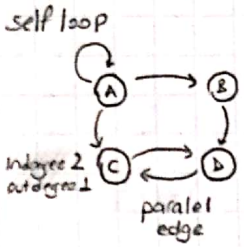
(50) 30 20 10 40 80 90 60 70
j i

(10 20 30 10) 50 (80 90 60 70)

CamScanner ile tarandı

GRAPHS

$G(V, E)$



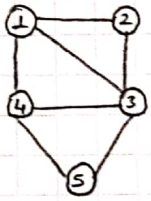
Directed graph : if edges are having direction

Simple graph : without self loop and parallel edge

Strongly connected : from every vertices we can reach all other vertices . (there is a path between every vertex

Cycle : starting from one vertices and ending at the same vertices

Adjacency Matrix



$G = (V, E)$

$|V| = 5$

$|E| = 7$

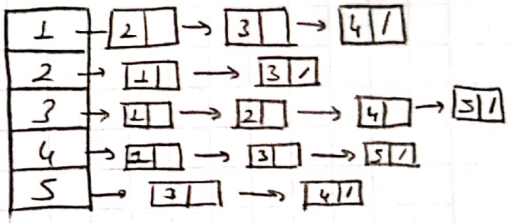
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	0
3	1	1	0	1	1
4	1	0	1	0	1
5	0	0	1	1	0

$n \times n$

$O(n^2)$

time complexity : depend on vertices $O(n^2)$

Adjacency List



$|V| + 2|E|$

time complexity: depend on vertices and edge

$O(n)\ |V| + 2|E|$

CamScanner ile tarandı