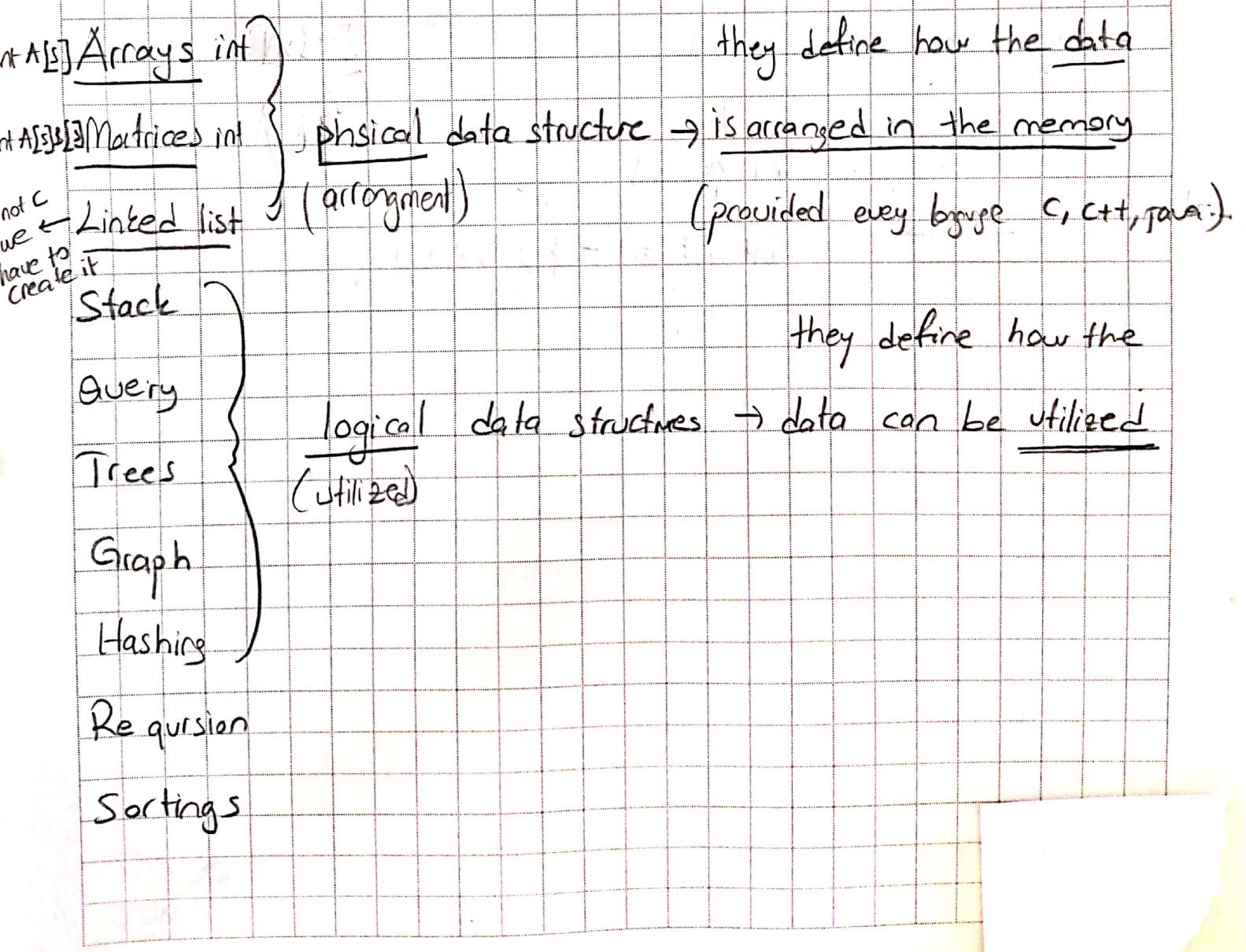
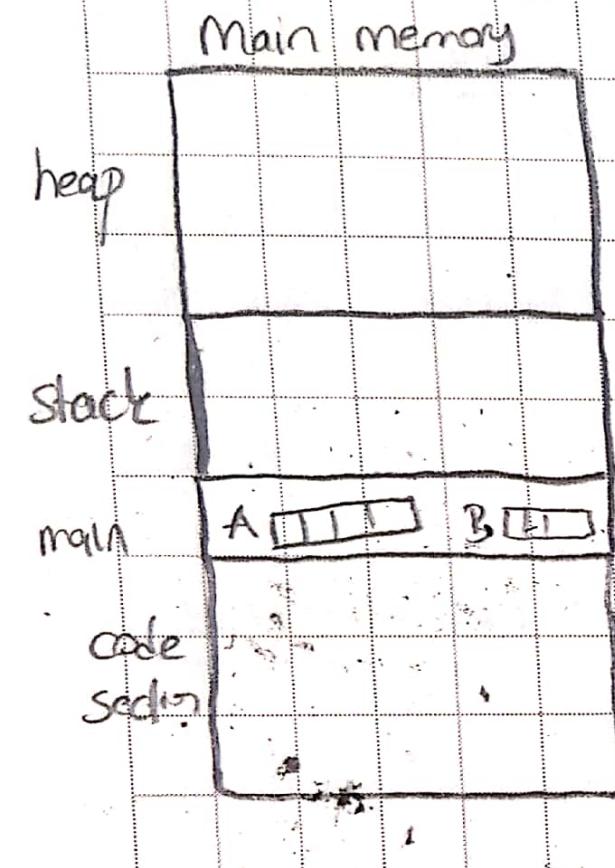


A program is a set of instructions which performs operations of data. So without data instructions cannot be performed. So data is important part of the program. So when a program is dealing with the data will organize the data structures. Depending on requirements of a program the type of procedures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



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



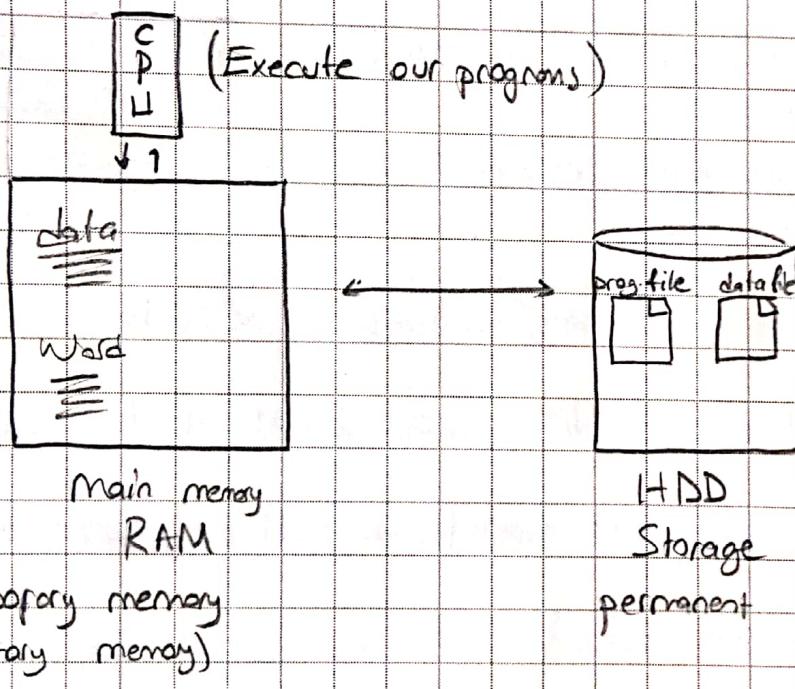
int A [5];

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

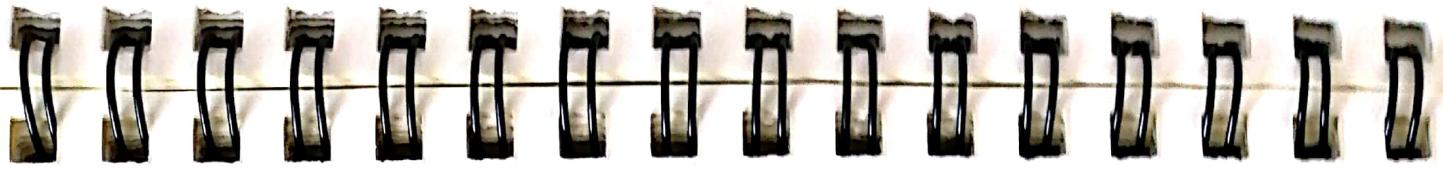
declaration

initialization

- Data is an integral part of our applications or programs
 data warehouse, databases, big data
- Data Structure arrangement 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 data structure
- How this data organized in form of table on the disk is data base



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 structure → Main memory arrangement of data

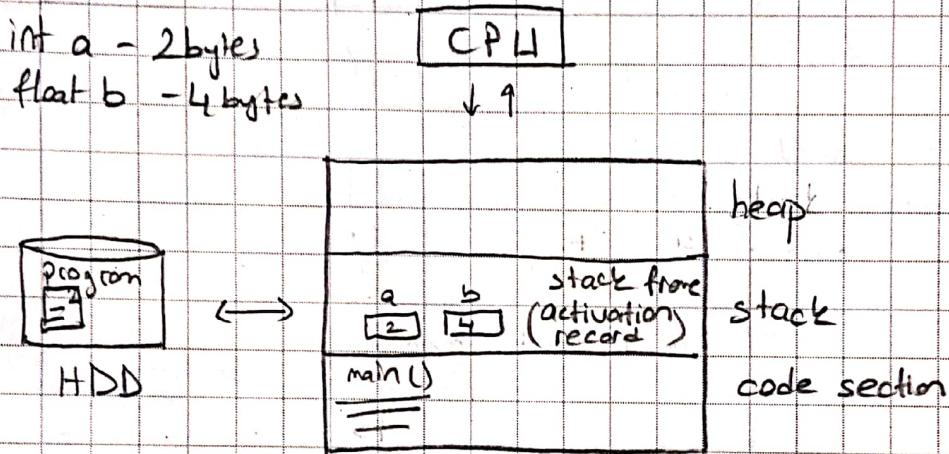
Data base → Array of disk arrangement of data

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

Big data → Internet

Stack vs Heap Memory

- Memory is divided into small addressable units that are called as bytes. Every byte is having address
- Memory is 2 dimensional but addresses are single dimension (linear)



- Whatever the variables declared 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: It has many bytes of memory

is required by this function was it decided at compile time

Stack top top top delete delete delete

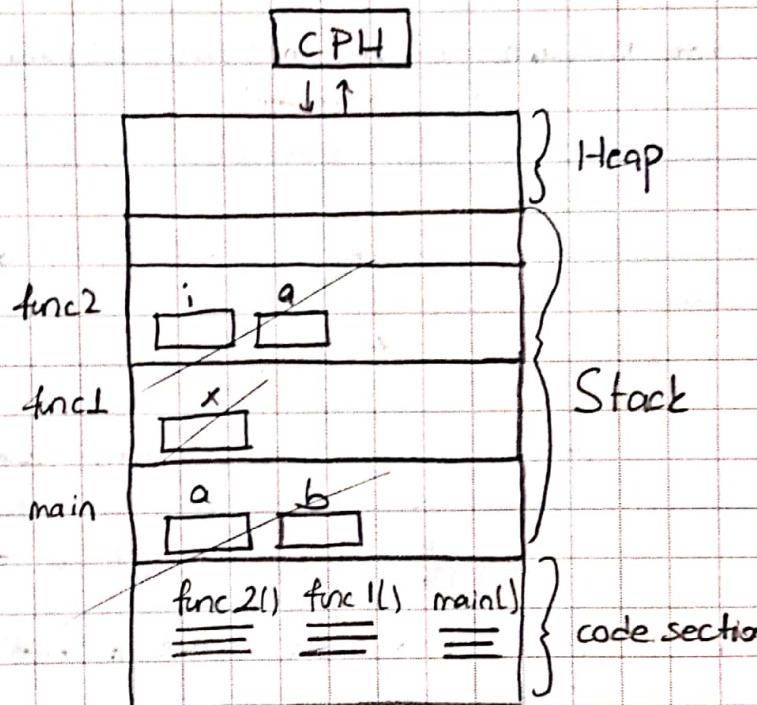
void func2 (int i)

int a

void func1 ()
int x;
func2 (x);

void main ()
int a
float b
func1 ();

}



Heap unorganized program cannot directly access heap memory. How access

↓
using pointer *

void main()

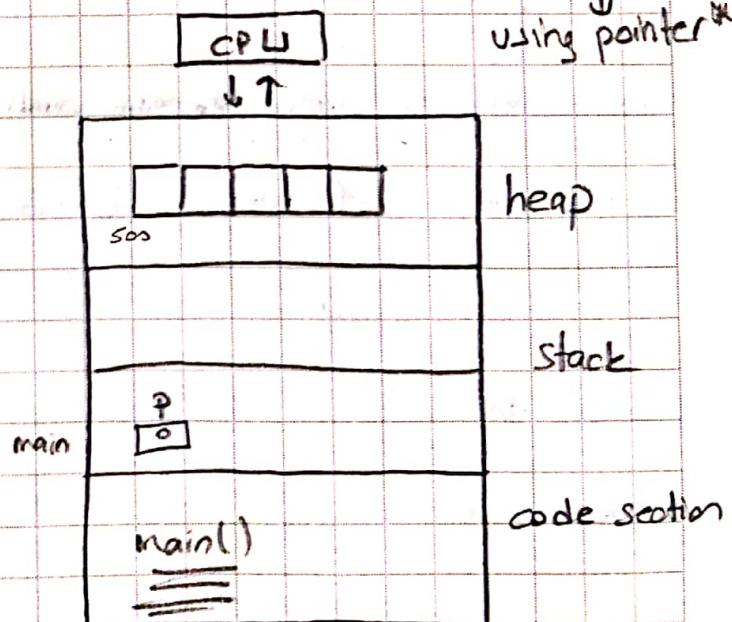
int * p;

C++ → p = new int[5];

C → p = (int *) malloc (2 * s);

delete [] p;

p = NULL;



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

delete

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 contiguous memory locations . fixed size (static)
can be created in stack or heap. If you are sure max lenght use

A	8	3	5	7	9	2
	0	1	2	3	4	5

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



- The lenght of list can grow and reduce , dynamically - So its having variable lenght.

2 - Logical data structures

- These data structures is discipline to performing operations like inserting more values or deleting existing values or searching for the values and us.

linear
a) Stack LIFO (last in first out)

A	8	3	5	7		
---	---	---	---	---	--	--

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

Abstract Data Type (ADT)

1- Representing 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)

2- Operation of Data

List → 8, 3, 5, 7, 9, 6, 2, 7
0 1 2 3 4 5 6 7

data : 1- space for storing elements

2- Capacity

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)

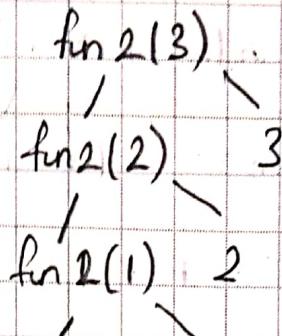
• search(x) / contains(x)

• sort

Recursion

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

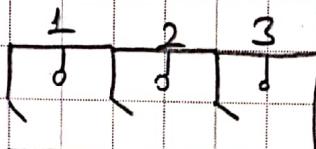
```
void main()
{
    int x=3;
    fun2(x);
}
```



output = 1 2 3

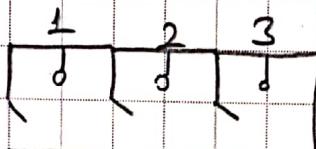
```
void fun1(int n)
{
    if(n > 0)
        printf("%d", n)
    fun1(n-1);
}
```

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



output = 3 2 1

1 - Go to next room



1 - switch on bulb

2 - Switch on bulb

3, 2, 1

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

Static Variables in Recursion

..... / /

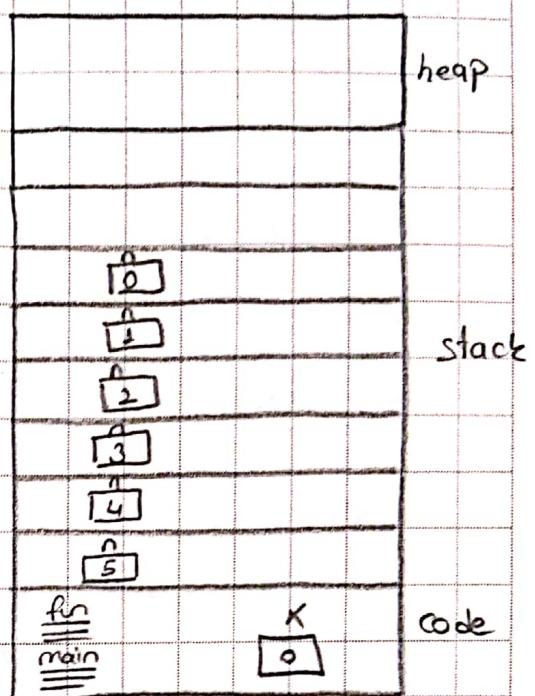
$$\begin{aligned}
 & \text{fun}(5) = 15 \\
 / \\
 & \text{fun}(4) + \underline{5} = 15 \\
 / \\
 & \text{fun}(3) + \underline{4} = 10 \\
 / \\
 & \text{fun}(2) + \underline{3} = 6 \\
 / \\
 & \text{fun}(1) + \underline{2} = 3 \\
 / \\
 & \text{fun}(0) + \underline{1} = 1 \\
 | \\
 & 0
 \end{aligned}$$

```

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

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

```



$$\begin{aligned}
 & \text{fun}(5) \\
 / \\
 & \text{fun}(4) + \underline{5} = 25 \\
 / \\
 & \text{fun}(3) + \underline{5} = 20 \\
 / \\
 & \text{fun}(2) + \underline{5} = 15 \\
 / \\
 & \text{fun}(1) + \underline{5} = 10 \\
 / \\
 & \text{fun}(0) + \underline{5} = 5 \\
 | \\
 & 0
 \end{aligned}$$

```

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

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

```

Types of Recursion

1 - Tail Recursion

2 - Head Recursion

3 - Tree recursion

4 - Indirect recursion

5 - Nested recursion

Arrays

scalar \rightarrow int $x = 10;$

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 some type.

vector \rightarrow int $A[5];$ A

0	1	2	3	4
---	---	---	---	---

 $A[2] = 15;$

Array declaration

\rightarrow int $A[5];$

0	1	2	3	4
?	?	?	?	?

garbage value

\rightarrow int $A[5] = \{2, 4, 6, 8, 10\}$

0	1	2	3	4
2	4	6	8	10

\rightarrow int $A[5] = \{2, 4\}$

0	1	2	3	4
2	4	0	0	0

\rightarrow int $A[5] = \{0\};$

0	1	2	3	4
0	0	0	0	0

\rightarrow int $A[5] = \{2, 4, 6, 8, 10\}$

0	1	2	3	4
2	4	6	8	10

Access of elements

\rightarrow for($i = 0; i < 5; i++$)
 printf("%d, A[i];

\rightarrow printf("%d", A[2]);

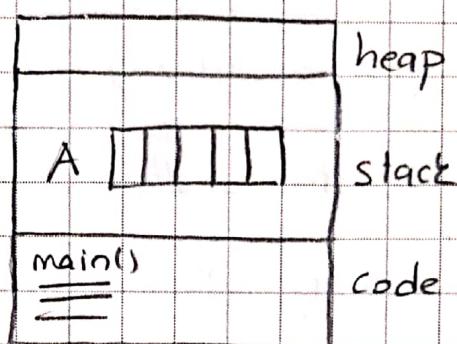
\rightarrow printf("%d", 2[A]);

\rightarrow printf("%d", *(A+2));

Static vs Dynamic Array

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

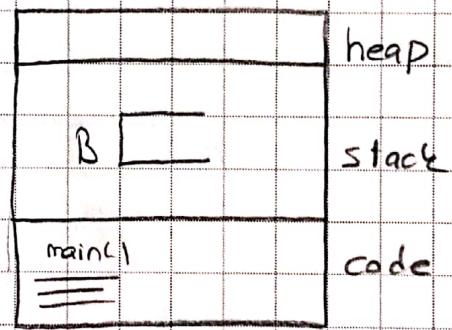
```
void main ()  
{  
    int A[5];
```



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

In c language size of array it must be constant value, it cannot be variable

```
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 create an array inside the heap?

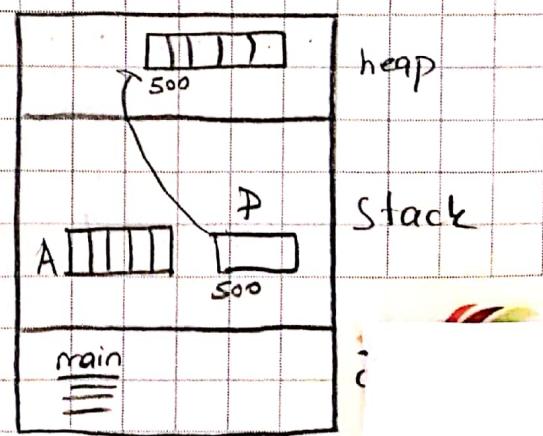
→ for accessing anything from heap we must have pointer

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

C++ → P = new int [5]
C → P = (int *)malloc(5 * sizeof(int));

C++ → delete [] P.

C → free (p) ;



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

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 have small car. p needs bigger size car.

`p = q;`

so p purchase a new car - q help in purchasing

`q = NULL;`

a new car. and all things transferred new car

and old car sold out. P has bigger new car.

in stack

2D Array

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

0	1	2	3
1			15
2			

`A[1][2] = 15`

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

some are in stack some are in heap

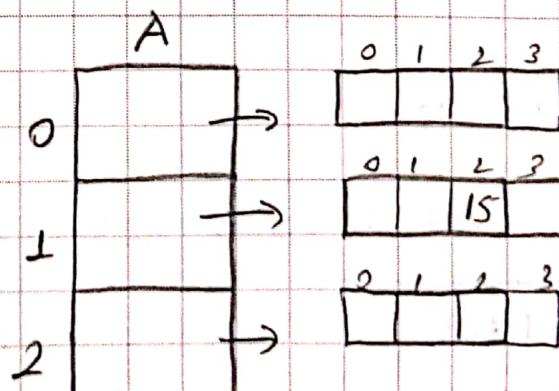
2) `int *A[3];`

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

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

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

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



→ completely in heap

③

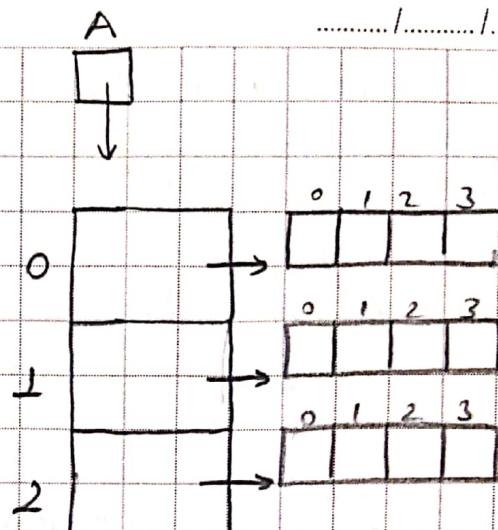
int **A;

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};

0	1	2	3	4
3	5	8	4	2

20011 213 415 617 819

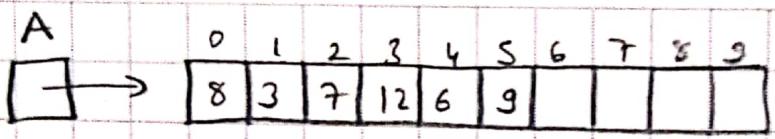
A[3] = 10;

$$\text{Add}(A[3]) = 200 + 3 \times 2 = 206$$

$$\boxed{\text{Add}(A[i]) = \underline{\underline{\text{base address}}} + i * \underline{\underline{\text{size of data type}}}}$$

base address index

Array ADT



① int A[10];

② int *A;
A = new int [size];

Display()

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

Add(x)/Append(x)

A[length] = x;

Insert(index, x)

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

LINKED LIST

Array vs. Linklist

Array → Dynamic $\text{int } *P = \text{new int [5];}$

↓
Static int [5];

- Fixed size

- insert, delete are inefficient
elements are usually shifted

- Memory assign during compile time

- Elements are stored contiguously

- Sequential acces is faster because
elements in contiguous 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 contiguous
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 organically. 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

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

- Link list created in heap

C → Structure

C++ → Structure, Class

struct Node *p

p = new Node;

struct Node *p = NULL;

if (p == NULL)

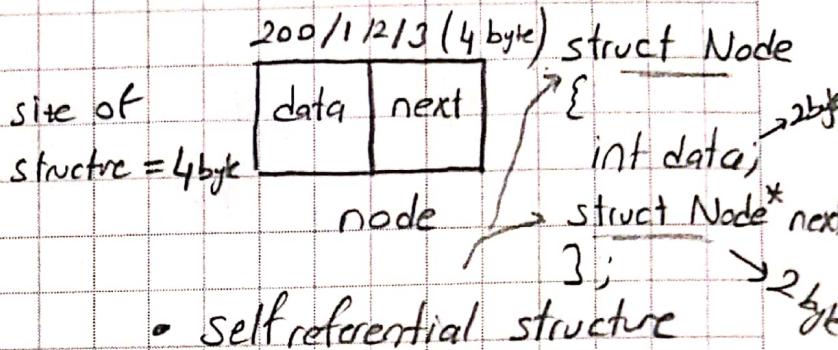
if (p == 0)

if (!p)

if (p != NULL)

if (p != 0)

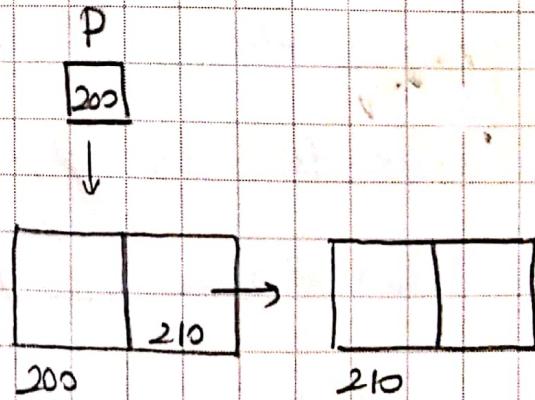
if (p)



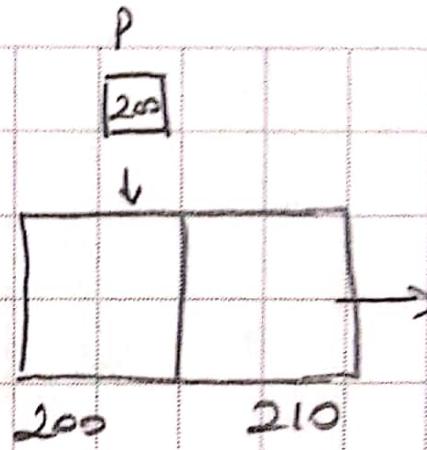
difference between class and structure

Class → everything by default is private

Structure → everything by default is public



if ($p \rightarrow \text{next} == \text{NULL}$)



P is a last node

if ($p \rightarrow \text{next} != \text{NULL}$)

p is continuous we have another nodes

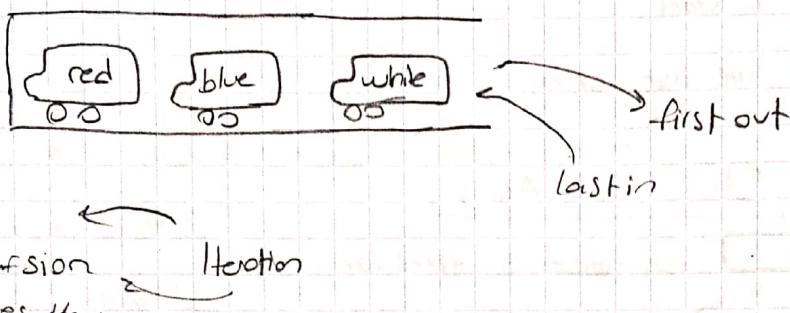
STACK

Stack ADT

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

En son eklenen ilerle silinir

LIFO : Last in first out



Recursion are the functions

which call themselves like loop

Recursive use stack

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

→ If you have a known procedure recursion and you want to convert iteration maybe 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 programmer should create stack and implement the stack in the program

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

Stack Adt

ADT
T

→ Abstract 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

→ Top pointer: pointing on the topmost element in the stack for storing both collection of elements because the important one of is topmost element which one was recently inserted in

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

10	15	17		
0	1	2	3	4

↑
top

size : 5 fixed size

O(1) constant

3 things we need

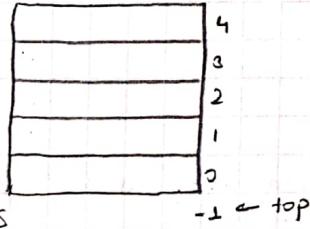
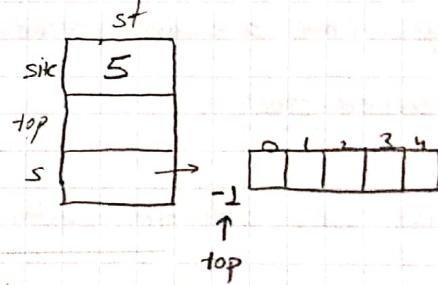
1- array

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

Full

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

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

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

Push

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

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

time taken constant

POP

→ Time taken by push and pop operation is constant.

	4	position	index = top - position + 1
20	3	1	$3 = 3 - 1 + 1$
8	2	2	$2 = 3 - 2 + 1$
15	1	2	$1 = 3 - 3 + 1$
10	0	3	$0 = 3 - 4 + 1$

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;

}

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)

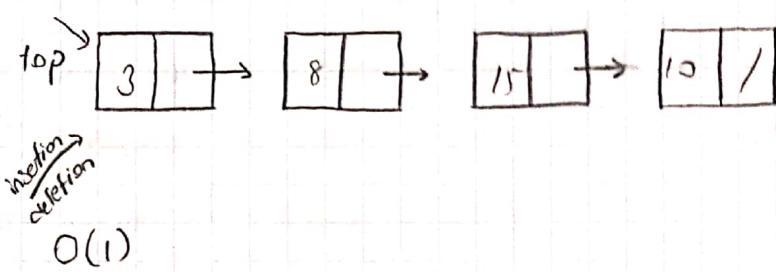
else

return 0; (false)

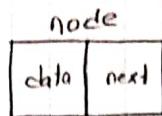
}

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 deletions is done by using LIFO.



insertion
deletion
 $O(n)$



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)
```

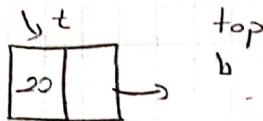
```
    print ("stack overflow");
```

```
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;
```

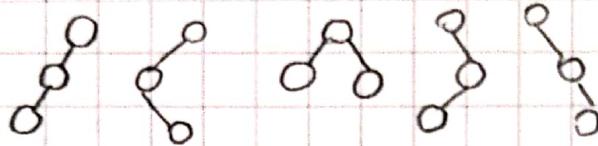
int isFull()

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

Trees

Number of binary trees

$$n=3$$

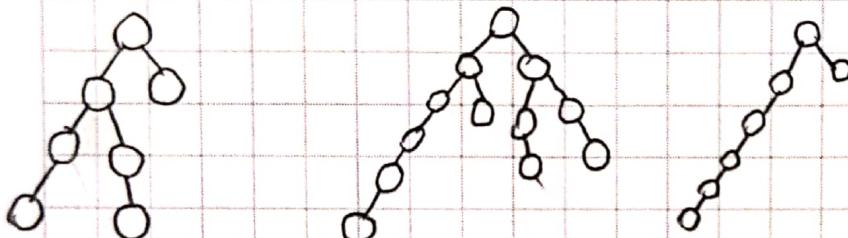


combination formula

$$T(3)=5$$

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

Internal Nodes & External (leaf) nodes



$$\begin{aligned} \deg(2) &= 2 \\ \deg(1) &= 2 \\ \deg(0) &= 3 \end{aligned}$$

$$\begin{aligned} \deg(2) &= 3 \\ \deg(1) &= 5 \\ \deg(0) &= 4 \end{aligned}$$

$$\begin{aligned} \deg(2) &= 1 \\ \deg(1) &= 4 \\ \deg(0) &= 2 \end{aligned}$$

$$\rightarrow \boxed{\deg(0) = \deg(2) + 1}$$

Strict binary tree $\{0, 1, 2\}$

if height is given h

$$\text{Min nodes } n = 2^h + 1$$

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

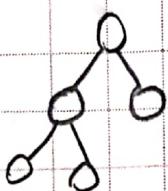
if n nodes are given

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

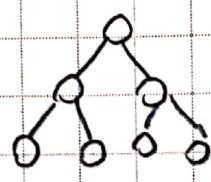
$$\text{Max height } h = \frac{n-1}{2}$$

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

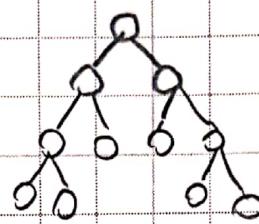
Strict binary tree
Internal x external nodes



$$i=2 \\ e=3$$



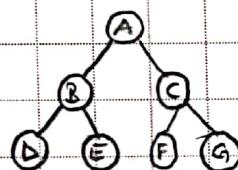
$$i=3 \\ e=4$$



$$i=5 \\ e=6$$

$$e = i + 1$$

Array Representation of Binary Tree



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

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

heap - stack \rightarrow it can be

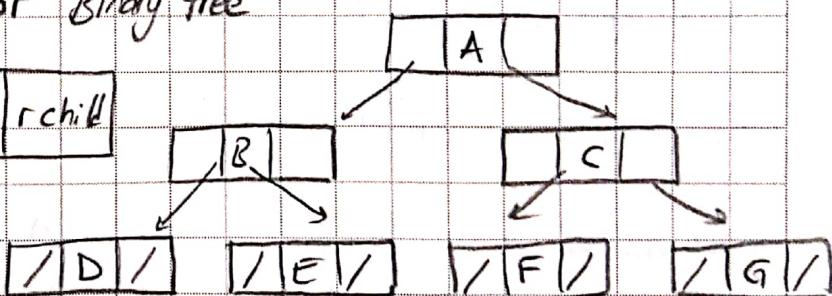
- element i
- leftchild $2i$
- rightchild $2i+1$
- parent $i/2$

Link Representation of Binary tree

• doubly linked list

lchild	data	rchild
/D/	B	/E/
/F/	C	/G/

definitely heap \rightarrow dynamic

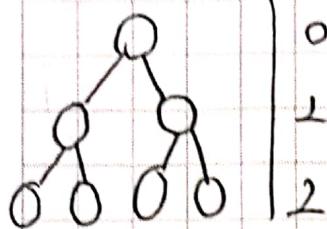


$$n=7$$

$n+1 \rightarrow$ null pointers save

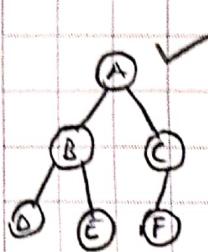
$$e = i + 1$$

Full vs Complete Binary Tree

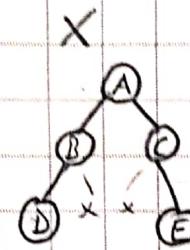


0
1
2

Full = each height have max number of nodes
(herdeki bir yere node eklenmeye calistigimda height yuzerinden)



Complete ✓

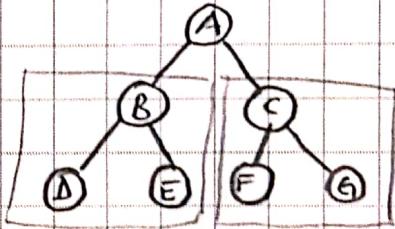


Not complete X

A | B | C | D | E | F

A B C D - - E

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



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

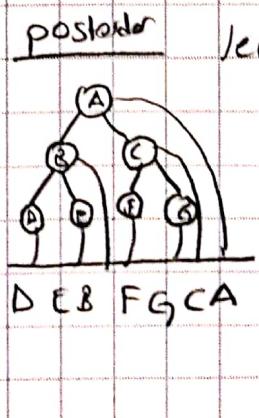
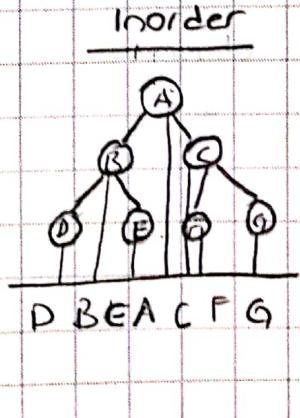
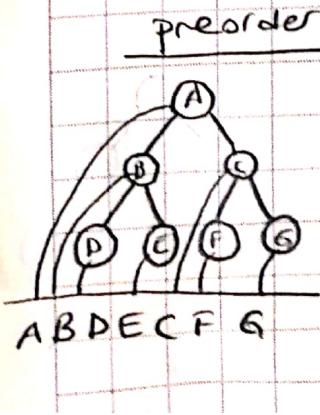
inorder = (D,B,E) A (F,C,G)

D B E A F C G

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

D, E, F, C, G, A

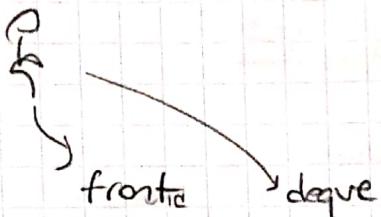
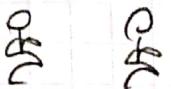
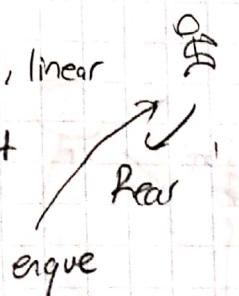
level = A, B, C, D, E, F, G



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

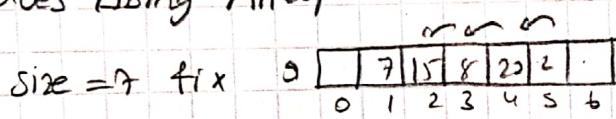
Operations

- 1 - enqueue
- 2 - dequeue
- 3 - isEmpty
- 4 - isFull
- 5 - first()
- 6 - last()

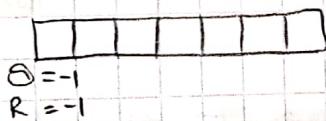
Queue can be implemented as

- 1 - Array
- 2 - Linked List

Queue Using Array



Insert - $O(1)$
Delete - $O(n)$ \rightarrow * one pointer



empty

initially

insert $O(1)$
delete $O(1)$

* * 2 pointers

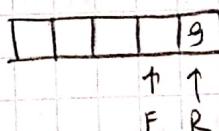
full

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

SORTING TECHNIQUES

Criteria for analysis

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

Bubble Sort, adaptive ✓ stable ✓

A [8 | 5 | 7 | 3 | 2] max element

1st pass	[8]	5	5	5	5	2 nd	[5]	5	5	5
	5	[8]	7	7	7		7	7	3	3
	7	7	[8]	3	3		3	3	7	2
	3	3	3	[8]	2		2	2	2	7
	2	2	2	2	[8]		8	8	8	8

4 compare 4 swap

3 compare 3 swap

3rd pass	[5]	3	3	3	3	4 th	[3]	2	2	2
	3	5	2	2	2		2	3	5	5
	2	2	5	7	7		7	7	7	8
	7	7	7	8	8		8	8	8	8

2 compare 2 swap

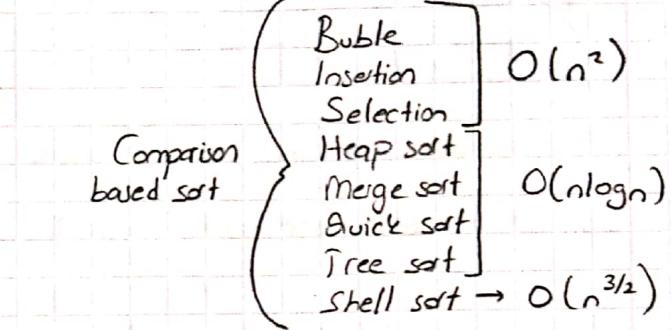
1 compare 1 swap

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

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

```
if(flag == 0)
    return;
```

```
}
```



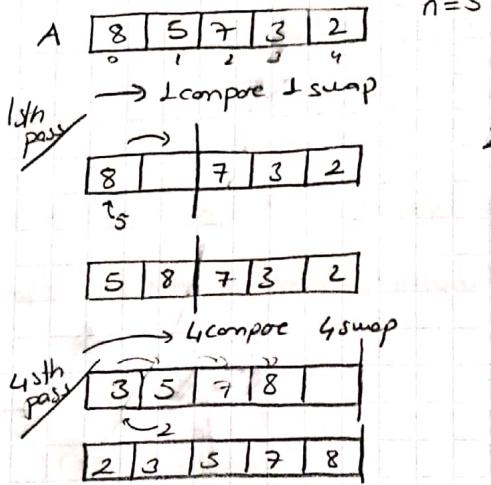
$n-1$ passes
 Comparison = $O(n^2)$
 Swap = $O(n^2)$

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

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

* Bubble ismi suya tıas atıldıgında yubri gitken bulanıklar gibi yavas yavas yer degistiren dezi

Insertion Sorts Adaptive ✓ stable ✓



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

So insertion sort more useful with linked list 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;
    }
}
```

	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

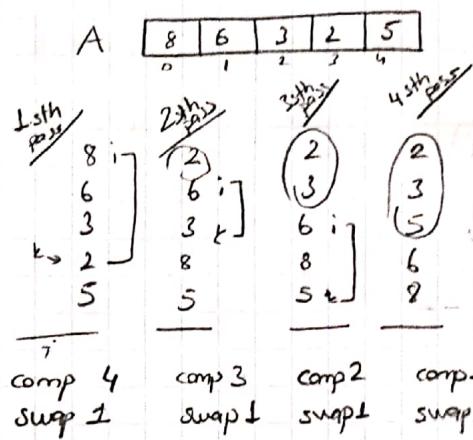
```
int main() {
    int A[] = {10, 12, 8, 9, 5, 11, 8, 3};
    print(A, sizeof(A)/sizeof(A[0]));
    insertionSort(A, sizeof(A)/sizeof(A[0]));
    print(A, sizeof(A)/sizeof(A[0]), "Sorted A");
    return 0;
}
```

	Bubble 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	$O(n^2)$	$O(n^2)$
Adaptive	✓	✓
Stable	✓	✓
Linked list	✗	✓
k passes	✓	✗

Already in ascending

" " descending

Selection Sort, stable X adoptive X



minimum swap

min element

	Comparison	Swap
Best case	$O(n)$	$O(n)$
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]);
}
    
```

	Insertion	Bubble	Selection	(Exchange Sorts)
Best	$O(n)$	$O(n^2)$	$O(n^2)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Average	$O(n)$	$O(n)$	$O(n)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Worst	$O(n)$	$O(n)$	$O(n)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
	$O(n^2)$	$O(n^2)$	$O(n^2)$	
P	0	0	$O(n)$	
	$O(n^2)$	$O(n^2)$	$O(n)$	
	$O(n^2)$	$O(n^2)$	$O(n)$	

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

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

→ Exploits the best case performance of insertion sort

→ Strategy = make list mostly sorted, final insertion sort can complete sorting

→ limited shifting few element shifting

// code similar to insertion 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;
        }
    }
}
    
```

Merge Sort

state ✓ void Merge(int x[], int y[], int z[], int m, int n)

i → 0	2	j → 0	4
1	10	1	9
2	18	2	19
3	20	3	25
4	23		

```
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++];
```

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

```
for (; i < m; i++)
    C[k++] = A[i];
for (; j < n; j++)
    C[k++] = B[j];
```

→ Requires extra spaces

→ $n \log n$

→ divide and conquer

→ requires extra space

Quick Sort

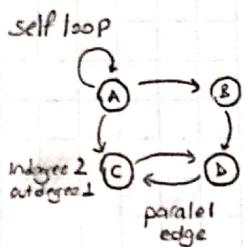
→ pivot

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

GRAPHS

$G(V, E)$

Directed graph: if edges are having direction



Simple graph: without self loop and parallel edge

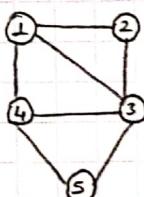
Strongly connected: from every vertices we can reach all other vertices. (there is a path between every vertex)

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

Adjacency Matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 1 & 2 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 & 1 & 1 \\ 4 & 1 & 0 & 1 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

$n \times n$
 $\mathcal{O}(n^2)$



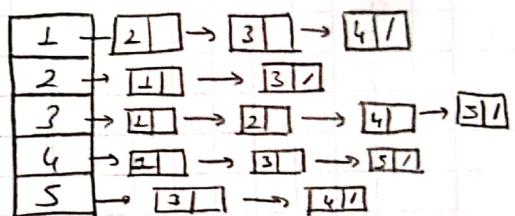
$G = (V, E)$

$|V| = 5$

$|E| = 7$

time complexity: depend on vertices $\mathcal{O}(n^2)$

Adjacency List



$|V| + 2|E|$

time complexity: depend on vertices and edges

$\mathcal{O}(n)(|V| + 2|E|)$