2022

STRUTTURE DATI

ITI G.B. BOSCO LUCARELLI
DANIIL TOPCII 4IA

COSA SONO?

Le strutture dati sono un metodo per contenere dati o raggruppamenti di dati che permettono una veloce interazione con essi, ogni struttura dati ha poi un proprio metodo per interagire con essi

LE STRUTTURE DATI PRINCIPALI

- MATRIX
- ARRAY
- LINKED LIST
- QUEUE & PRORITY QUEUE
- STACK
- HASH TABLE
- GRAPHS & TREES

SI DIVIDONO PRINCIPALMENTE IN

- LINEARI
  - I dati sono disposti in una sequenza lineare; in cui gli elementi sono collegati uno dopo l'altro, fra questi abbiamo: array, matrix, stack, linked list, hash table
- NON LINEARI
  - I dati sono collegati gerarchicamente e sono presenti a vari livelli come: grafi ed alberi

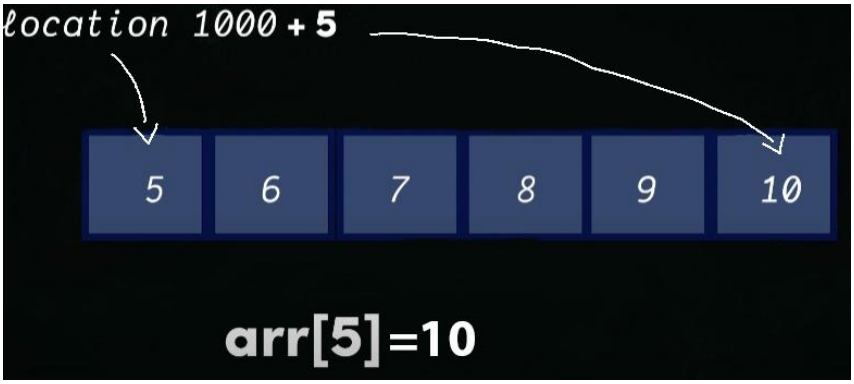
ARRAY

L'Array è una struttura dati base che immagazzina dati simili, usa la scrittura continua in memoria quindi è facile recuperare un dato al suo interno sapendo dove inizia l'array, alla posizione iniziale basta aggiungere il suo indice nell'array es:

posizione di memoria dell'inizio dell'array =100

per trovare il dato posizionato in indice 6:

1000+5=106 che sarebbe la posizione in memoria



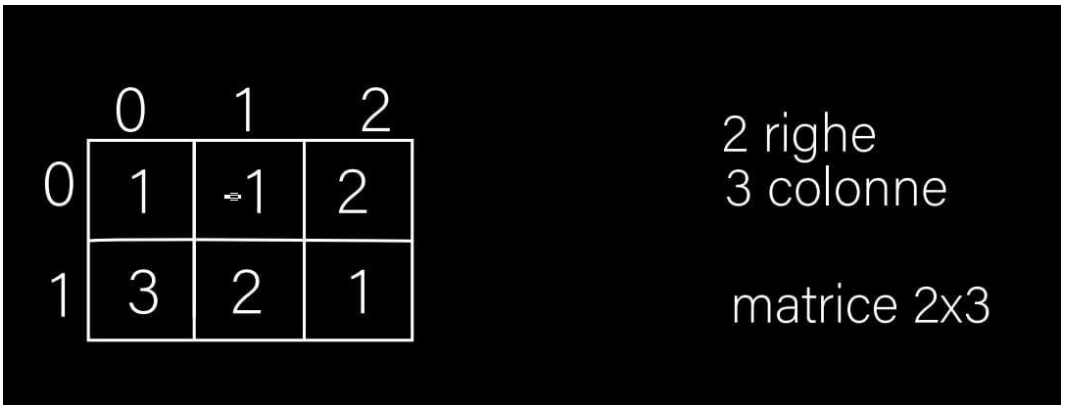
- PRO: come già detto in è facile recuperare un dato dall'array
- CONTRO: nei linguaggi a basso livello va dichiarata la lunghezza dell'array inizialmente altrimenti si va a sovrascrivere altri dati in caso si superi la dimensione indicata inizialmente, questo succede perché una volta dedicato lo spazio di memoria per un array questo resta statico, per "aggiungere" altri elementi bisognerebbe distruggere tutto l'array in memoria e crearlo con la cella di memoria in più, ciò non avviene però nei linguaggi più avanzanti come python dove il cambio della dimensione dell'array avviene in automatico ,

l 'array viene spesso utilizzato come base per altri modelli come le matrici e le linked list

```cpp
//codice implementazione array
#include <iostream>
using namespace std;

int main()
{
    int arr[5];
    return 0;
}
```

MATRIX

Le matrici sono una struttura dati statica ed omogenea con due dimensioni. Come i vettori condividono la caratteristica di staticità (la grandezza della struttura dati non può essere modificata) e l'omogeneità (il tipo dei dati che contiene è lo stesso per ogni cella) ma hanno due dimensioni righe e colonne.




```cpp
//codice implementazione matrice
#include <iostream>
using namespace std;

int main()
{
    //array con 3 righe 2 colonne .
    int x[3][2] = {{0,1}, {2,3}, {4,5}};

    // come fare l output di tutti gli elementi
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cout << "Element at x[" << i
                << "][" << j << "]: ";
            cout << x[i][j]<<endl;
        }
    }
    system("PAUSE");
    return 0;
}
```

LINKED LIST

La Linked List è una lista organizzata di dati o nodi dove ogni nodo contiene un dato e il puntatore che punta al prossimo elemento della lista

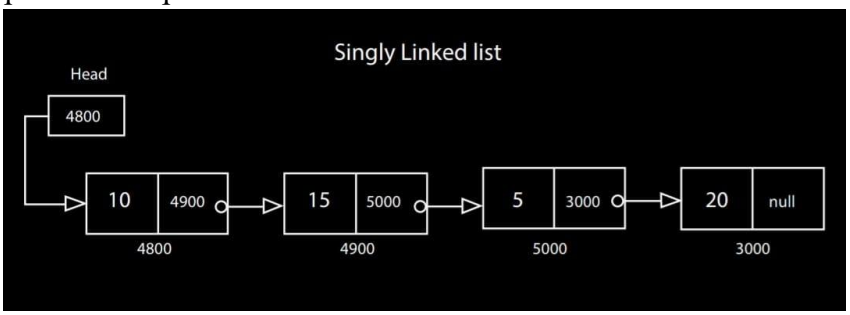
il primo elemento della lista viene definito HEAD, mentre l'ultimo TAIL che non contiene puntatori

PRO: molto semplice aggiungere nuovi nodi in quanto si può semplicemente cambiare il nodo a cui punta il puntatore

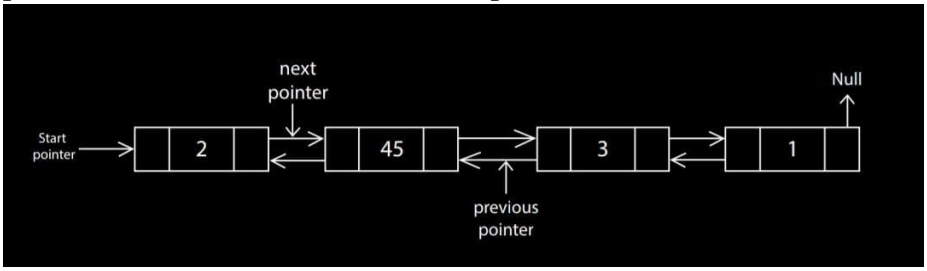
CONTRO: utilizza tanta memoria in quanto oltre ai dati nei nodi sono presenti 1 o 2 puntatori ad altri nodi, ricerca o recupero dati lento anche se si conosce l'indice di posizione in quanto ogni nodo sa solo dell'esistenza del nodo vicino

TIPI:

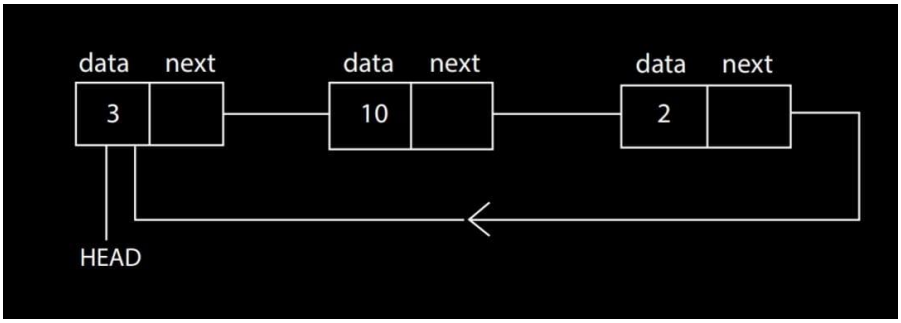
- SINGLE LINKED LIST
  - la navigazione è solo in avanti cioè unilaterale a causa della presenza di un puntatore per nodo



- DOUBLY LINKED LIST
  - è possibile una navigazione sia in avanti che indietro a causa della presenza di 2 puntatori nel nodo, uno al nodo precedente e uno al successivo



- CIRCULAR LINKED LIST
  - l'ultimo elemento ha un puntatore al primo che rende la lista "circolare"



IMPLEMENTAZIONE IN CPP DELLA LINKED LIST:

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int data; //dato inserito
    Node* next; //puntatore al prossimo n
};

//stampa i dati
//a partire dal nodo dato
void printList(Node* n)
{
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}

// Driver code
int main()
{
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

    // alloco 3 nodi
    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1; //dato del primo nodo
    head->next = second; //1 nodo -> 2 nodo

    second->data = 2; // dato secondo nodo
    second->next = third;//2 nodo -> 3 nodo

    third->data = 3; //dato  terzo nodo
    third->next = NULL;//3 nodo-> NULL

    printList(head);
    system("PAUSE");
    return 0;
}
```

QUEUE & STACK

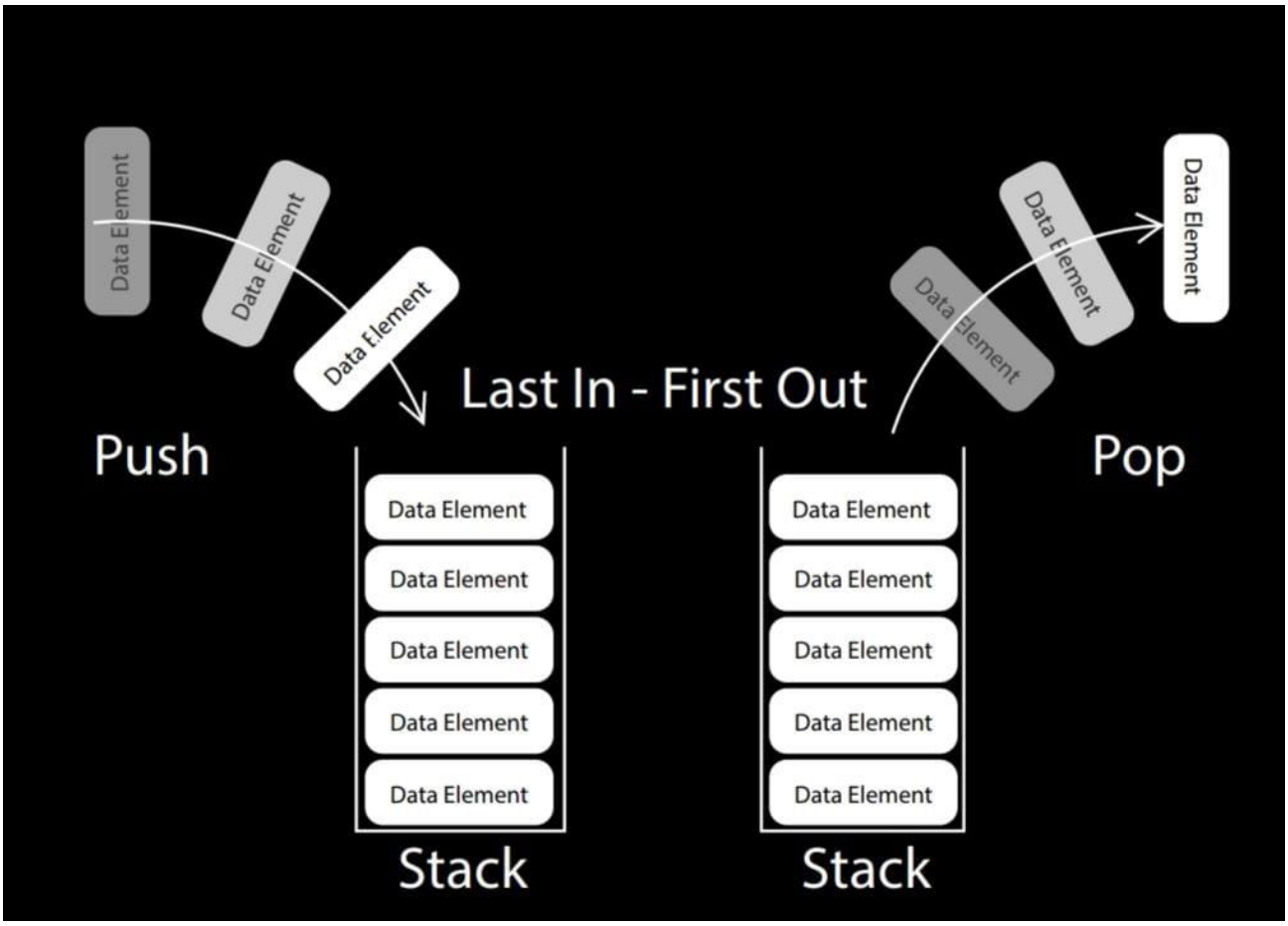
lo stack e la queue sono strutture dati abbastanza simili con caratteristiche diverse , entrambi costruite sul concetto di array.

STACK:

Lo stack è una Last In Frist Out struttura dati simile a una pila di piatti, l'ultimo piatto ad essere lavato è il primo ad essere asciugato.

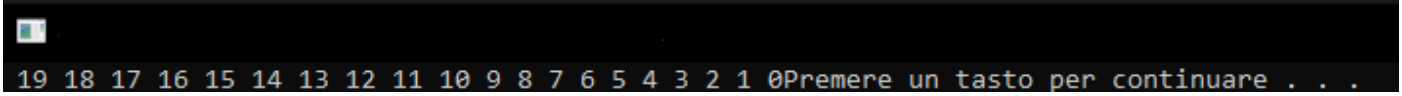
Inserire un dato viene definito "pushing" invece prelevare un dato invece viene detto "pop"

Si ottiene il famoso Stack Overflow quando vengono inseriti troppi dati nella struttura dello stack e quindi si ha una fuoriuscita dai limiti della memoria



IMPLEMENTAZIONE IN CPP DELLO STACK:

```cpp
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> stack;                   //dichiaro lo stack
    for (size_t i = 0; i < 20; i++)     //con un ciclo for
    {
        stack.push(i);                  //lo riempio di 20 elementi
    }

    while (!stack.empty()) {            //finchè non è vuoti
        cout << ' ' << stack.top();     //stampo il valore
        stack.pop();                    //e lo prelevo dallo stack
    }
    system("PAUSE");
    return 0;
}
```



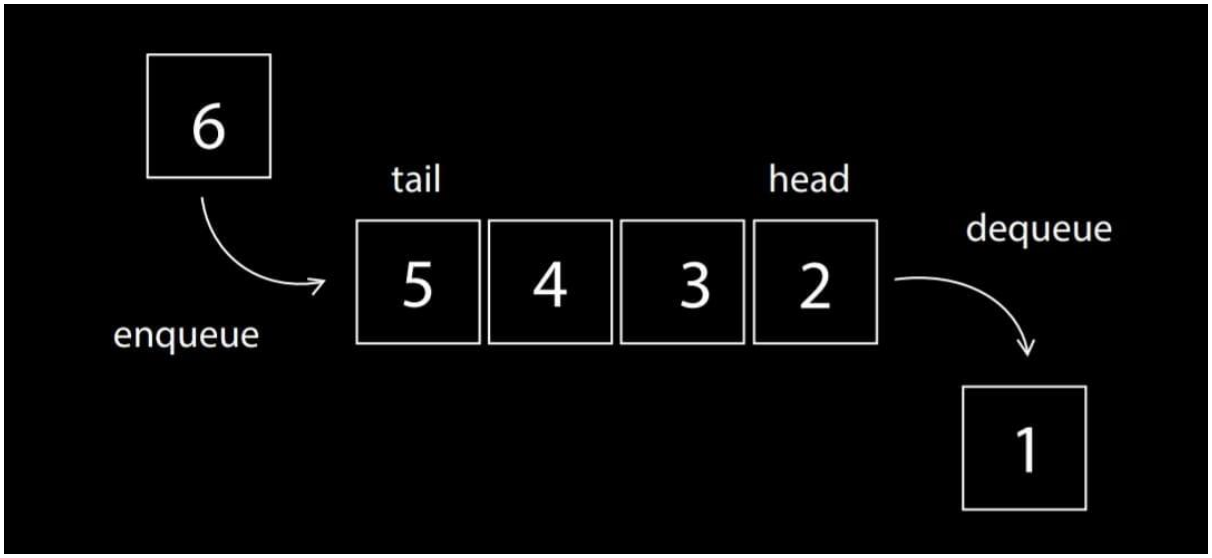
```
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0Premere un tasto per continuare . . .
```

QUEUE:

è in tutto e per tutto una FILA in quanto è una struttura dati Frist In Frist Out, come la fila per la cassa il primo ad entrare è il primo ad essere servito, a meno che non ci sono precedenza che può cambiare l'ordine di uscita.

Aggiungere un elemento in questo caso alla coda viene definito "enqueue" mentre rimuoverlo viene definito "dequeue".

utilizzato maggiormente per organizzare dati che non verranno processati immediatamente



IMPLEMENTAZIONE IN CPP DELLA QUEUE:

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main()
{
    //FILA VUOTA
    int c = 0;

    queue<int> myqueue;             //dichiaro la queue
    for (size_t i = 0; i < 20; i++) //con un ciclo for
    {
        myqueue.push(i);            //aggiuno 20 elementi alla coda
    }


    while (!myqueue.empty()) {      //finchè la coda non è vuota
        cout << ' ' << c;                   //stampo l elemento
        myqueue.pop();              //lo levo dalla coda
        c++;                        //aumento la posizione di 1
    }
    system("PAUSE");
    return 0;
}
```

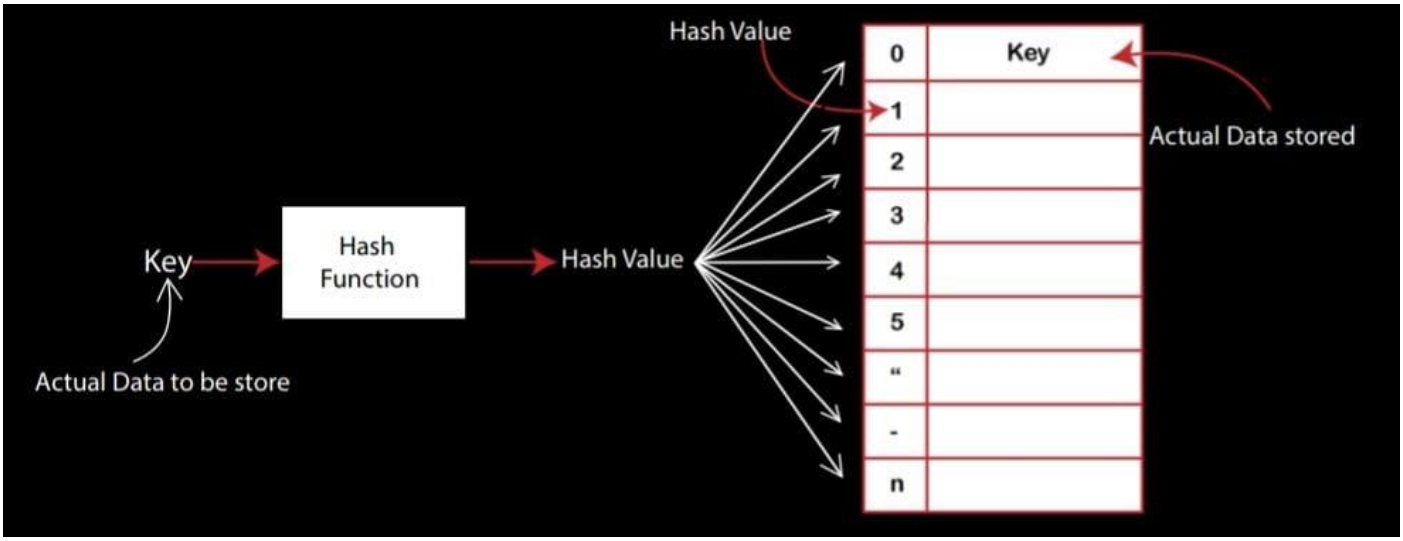
```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19Premere un tasto per continuare . . .
```

HASH TABLE

L'Hash Table o semplicemente Table è una struttura dati con un funzionamento logico simile all array , l'index viene dato in pasto a una funzione chiamata "hashing funciotn" che ne tira fuori l indirizzo di memoria, la cosa che cambia però dall'array è che le celle in memoria non sono poste una di seguito all'altra in quanto non necessario dato che con ogni index si ha la posizione esatta in memoria, può essere considerata come una sorta di libreria dove in base al nome de libro si ricorda la sua posizione esatta nella libreria

PRO: aumentandone le dimensioni quindi a differenza dell'array non si ricorre in una violazione di memoria dato che i vari dati vengono scritti in maniera casual, conoscre la posizione precisa in memoria permette anche una ricerca e un recupero dei dati praticamente immediato

CONTRO: in varie circostanze la funzione "hashing function", in base all'algoritmo utilizzato, può dare con index diversi lo stesso indirizzo di memoria, ciò viene definito "collition"



IMPLEMENTAZIONE IN CPP DELL HASH TABLE:

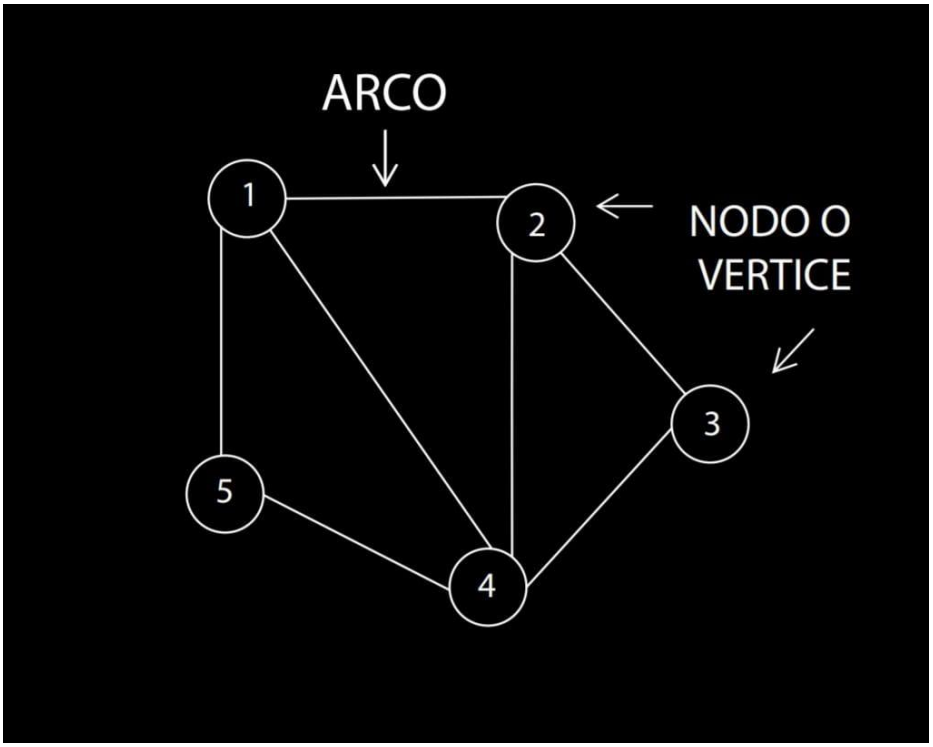
```cpp
#include<bits/stdc++.h>
using namespace std;

class Hash
{
    int BUCKET;
    //Puntatore a un array contenente "BUCKETS"
    list<int> *table;
public:
    Hash(int V); //Costruttore

    // inserire una kay nell hash table
    void insertItem(int x);

    // eliminare una kay dall hash table
    void deleteItem(int key);

    // funzione hash per mappare i valori sulla chiave
    int hashFunction(int x) {
        return (x % BUCKET);
    }

    void displayHash();
};

Hash::Hash(int b)
{
    this->BUCKET = b;
    table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
{
    int index = hashFunction(key);
    table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
// ottenere l'indice hash della chiave
int index = hashFunction(key);

// trova la chiave nell'(indice)esimo elenco
list <int> :: iterator i;
for (i = table[index].begin();
        i != table[index].end(); i++) {
    if (*i == key)
    break;
}
```

```
if (i != table[index].end())
    table[index].erase(i);
}

// se la chiave viene trovata nella tabella hash, rimuoverla
void Hash::displayHash() {
for (int i = 0; i < BUCKET; i++) {
    cout << i;
    for (auto x : table[i])
    cout << " --> " << x;
    cout << endl;
}
}


int main()
{
// array che contiene le chiavi da mappare
int a[] = {1, 2, 3, 4, 5};
int n = sizeof(a)/sizeof(a[0]);

// inserire le chiavi nella tabella hash
Hash h(5); // 5 è il numero di BUCKET
for (int i = 0; i < n; i++)
    h.insertItem(a[i]);

// elimino 1 dall hash table
h.deleteItem(1);

// stampa
h.displayHash();
system("PAUSE");
return 0;
}
```

GRAPHS

DEFINIZIONE MATEMATICA:

Un grafo è definito come una coppia di insiemi <V,E> ; dove "V" è l' insieme dei vertici ed "E" l'insieme degli archi, "E" presenta relazioni fra due vertici e finchè G sia un grafo è necessario che l'insieme "E" è incluso nel prodotto fra V*V ciò significa che l'insieme "E" non è altro che un insieme di vertici appartenenti all'insieme "V", cioè coppie di vertici appartenenti al grafo.



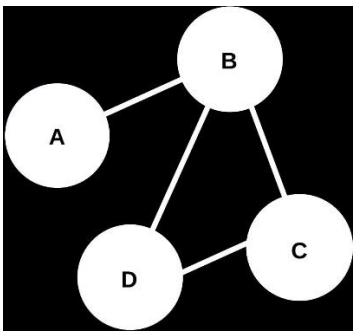
Le strutture dati "grafo" sono quindi una collezzione di nodi uniti attraverso degli archi, un arco che unisce due vertici prende il nome di spigolo.

Nei grafi non ci sono regole che definiscono le connessioni fra i vertici, ci può essere un numero qulsiasi di nodi e un numero

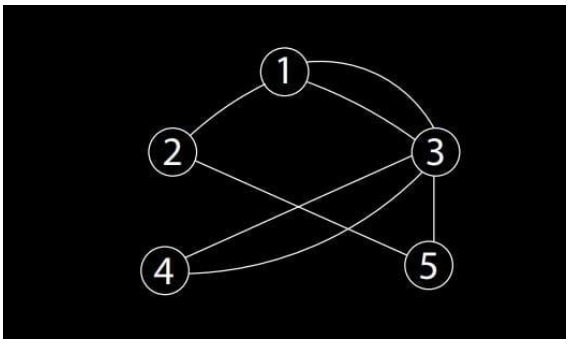
qualsisi di connessioni fra questi, ogni vertice nel grafo deve presentare una qualsisi identificazione.

SI DIVIDONO IN BASE ALLA MOLTEPLICITA' DEGLI ARCHI:

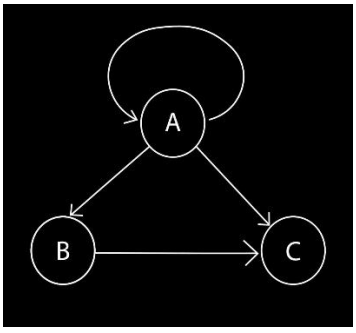
- SEMPLICI:
  - tra ogni coppia di vertici può o non può esserci un arco, non ci sono più archi che collegano gli stessi vertici



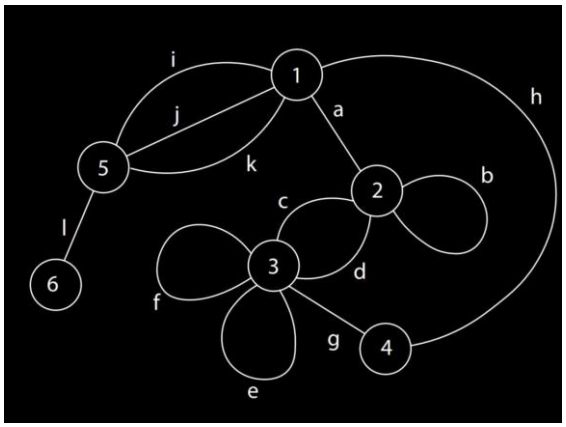
- MULTIGRAFO:
  - tra due e più vertici possono non esserci archi, può esserci un solo arco e possono esserci più archi



- LOOPS o PSEUDOGRAFO:
  - presenta un arco che si richiude su stesso



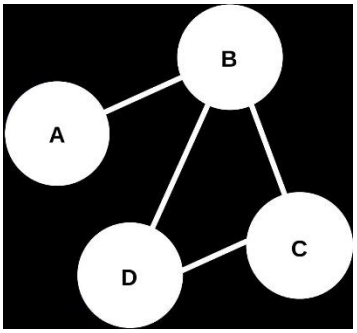
- PSEUDOMULTIGRAFO:



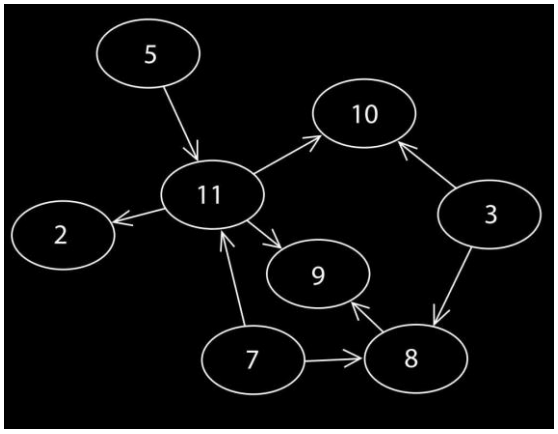
  - se presenta più loops sullo stesso vertice

E IN BASE ALLA DIREZIONE:

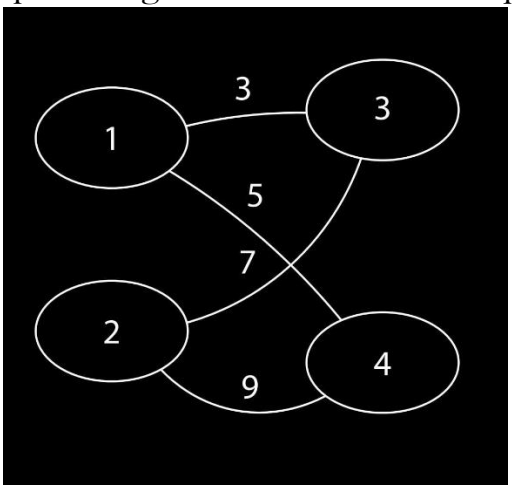
- NON DIRETTI:
  - in una connessione bilaterale non c'è identificazione fra nodo di origine e nodo di destinazione. uguale al grafo orientato con la differenza che l'insieme E è formato da coppie non orientate se per esempio c'è l'unione da un arco 1,5 non essendo orientato e di conseguenza bilaterale, viene sottointeso che esiste anche la coppia 5,1



- ORIENTATI:
  - quando si ha una connessione unidirezionale un nodo viene definito come ORIGINE mentre l'altro sarà la DESTINAZIONE se si vuole avere una comunicazione bilaterale fra i due nodi ci dovranno essere due connessione, avendo così entrambi i nodi che sono sia ORIGINE che DESTINAZIONE. matematicamente il grafo G ha un insieme di V finiti e un insieme di E finiti dove E è rappresentato da una coppia di numeri (cioè i due vertici che mette in relazione).

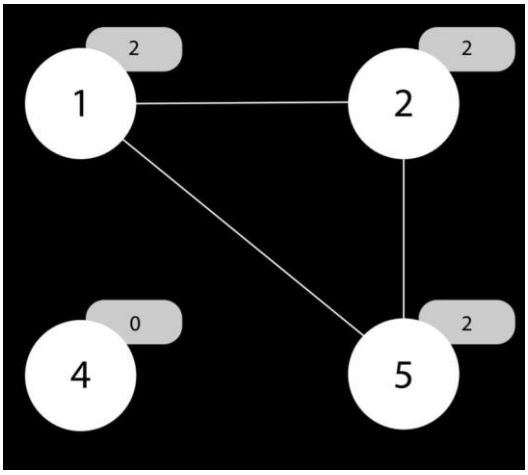


- PESATI O NETWORK:
  - quando agli archi viene data una proprietà

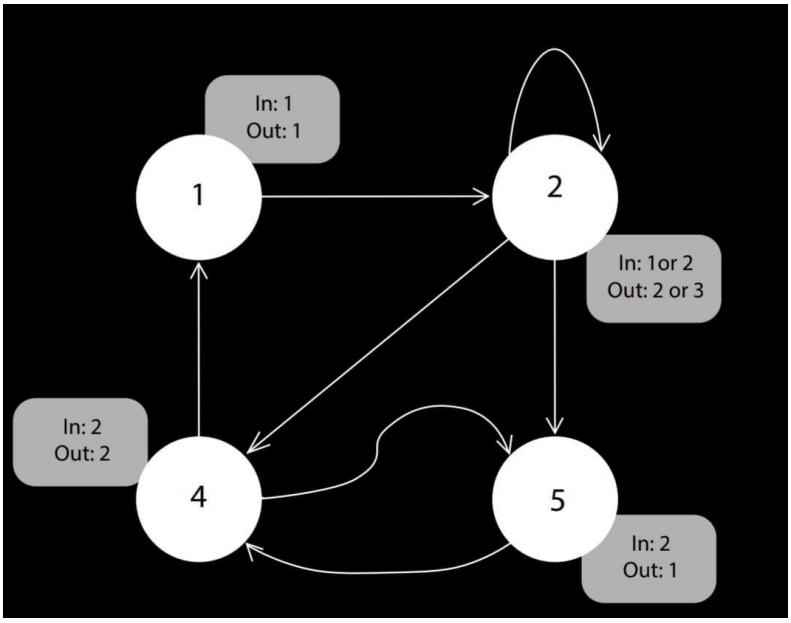


GRADI:

in un grafo non orientato è semplice in quanto basta vedere il numero di archi presenti nel vertice e si ottiene il grado



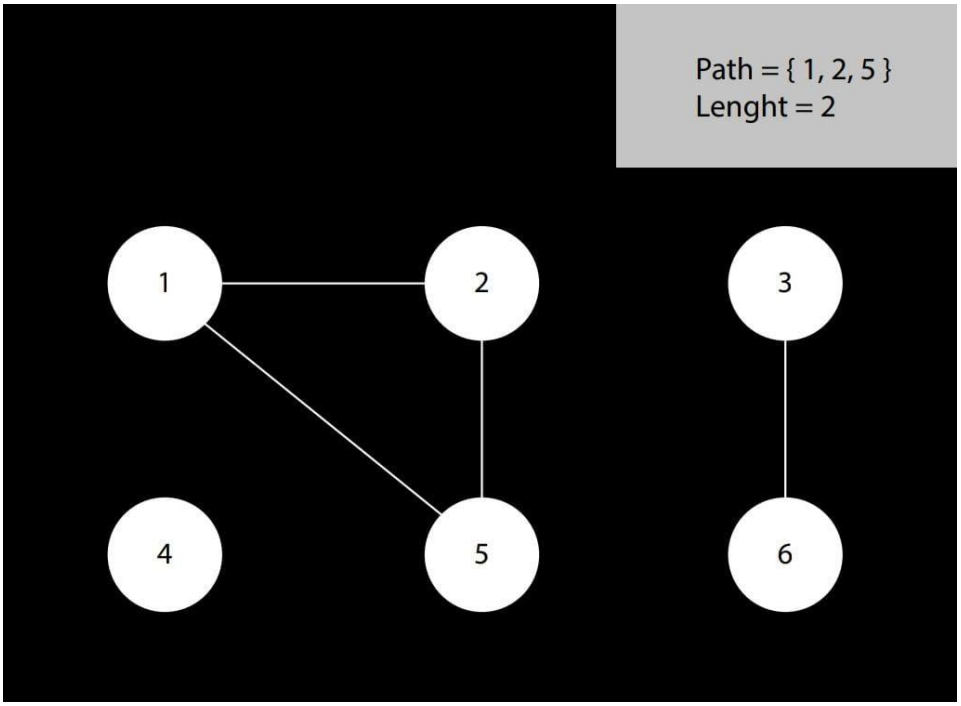
con un grafo orientato la cosa si complica perché bisogna aggiungere la distinzione fra archi uscenti ed archi entranti il grado sarà quindi la somma dei due un arco senza archi viene definito isolato quindi pari a 0 (gradi)



COLLEGGAMENTI: i collegamenti fra i vertici possono essere distinti in semplici collegamenti fra un vertice e l'altro attraverso un singolo arco e "path" da un vertice all' altro, passando attraverso più archi, si può quindi avere un collegamento da un vertice all altro non diretto ma attraverso una path.

Un cammino può essere definito come una sequenza di vertici, se esiste un cammino fra il V1 e il V2 si dice che V2 è raggiungibile da V1

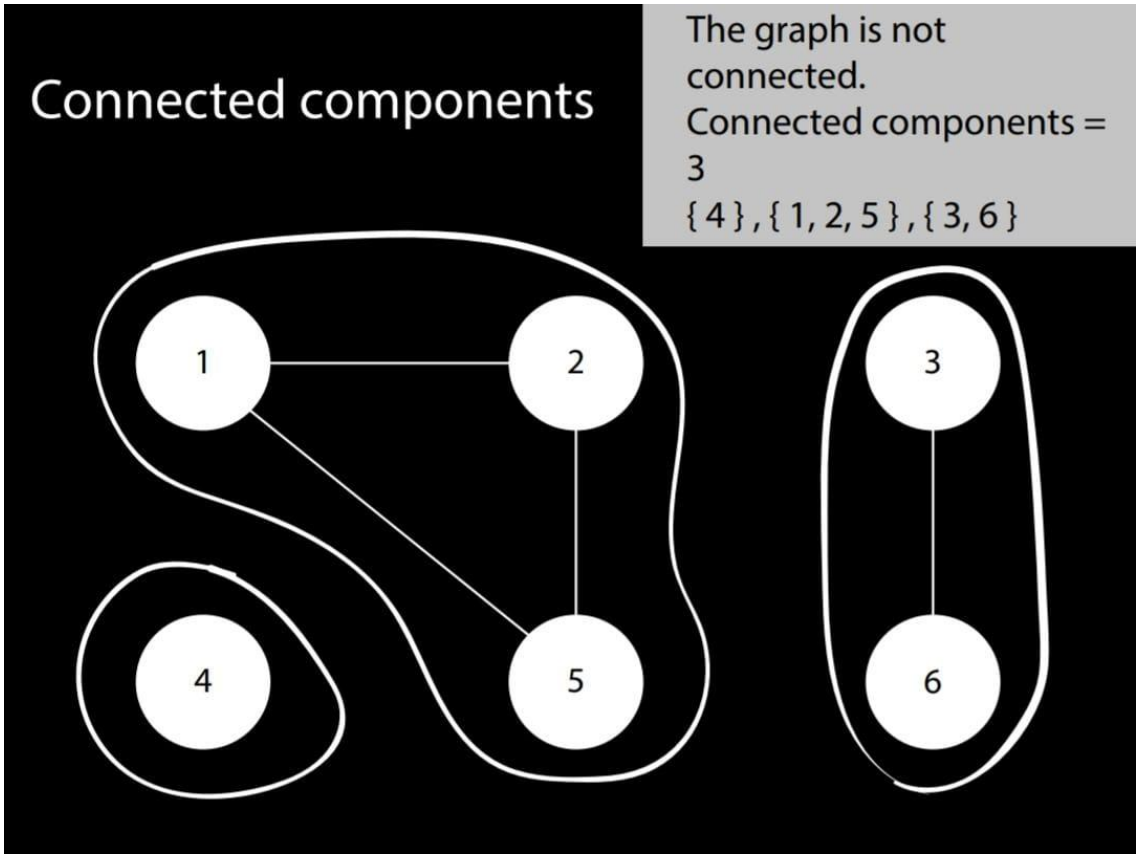
CICLI: un caso particolare di cammino è quando il vertice di cammino è uguale a quello di partenza, questo caso viene definito ciclo,i grafi senza ciclo vengono defini aciclici



CONNETTIVITA' DEI GRAFI:

un grafo viene definito connesso quando per ogni coppia di vertici c'è un cammino che li collega, tutti vertici quindi devono raggiungere l'uno l'atro attraverso un cammino, quando un grafo non è connesso significa che un vertice può raggiungere solo un determinato gruppo di vertici cioè una componente connessa del grafo

Grafi diretti: in questi non è scontato che si possa avere una path 1,2,3 e poi 3,2,1 essendo unidirezionali .Si definisce una "connessione forte" nei grafici diretti quando per ogni vertice esiste almeno un cammino che collega a tutti agli altri vertici rispettando l'orientamento.



GRAFO COMPLETO :

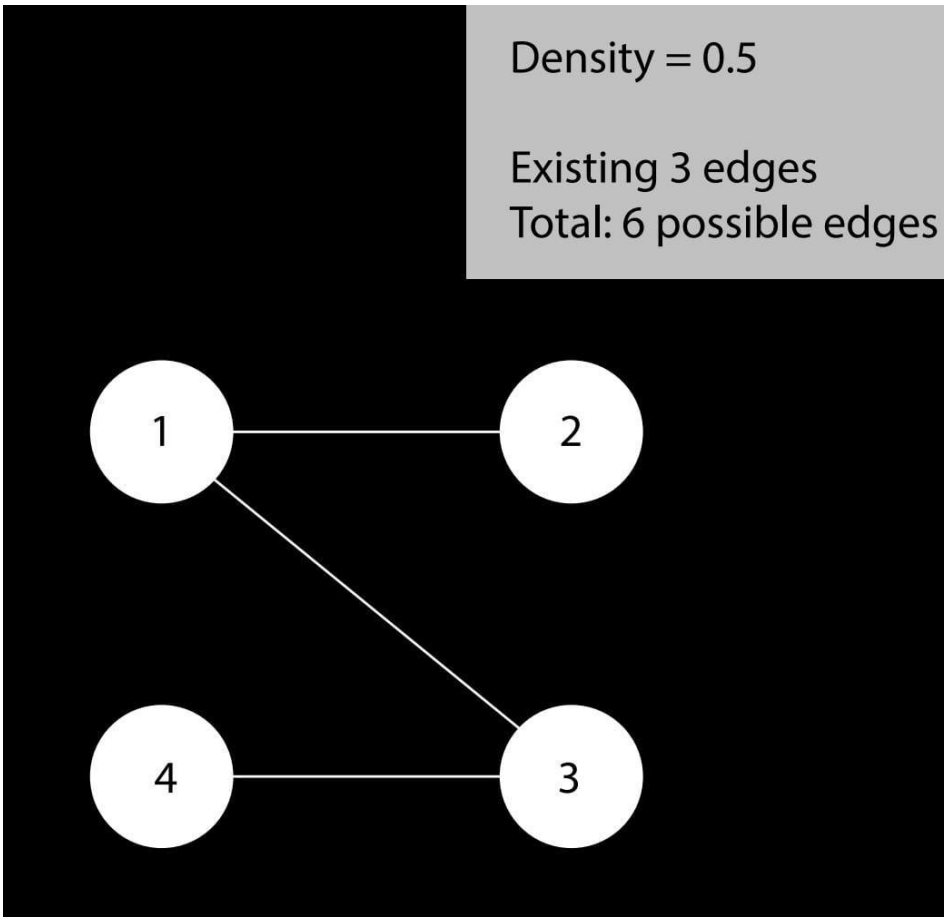
viene definito grafo completo quando per ogni coppia di vertici esiste un arco che li collega, da non confondere con il grafo connesso dove esiste il collegamento con tutti i vertici ma tramite "path" e non tramite archi direttamente connessi,

il numero di archi in un grafo completo è facilmente calcolabile, cioè l'insieme di tutte le coppe di vertici possibili

- diretti: n(n-1) n=numero di vertici
- indiretti: n(n-1)/2
- con i loop=
  - diretti n^2
  - indiretti n(n-1)

grazie a ciò si può definire la densità di un grafo, cioè il rapporto di numero di archi presenti nel grafo e il numero di archi che ci sarebbero se il grafo fosse completo (di pari dimensioni)

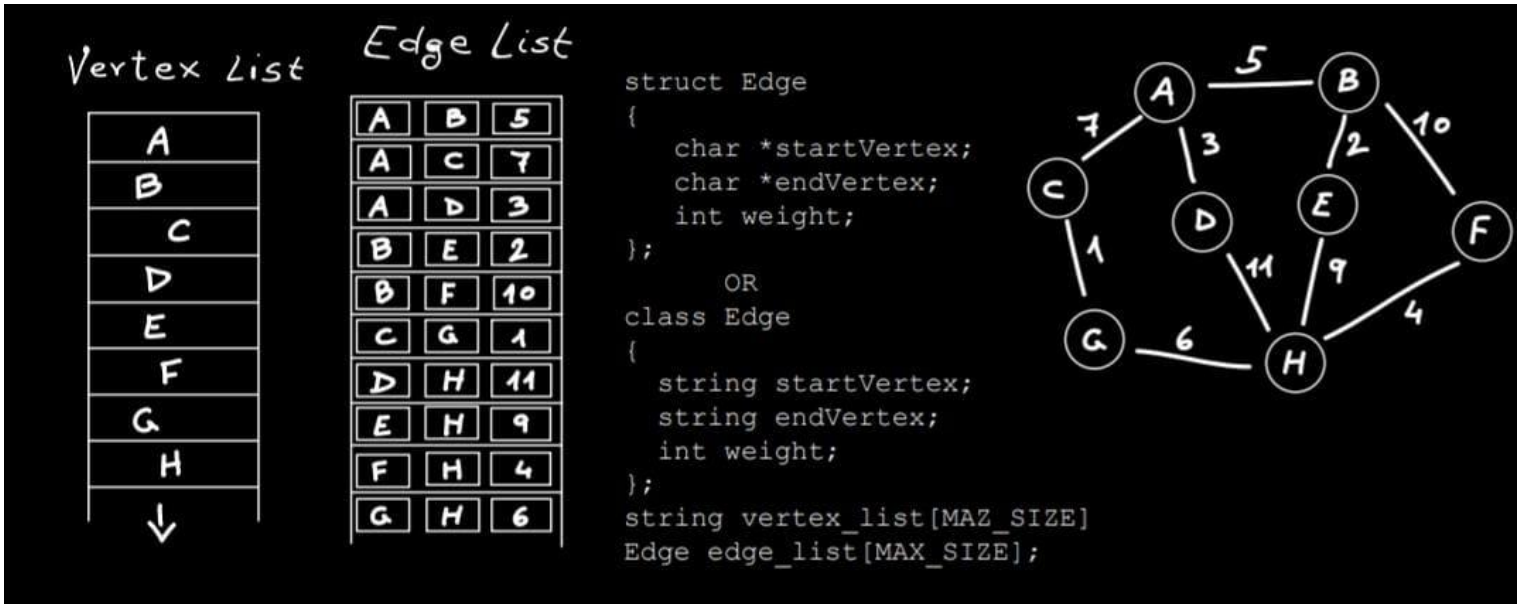
es: si hanno 4 vertici (non orientati) e 3 archi, se si rende completo si possono avere 6 archi 3/6 =0.5, la densità va da 0 a 1



STRUTTURE DATI CON I GRAFI:

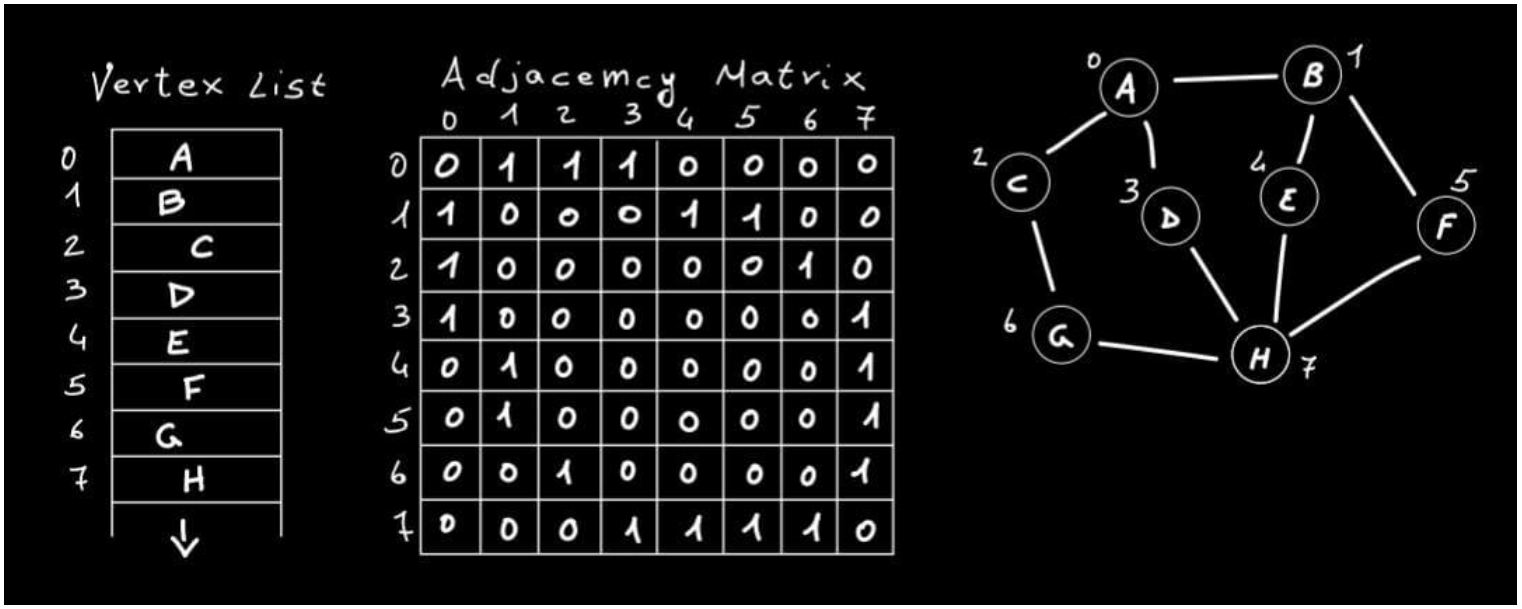
EDGE LIST:

- per memorizzare un grafo nel computer e creare una struttura dati, la cosa più semplice che viene in mente è creare 2 liste (array)
  una che contiene tutti i vertici (vertx list): semplici stringe in quanto i vortici vengono identificati solo dal nome e una che contiene tutti gli spigoli (edge list), un oggetto che contiene due campi uno per il puntatore al vertice di partenza e quello di destinazione, in questa poi si possono aggiungere anche le proprietà.
  NON EFFICIENTE IN COSTI DI TEMPO, come per esempio trovare se due nodi sono collegati o meno perché si deve scorrere tutta la edge list



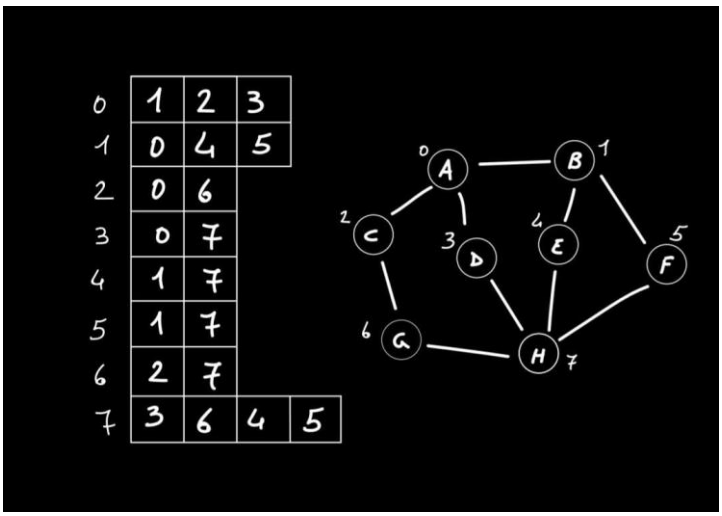
ADJACENCY MATRIX:

- un altro metodo è salvare sempre un array per la (vertex list) e una matrice per l edge list, il numero della riga corrisponde al vertice che si ha in considerazione (nell'array), e come numero della colonna gli altri vertici, se un vertice è collegato all'altro la casella
  che incrocia riga e colonna viene impostata a 1 ,
  es: il vertice 0 ha un arco con il vortice 1,2,3 le caselle 0;1, 0;2, 0;3 verranno impostate a 1
  per trovare i vertici adiacenti il tempo massimo che ci può costare è pari al numero di vertici nella riga + i vortici nella colonna,
  per trovare invece i vertici collegati basta vedere i valori imposti a uno in ogni riga, per velocizzare ciò si può dare ogni indice in pasto all'hash table, si utilizza però tanta memoria, cioè il numero di vertici al quadrato (a causa dei tanti 0 nella matrice)

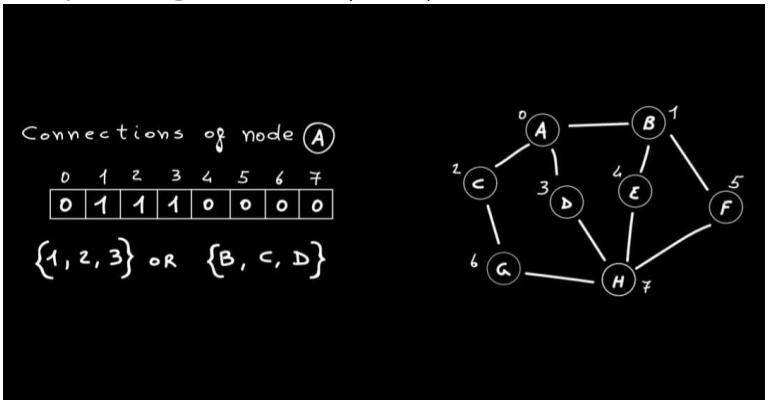


ADJACENCY LIST:

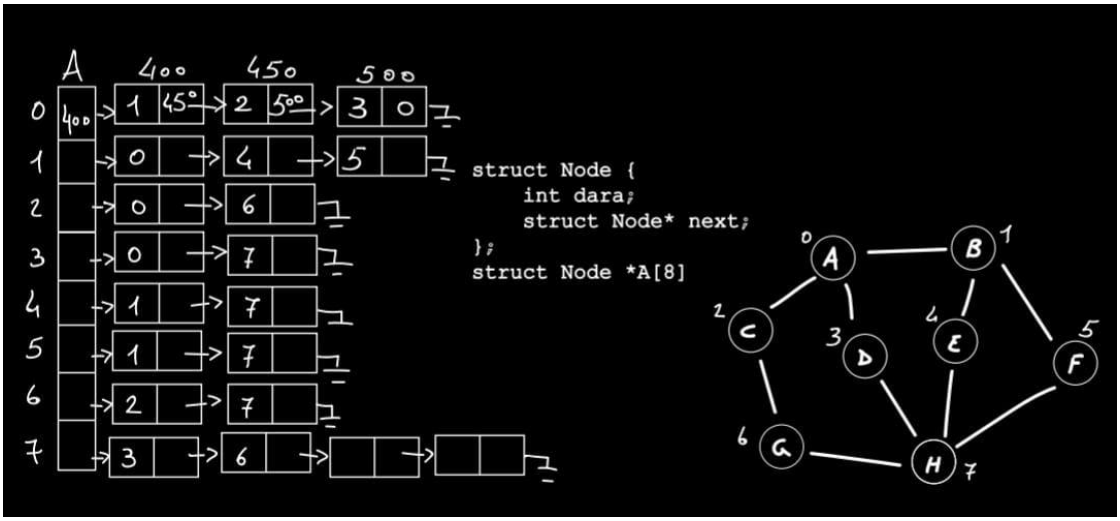
- Matrice



- array per ogni vertice (1,2,3)



- linked list



in ognuna di queste al posto di conservare il flag 0 o 1 si conserva in memoria direttamente l'indice del vertice.
in questo caso lo spazio utilizzato è pari al numero di spigoli

con la matrice e l'array però risulta complicato aggiungere un nuovo elemento quindi la linked list risulta più efficace, ogni riga sarà una linked list.
Prima di questo però si ha un array il cui numero della cella indica il vertice e al suo interno è presente un puntatore all'inizio della linked list (cioè a tutti i vertici a cui è collegato tramite arco)

TREE

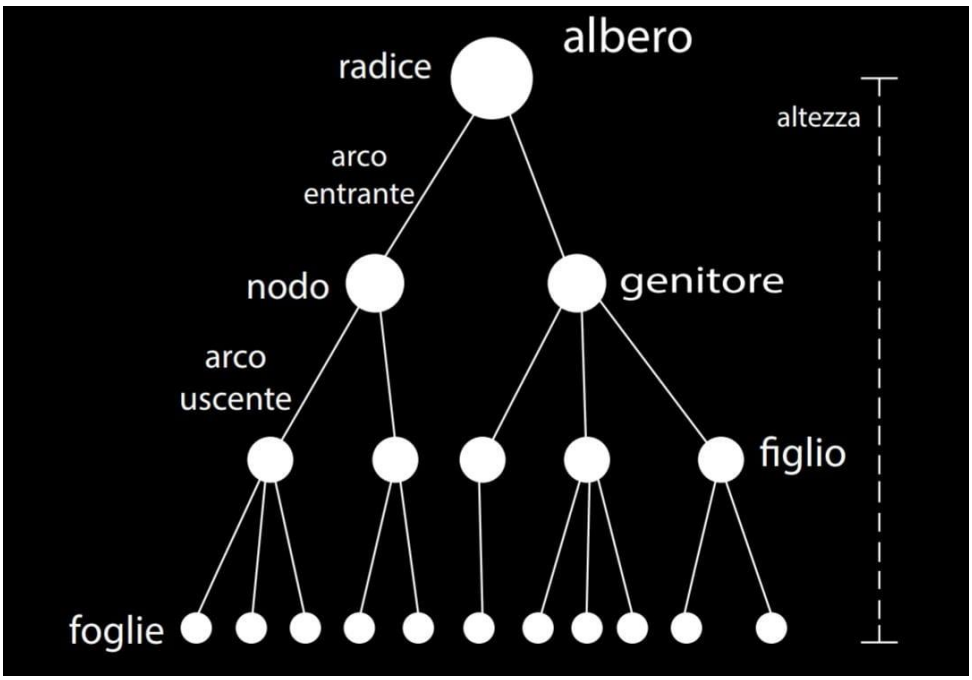
una struttura dati ad albero viene definita come un grafo, indiretto, aciclico e connesso se, un grafo viene definito solo come orientato ed aciclico prende il nome di foresta, si dividono in:

ROOTED TREE:

in un albero viene scelto un nodo che prende il nome di root, cioè il nodo che dà poi l'orientamento ai vari archi, ogni nodo della struttura può diventare la root. Quando si parla di albero con root, il root diventa l'ancestor di tutti gli altri nodi, i nodi padri son quelli che vanno verso la root, mentre i figli son quelli che vanno dalla root in giù, quando da un padre discendono più figli questi vengono chimati fratelli, la radice di conseguenza non ha genitori e tutti i nodi hanno almeno un genitore, gli ultimi figli dell'albero vengono definiti anche come "foglie"

per memorizzare un albero generico il metodo utilizzato dipende dal numero dei figli:

- realizzazione con vettore dei figli , si ha una refernce al nodo del padre (succedeva nel CPM, predecessore dell msdos a sua volta predecessore di windows, si ha un grande spazio sprecato in quanto non si pensava di poter avere più di 16 file nella stessa directory)
- realizzazione basata su primo figlio prossimo fratello (utilizzato nei file system), un nodo avrà il puntatore al primo figlio più a sinistra che ha una lista di tutti i suoi fratelli, si ha anche il puntatore al padre
- realizzazione con vettore dei padri, si memorizza un vettore con il primo elemento il contenuto del primo elemento e il padre di quell'elemento

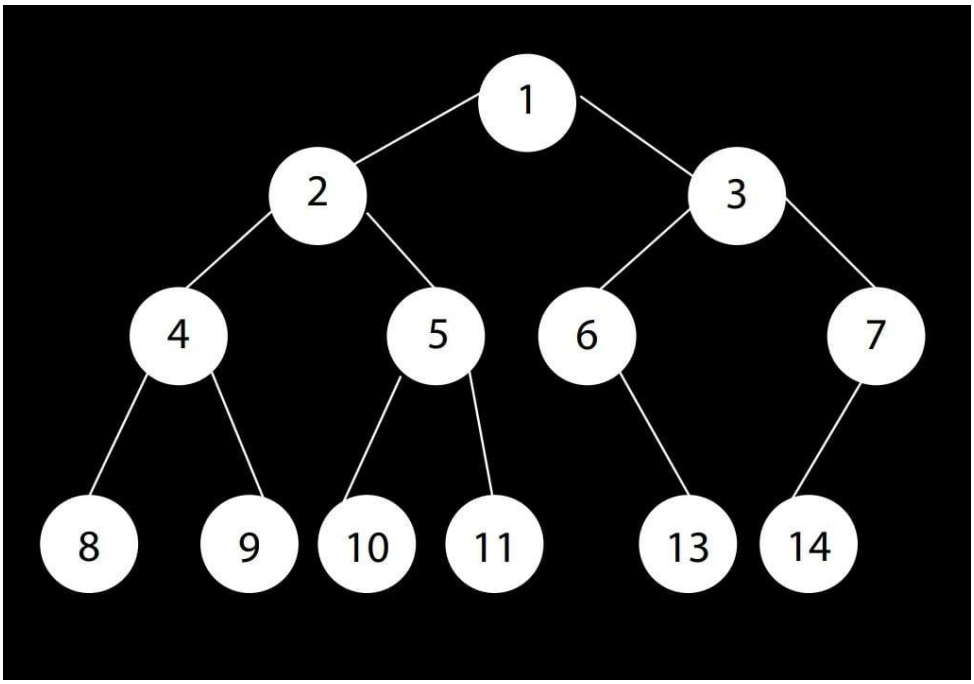


la profondità di un albero è il cammino che va dalla root a un determinato nodo, questa viene misurata in archi, il livello invece è l insieme di nodi alla stessa profondità, mentre l altezza di un alberò è la profondità che va dalla radice alle foglie.

BINARY TREE:

un albero binario è un albero in cui un solo nodo è la root quindi viene definito radicato e questo non può cambiare, ogni nodo ha al massimo 2 figli, il figlio sinistro e quello destro,

per memorizzare un albero binario si utilizza un oggetto che ha il puntatore al padre, un puntatore al ramo sinistro e quello destro



per trovare un dato in un albero si utilizza la "visita" dell'albero, cioè si passa attraverso tutti i nodi.

Le visite si dividono in due tipi:

- DPS:
  - di ogni albero si vistano i suoi sottoalberi e così via
- BFS:
  - si fa una visita per livelli

IMPLEMENTAZIONE IN CPP DELL'ALBERO BINARIO:

In base alla tipologia di albero si decide quanti nodi dare.

```cpp
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data; //DATO
    Node* left; //puntatore al nodo destro
    Node* right//puntatore al nodo sinistro
    //val sarebbe il dato da aggiungere
    Node(int val)
    {
        data = val;
        // i figli vengono dichiarati "NULL"
        left = NULL;
        right = NULL;
    }
};

int main()
{
    //definisco un nodo root
    Node* root = new Node(1);
    /* com'è l'albero ora
            1
            / \
        NULL NULL
    */

    root->left = new Node(2); //i due figli "NULL" diventano
    root->right = new Node(3); //figlio 2 e figlio 3 c
    /* l'albero risulta così
                1
               / \
              2     3
             / \   / \
             NULL NULL NULL NULL
    */
    root->left->left = new Node(4); //dalla root si scende per livelli in base alla
direzione
    /* 4 diventa figlio di 2
            1
           / \
          2   3
         / \   / \
         4 NULL NULL NULL
        / \
    NULL NULL
    */
    return 0;
}
```