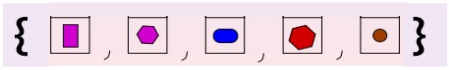
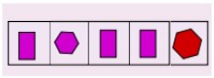
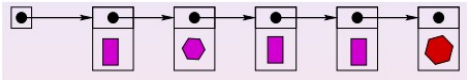
20. Scheda di lavoro (Liste, alberi e grafi) 30/11/2018

Lista

Una lista, o sequenza, è una successione finita ed eventualmente vuota di dati di un certo tipo t.



Può essere implementata utilizzando strutture dati differenti.

ArrayList Implementa le operazioni di una lista utilizzando un array	
LinkedList Implementa le operazioni di una lista facendo uso di reference Java offre la classe LinkedList che implementa la gestione puntatori rendendola trasparente agli sviluppatori.	

Difference between ArrayList and LinkedList in Java (facoltativo)

https://beginnersbook.com/2013/12/difference-between-arraylist-and-linkedlist-in-java/

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

ArrayList Vs LinkedList

1. Search: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).
   Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.
2. Deletion: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).
   Conclusion: LinkedList element deletion is faster compared to ArrayList.
   Reason: LinkedList's each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.
3. Inserts Performance: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.
4. Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbour nodes hence the memory consumption is high in LinkedList comparatively.

There are few similarities between these classes which are as follows:

- Both ArrayList and LinkedList are implementation of List interface.
- They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.

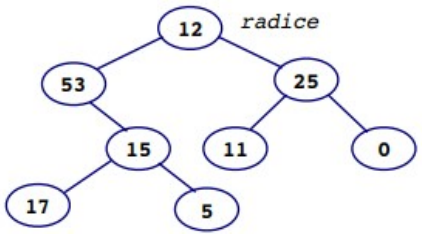
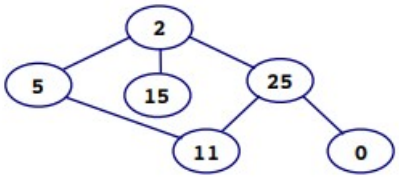
When to use LinkedList and when to use ArrayList?

1. As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.
2. Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet..

Albero

Un albero è una collezione di nodi e di archi, per cui

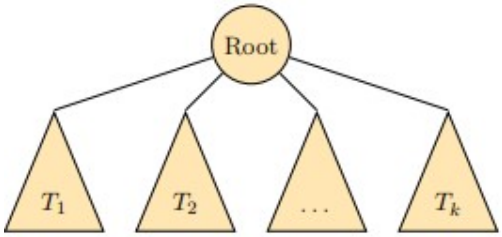
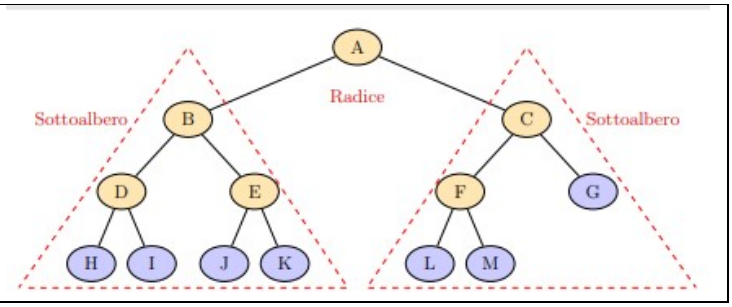
- ogni nodo, eccetto uno, detto radice, ha un solo predecessore e 0 o più successori, la radice non ha predecessori
- esiste un unico cammino dalla radice ad ogni altro nodo
- ogni nodo contiene un valore di un qualche tipo

Un albero	Una struttura che non è un albero (ma è un grafo)
	 In effetti l'albero è un caso particolare di grafo.

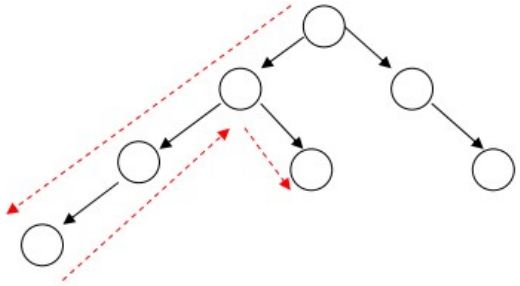
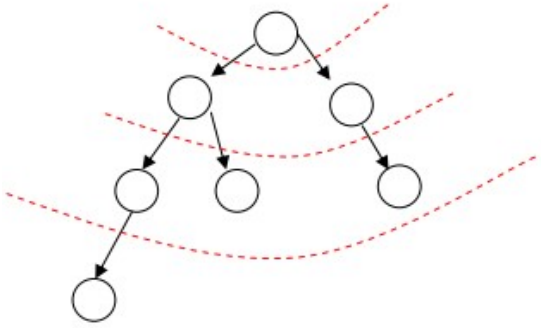
Definizione ricorsiva di albero:

Un albero è dato da:

- un insieme vuoto, oppure
- un nodo radice e zero o più sottoalberi, ognuno dei quali è un albero;
- la radice è connessa alla radice di ogni sottoalbero con un arco.

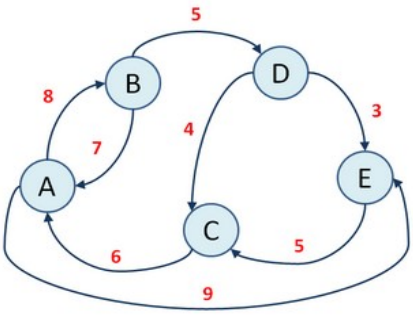
Visita di un albero, cioè visita di tutti i suoi nodi

Visità in profondità - Depth-First Search (DFS)	**Visita in ampiezza - Breadth First Search (BFS)**
Per visitare un albero, si visita ricorsivamente ognuno dei suoi sottoalberi  Algoritmo ricorsivo oppure iterativo con stack perché così i nodi adiacenti all'ultimo nodo visitato sono i primi ad essere visitati	Ogni livello dell'albero viene visitato, uno dopo l'altro, a partire dalla radice  Richiede una queue

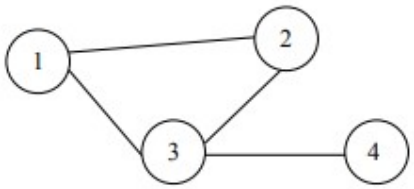
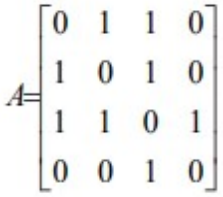
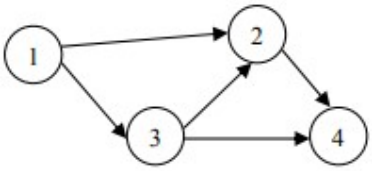
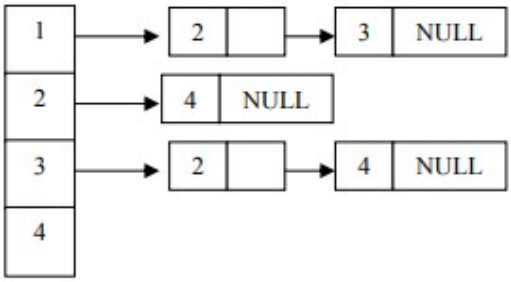
Grafo

Un grafo è una coppia di insiemi:

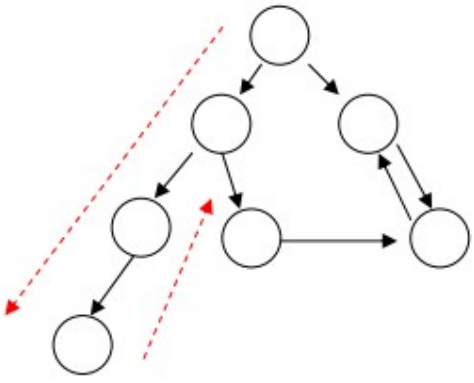
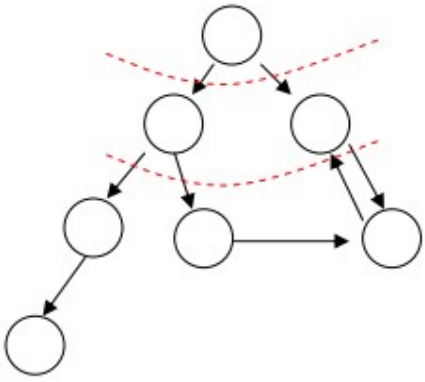
- **un insieme di nodi**
- **un insieme di archi (che possono essere pesati)**



Strutture dati per memorizzare l'ADT grafo

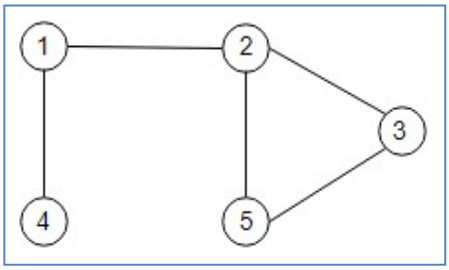
Matrice di adiacenza	**Liste di adiacenza per ciascun nodo**
  a	 

Visita di un grafo

Visità in profondità **Depth-First Search (DFS)**	**Visita in ampiezza** **Breadth First Search (BFS)**
	
Visita ricorsiva, oppure visita iterativa con uno stack	Algoritmo solo iterativo, richiede una coda.

Codice Java visita grafo DFS ricorsiva - facoltativo

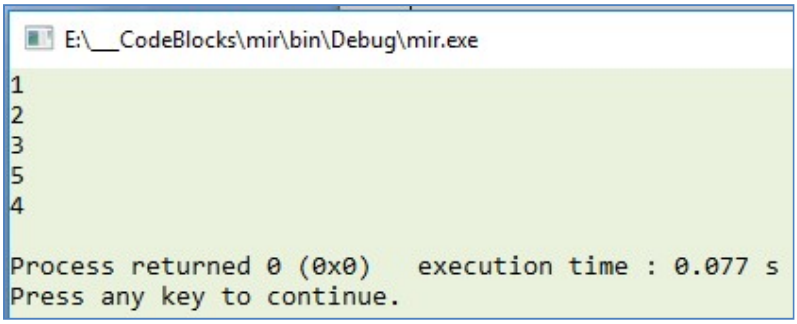
```
package pkg4binfo;
import java.io.*;
import java.util.ArrayList;
public class Main {
   public static void main(String[] args) {
      ArrayList nodo[] = new  ArrayList [6]; //vettore di liste di adiacenza. Ogni elemento del vettore rappresenta un nodo
      for (int i=0; i<6; i++)
         nodo[i] = new ArrayList();
      boolean visitato[] = new boolean[6]; //vettore che indiva quali nodi sono già stati visitati
      for (int i=1; i<6; i++)
         visitato[i] = false; //all'inizio nessun nodo è stato visitato
      //inserisco tuttti gli archi del grafo
      nodo[1].add(2); nodo[2].add(1);
      nodo[1].add(4); nodo[4].add(1);
      nodo[2].add(3); nodo[3].add(2);
      nodo[2].add(5); nodo[5].add(2);
      nodo[3].add(5); nodo[5].add(3);

      //visualizzo nodi e archi del grafo
      for (int i=1; i<6;i++)
         System.out.println("nodo "+i + " --> archi " + nodo[i].toString());

      //chiamo il metodo per la visita in profondità a partire dal nodo 1
      // dfs, cioè depth-first search
      System.out.println("\nvisita in prodondità ddel grafo (dfs - depth-first search) ");
      dfs(1, nodo, visitato);
   }//fine main

//visita in profondità ricorsiva del grafo a partire dal nodo n

static void dfs(int n, ArrayList nodo[], boolean visitato[]) {
   if (visitato[n]) return;

   visitato[n] = true; //visito l'elemento corrente
   // elaboro il nodo n, in questo caso lo mando a video
   System.out.println("nodo " + n);

  //scorro la lista di adiacenza del nodo corrente
   for (Object x : nodo[n])  { //per ogni elemento della lista degli archi del nodo n
       dfs( (int)x, nodo, visitato);
   }
} //fine metodo dfs

}  // fine classe
```

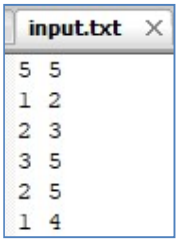


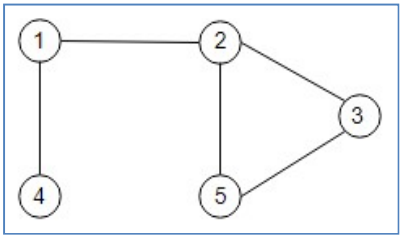

```
Output - 4Binfo (run)
run:
nodo 1 --> archi [2, 4]
nodo 2 --> archi [1, 3, 5]
nodo 3 --> archi [2, 5]
nodo 4 --> archi [1]
nodo 5 --> archi [2, 3]

visita in profondità ddel grafo
(dfs - depth-first search)
nodo 1
nodo 2
nodo 3
nodo 5
nodo 4
```

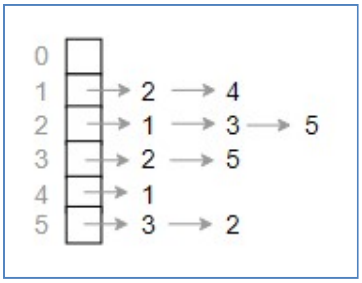
Codice C++ visita grafo DFS ricorsiva - facoltativo

```
#include <iostream>
#include <list>
#include <fstream>
using namespace std;

list<int> liste[100]; //vettore di liste di adiacenza. Ogni elemento del vettore rappresenta un nodo
int visitato[100];
int N, M; //N=nodi, M=archi

//grafo: visita in profondità ricorsiva a partire dal nodo n
void dfs(int n) {
   if (visitato[n]) return;
   visitato[n] = true;
   // elaboro il nodo n
   cout <<n << endl;
  //scorro la lista di adiacenza del nodo
   for (auto x: liste[n]) {
       dfs(x);
   }
}

int main(int argc, char** argv) {

          ifstream in("input.txt");
          ofstream out("output.txt");
          in >> N >> M;
           for (int i=0; i<N; i++)
           visitato[i] = false;
           for (int i=0; i<M; i++)  {
                    int a,b;
                    in >> a >> b;
                    liste[a].push_back(b);
                    liste[b].push_back(a);
          }

          dfs(1);

          return 0;
}
```









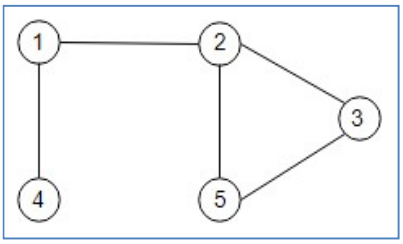
Codice C++ visita grafo DFS iterativa (bugatti) - facoltativo

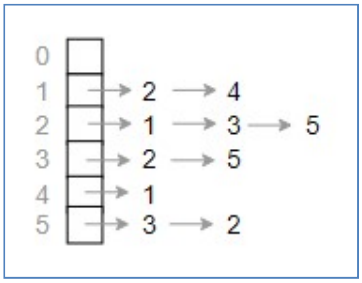
```
#include <iostream>
#include <list>
#include <stack>
#include <fstream>
using namespace std;
   list <int> liste[100]; //vettore di liste di adiacenza. Ogni elemento del vettore rappresenta un nodo
   int visitato[100];
   stack <int> pila;
   int N,M; //N=nodi, M=archi


void visita_profondita(int n) {
        pila.push(n);
        while (!pila.empty()) {
                int corrente = pila.top();
            pila.pop();
            if (visitato[corrente] == false) {
                visitato[corrente] = true;
                cout << corrente << endl;
                for (auto x:liste[corrente])
                    pila.push(x);
                //si può scrivere anche in forma più estesa
                // for (list <int>::iterator i = liste[corrente].begin(); i!=liste[corrente].end(); i++)
                //      pila.push(*i);
            }
        }
}

int main(int argc, char** argv) {
        ifstream in("input.txt");
        ofstream out("output.txt");
        in >> N >> M;
         for (int i=0; i<N; i++)
         visitato[i] = false;
         for (int i=0; i<M; i++)  {
                int a,b;
                in >> a >> b;
                liste[a].push_back(b);
                liste[b].push_back(a);
        }

         visita_profondita(1); // visita il grafo a a partire dal nodo 1

        return 0;
}
```

```
E:\__CodeBlocks\mir\bin\Debug\mir.exe
1
4
2
5
3

Process returned 0 (0x0)   execution time : 6.236 s
Press any key to continue.
```

input.txt

```
5 5
1 2
2 3
3 5
2 5
1 4
```





Codice C++ visita grafo BFS (Breadth-first search) iterativa (bugatti)

```
#include <iostream>
#include <list>
#include <queue>
#include <fstream>
using namespace std;
   list <int> liste[100]; //vettore di liste di adiacenza. Ogni elemento del vettore rappresenta un nodo
   int visitato[100];
   queue <int> coda;
   int N,M; //N=nodi, M=archi

//grafo: visita in ampiezza iterativa a partire dal nodo n
void bfs(int nodo_partenza) {
   coda.push(nodo_partenza);
   while (!coda.empty()) {
      int corrente = coda.front();
      coda.pop();
      if (visitato[corrente] == false) {
            visitato[corrente] = true;
           //elabora elemento corrente
            cout << corrente << endl;
           for (auto x:liste[corrente])
                     coda.push(x);
      }
   }
}
int main(int argc, char** argv) {
           ifstream in("input.txt");
           ofstream out("output.txt");
           in >> N >> M;
            for (int i=0; i<N; i++)
                     visitato[i] = false;
            for (int i=0; i<M; i++)  {
                     int a,b;
                     in >> a >> b;
                     liste[a].push_back(b);
                     liste[b].push_back(a);
           }

           bfs(1);

           return 0;
}
```

```
E:\__CodeBlocks\mir\bin\Debug\mir.exe
1
2
4
3
5

Process returned 0 (0x0)   execution time : 4.289 s
Press any key to continue.
```

```
input.txt
5 5
1 2
2 3
3 5
2 5
1 4
```