PROGETTO DI ALGORITMI E STRUTTURE DATI

ANNO: 2014/2015

NOME: MATTIA

COGNOME: D'AUTILIA

MATRICOLA: 5765968

Un **albero A** é una coppia di insiemi **A = (V,E)** dove **V** é un insieme non vuoto di elementi detti vertici o nodi ed **E ={(x,y)|x,y∈V}** é un insieme di coppie di elementi di V detti archi o rami, tali che per ogni coppia di nodi dell'albero esiste uno ed un solo cammino che li unisce.

Nello specifico, un **albero m-ario** è un nodo esterno oppure un nodo interno connesso ad una sequenza ordinata di m alberi m-ari. Normalmente i nodi di un albero m-ario sono rappresentati da strutture con **m** puntatori.

Quindi, la mi applicazione java, con l'utilizzo di due classe, **Nodo_m_ario** e **Albero_m_ario**, sotto descritte, crea attraverso un insieme di metodi, un albero m-ario con **m (quantità scelta dall'utente) figli**, ognuno dei quali a partire dal nodo m-ario **radice** sono tutti dei dei nodi m-ari collegati con la relazione **padre-figli**.

Per entrambe le classi, ho usato, in base a quello che il metodo doveva gestire e a quello che doveva restituire come ritorno, tre tipi di pacchetti offerti da java (**java.util.ArrayList; java.util.LinkedList),** in più per la classe Albero_m_ario ho utilizzato il pacchetto java(**java.util.Stack**), comunque questi pacchetti contengono tre tipi di strutture dati con tutti i metodi (inserimento, cancellazione ecc.) per facilitare la gestione dei dati.

La **Classe Nodo_m_ario,** gestisce le **informazione,** il **padre,** l'**ArrayList figli,** il **livello** e l'**arietà m, di un nodo** e tutti i metodi in questa classe gestiscono tutto ciò che riguarda il singolo nodo.

CLASSE NODO_M_ARIO

```
import java.util.ArrayList;
import java.util.LinkedList;

@SuppressWarnings("static-access")
public class Nodo_m_ario<T> {

	private T info;
	private Nodo_m_ario<T> padre;
	private ArrayList<Nodo_m_ario<T>> figli;
	private int livello;
	private  Albero_m_ario<T> A;
	private int m=A.getarietà();
```

Il Costruttore della Classe Nodo_m_ario, ha come parametro un valore di tipo T, che costituisce l'informazione del nodo, assegna al padre il valore null, e crea un ArrayList di nodi_m_ari di lunghezza m=arietà dell'albero, ai quali assegna a tutti il valore null;

```
	// Costruttore
	public Nodo_m_ario(T x){
```

```
            info=x;
            padre=null;
            figli=new ArrayList<Nodo_m_ario<T>>();
            for(int i=0;i<=m-1;i++){
                  figli.add(null);
            }
      }
```

Il metodo changeInfo, ha come parametro un valore di tipo T, e ha la funzione di cambiare il valore dell'informazione di un nodo.

```
      // Cambiare il contenuto di un nodo.
      public void changeInfo(T info){
            this.info=info;
      }
```

Il metodo infoFigli, crea una LinkedList di valori T con il nome inf, alla quale assegna, tramite un ciclo for sull' ArrayList figli di un nodo, tutti i nodi con una informazione, e la restituisce.

```
      // Restituire la lista delle informazioni dei figli di un nodo.
      public LinkedList<T> infoFigli(){

            LinkedList<T> inf=new LinkedList<T>();
            for(int i=0; i<=m-1;i++){
                  if(figli.get(i)!=null){
                        inf.add(figli.get(i).getInfo());
                  }
            }
            System.out.println("Le informazioni dei figli del nodo sono= "+inf);
            return inf;
      }
```

Il metodo nFigli, è molto simile al metodo infoFigli, solamente che non usa nessuna LinkedList, in quanto come ritorno non restituisce le informazione dei figli di un nodo ma la quantità dei figli con informazione di un nodo, sempre però utilizzando un ciclo for sull' ArrayList figli di un nodo.

```
      // Restituire il numero di figli non vuoti di un nodo.
      public int nFigli(){
            int nNodiFigli=0;
            for(int i=0;i<=m-1;i++){
                  if(figli.get(i)!=null){
                        nNodiFigli++;
                  }
            }
      System.out.println("Il numero di figli non vuoti del nodo è=   "+nNodiFigli);
      return nNodiFigli;
      }
```

Il metodo getInfo, restituisce un valore di tipo T, che corrisponde all'informazione dei un nodo.

```
      // Restituisce l'Informazione di un nodo
      public T getInfo() {
            return info;
```

```
	}
```

Il metodo setInfo, è un mutatore che assegna ad uno specifico nodo, un valore parametro di tipo T che corrisponde all'informazione di un nodo.

```
	// Mutatore informazione di un nodo
	public void setInfo(T info) {
		this.info = info;
	}
```

Il metodo getPadre, restituisce un nodo m-ario, che corrisponde al nodo padre di un nodo.

```
	// Restituisce il padre di un nodo
	public Nodo_m_ario<T> getPadre() {
		return padre;
	}
```

Il metodo setPadre, è un mutatore che assegna ad uno specifico nodo, un valore parametro nodo padre corrisponde al padre di un nodo.

```
	// Mutatore padre di un nodo
	public void setPadre(Nodo_m_ario<T> padre) {
		this.padre = padre;
	}
```

Il metodo getFigli, restituisce un ArrayList di nodi, che corrisponde ai nodi figli di un nodo.

```
	// Restituisce l'Array dei figli di un nodo
	public ArrayList<Nodo_m_ario<T>> getFigli() {
		return figli;
	}
```

Il metodo setFigli, è un mutatore che assegna ad uno specifico nodo, un ArrayList figli contenente i nodi figli di un nodo.

```
	// Mutatore figli di un nodo
	public void setFigli(ArrayList<Nodo_m_ario<T>> figli) {
		this.figli = figli;
	}
```

Il metodo getLivello, restituisce un intero, che corrisponde al livello di un nodo.

```
	// Restituisce livello di un nodo
	public int getLivello() {
		return livello;
	}
```

Il metodo setLivello, è un mutatore che assegna ad uno specifico nodo, un valore intero livello che corrisponde al livello di un nodo.

```
	// Mutatore livello di un nodo
	public void setLivello(int livello) {
		this.livello = livello;
	}
```

Il metodo printPadre, stampa il padre di un nodo; se il nodo è la radice stampa “Il padre del nodo è inesistente, perché è una radice.”

```
	// Stampa padre di un nodo
	public void printPadre(){
		if(getPadre()!=null){
			System.out.println("Il padre del nodo è= "+padre.getInfo());
		}else{
			System.out.println("Il padre del nodo è inesistente, perchè è una
			radice.");
		}
	}
```

Il metodo printInfo, stampa l’informazione di un nodo.

```
	// Stampa informazione di un nodo
		public void printInfo(){
			System.out.println("L'Informazione del nodo è= "+getInfo());
		}
}
```

La **Classe Albero_m_ario**, gestisce il numero dei nodi **nNodi**,il numero dei nodi con informazione **nNodiInf**, il nodo **radice**, l’altezza **h** e l’**arietà m, di un Albero** e tutti i metodi in questa classe gestiscono tutto ciò che riguarda l’intero Albero.

CLASSE ALBERO_M_ARIO

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;

public class Albero_m_ario<T> {
	private static int m_ario;
	private int nNodi; // numero Nodi
	private int nNodiInf; // numero Nodi con informarzione
	private Nodo_m_ario<T> radice; // nodo radice
	private int h;
```

Il Costruttore della Classe Albero_m_ario, ha come parametro un intero m, che costituisce l’ arietà dell’albero, assegna all’altezza h=0, al numero dei nodi nNodi=0 e al numero dei nodi con informazione nNodiInf=0;

```
	// Costruttore
	public Albero_m_ario(int m){
		m_ario=m;
		h=0;
		nNodi=0;
		nNodiInf=0;
	}
```

Il metodo radice, controllo se il nodo radice è uguale a null, in tal caso stampa “radice non presente”, altrimenti restituisce il nodo radice.

```
	// Restituire radice di un albero
	public Nodo_m_ario<T> radice(){

		if (radice==null){
			System.out.println("radice non presente");
		    return null;
		 }
		return radice;
	}
```

Il metodo addRadice, se l'albero è vuoto, assegna al nodo radice, un valore info di tipo T, che corrisponde all'informazione del nodo radice e restituisce il nodo radice, altrimenti stampa "L'Albero non è vuoto".

```
	// Inserisci radice in un albero vuoto
	public Nodo_m_ario<T> addRadice(T info){

		if(!isEmpty()){
			System.out.println("L'Albero non è vuoto");
			return null;
		}else{
			radice=new Nodo_m_ario<T>(info);
			radice.setLivello(0);
			nNodiInf++;
			nNodi=nNodiInf+m_ario;
			h++;
			return radice;
		}
	}
```

Il metodo addFiglio, assegna e restituisce un nodo figlio, il quale deve avere come parametro il padre del al quale assegnare il nodo figlio, l'informazione da assegnare al nodo figlio e l'arieta numFiglio che corrisponde al i-esimo figlio del nodo padre. Prima di tutto si crea il nodo e gli si assegna l'informazione e si controlla se il numFiglio e compreso nell'arietà m dell'albero, altrimenti stampa "Arietà inesistente". Dopo di chè si controlla se la posizione dove aggiungere il nodo è libero, altrimenti stampa "Posizione m-aria del nodo occupata". Se la posizione è libera assegna nell' ArrayList figli il nuovo nodo figlio, al nuovo nodo figlio il padre e si aggiorna il livello del figlio. Aggiorni la quantità dei nodi totali e dei nodi con informazione e si controlla l'altezza dell'albero.

```
	// Inserire un nuovo nodo V come figlio i-esimo di un nodo U, gi`a presente
nell'albero. Il metodo riceve in ingresso il padre del nuovo nodo, l'informazione di
quest'ultimo e un intero i ∈{1,2,...,m}per indicare se V sar`a il primo, il secondo, ...,
l'm-esimo figlio di U.
	public Nodo_m_ario<T> addFiglio(Nodo_m_ario<T> padre,T info,int numFiglio){

		Nodo_m_ario<T> nuovoNodo=new Nodo_m_ario<T>(info);
		if(numFiglio>=0 && numFiglio<=m_ario-1){
			if(padre.getFigli().get(numFiglio)==null){
				padre.getFigli().set(numFiglio, nuovoNodo);
				nuovoNodo.setPadre(padre);
				nuovoNodo.setLivello(padre.getLivello()+1);
				nNodiInf++;
```

```
                        nNodi=nNodi+m_ario;
                        if(nuovoNodo.getLivello()>=h){
                                h=nuovoNodo.getLivello();
                        }
                }else{
                        System.out.print("Posizione m-aria del nodo già occupata");
                        return null;
                }
        }else{
                System.out.print("Arietà inesistente");
                return null;
        }
        return nuovoNodo;
    }
```

Il metodo addAlbero, innesta in un Albero B in un altro Albero. Prima di tutto controlla se l'arietà data come parametro è compresa tra il 0 e il massimo arietà dell'albero, altrimenti stampa "Arietà inesistente", dopo di chè controlla se nell'ArrayList figli la posizione numFiglio è libera altrimenti stampa "Posizione m_aria del nodo già occupata. Dopo di che inserisce il nodo radice dell'Albero da innestare nell'ArrayList figli, gli assegna il padre, e aggiorna i numero dei nodi totali e dei nodi con informazione. Richiama il metodo aggiornaLivelli e ricontrolla l'altezza dell'Albero.

```
    // Inserire un sottoalbero B in modo che la radice di B sia figlia i-esima di un nodo
u dell'albero.
    public void addAlbero(Albero_m_ario<T> B,Nodo_m_ario<T> padre, int numFiglio){
        if(numFiglio>=0 && numFiglio<=m_ario -1){
            if(padre.getFigli().get(numFiglio)==null){
                padre.getFigli().set(numFiglio, B.radice());
                B.radice().setPadre(padre);
                aggiornaLivelli(B.radice());
                nNodiInf=nNodiInf+B.getnNodiInf();
                nNodi=nNodi+B.getnNodi()-1;
                if(B.radice().getLivello()>h){
                    h=h+B.getH()+1;
                }
                if(B.radice().getLivello()<h){
                    if(B.getH()+B.radice().getLivello()>h){
                        h=B.getH()+B.radice().getLivello();
                    }
                }
                if(B.radice().getLivello()==h){
                    h=h+B.getH();
                }
                B.radice=null;
            }else{
                System.out.print("Posizione m-aria del nodo già occupata");
            }
        }else{
            System.out.print("Arietà inesistente");
        }
    }
```

Il metodo newRadice, creare, sostituisce e restituisce, una nuova radice rispetto a quella già esistente. Ha come parametri l'Albero dal quale sul quale inserire la nuova radice, l'informazione della nuova radice e la m-posizione alla quale collegare la radice

dell'albero come figlia della nuova radice. Prima di tutto crea la nuova radice e gli assegna l'informazione; poi controlla se il nodo radice dell'Albero è diverso da null, altrimenti stampa "Radice non presente quindi non sostituibile". Dopo controlla se la m-posizione è compresa nell'intervallo, altrimenti stampa "Arietò inesistente"; si assegna nell'ArrayList figli della nuova radice, la radice del vecchio Albero nella m-posizione parametro, aggiorna i livelli e assegna alla nodo radice dell'Albero il suo padre cioè la nuova radice. Aggiorna la quantità totale dei nodi, quelli con informazione e l'altezza.

```
	// Inserire una nuova radice in un albero non vuoto in modo che la vecchia radice sia figlia i-esima della nuova, dove i ∈{1,2,...,m}.
	public Nodo_m_ario<T> newRadice(Albero_m_ario<T> B,T info,int numFiglio){
		Nodo_m_ario<T> newRadice=new Nodo_m_ario<T>(info);
		if(B.radice()!=null){
			if(numFiglio>=0 && numFiglio<=m_ario-1){
				newRadice.getFigli().set(numFiglio, B.radice());
				B.radice().setPadre(newRadice);
				aggiornaLivelli(B.radice());
				radice=newRadice;
				nNodiInf++;
				nNodi=nNodi+m_ario;
				h++;
			}else{
				System.out.print("Arietà inesistente");
				return null;
			}
		}else{
			System.out.print("Radice non presente quindi non sostituibile");
			return null;
		}
		return newRadice;
	}
```

Il metodo visitaAnticipata, attraversa in profondità l'albero e restituisce una lista con le informazioni contenute nei nodi incontrati(visita anticipata iterativa). Crea una lista per inserire le informazioni dei nodi e un pila per gestire i nodi incontrati di volta in volta nell'albero. Quindi controlla se c'è almeno un nodo nell'albero, e inserisce tramite il metodo push del pacchetto Stack, nella pila, il nodo radice; altrimenti stampa "L'Albero non contiene nodi". Dopo di che attua un ciclo While finche la pila non è vuota; viene creato un nodo temporaneo al quale viene assegnato il nodo che si trova nella posizione top della pila, estraendolo con il metodo pop e la sua informazione viene inserita nella lista. Del nodo estratto poi si inseriscono nella pila tutti i suoi figli non nulli tramite la visita dell'ArrayList figli. Alla fine si restituisce la lista con le informazione dei nodi.

```
	// Attraversare l'albero in profondit`a e restituire la lista delle informazioni contenute nei nodi cos`ı incontrati (visita anticipata).
	public LinkedList<T> visitaAnticipata(){
		LinkedList<T> anti = new LinkedList<T>();
		Stack<Nodo_m_ario<T>> pila=new Stack<Nodo_m_ario<T>>();
		if(nNodi!=0){
			pila.push(radice);
			while(!pila.isEmpty())    {
				Nodo_m_ario<T> temp=(Nodo_m_ario<T>) pila.pop();
				anti.add(temp.getInfo());
```

```
                        for(int i=m_ario-1; i>=0;i--){
                                if(temp.getFigli().get(i)!=null){
                                        pila.push(temp.getFigli().get(i));
                                }
                        }
                }
        }else{
                System.out.print("L'ALbero non contiene nodi");
        }
        System.out.println("Visità anticipata= " +anti);
        return anti;
    }
```

Il metodo visitaLivelli, attraversa in ampiezza l'albero e restituisce una lista con le informazioni contenute nei nodi incontrati(visita per Livelli iterativa). Crea una lista per inserire le informazioni dei nodi e un ArayList coda per gestire i nodi incontrati di volta in volta nell'albero. Quindi controlla se c'è almeno un nodo nell'albero, e inserisce tramite il metodo add, nella coda, il nodo radice; altrimenti stampa "L'Albero non contiene nodi". Dopo di che attua un ciclo While finche la coda non è vuota; viene creato un nodo temporaneo al quale viene assegnato il nodo che si trova nella posizione 0 della coda, estraendolo con il metodo remove e la sua informazione viene inserita nella lista. Del nodo estratto poi si inseriscono nella coda tutti i suoi figli non nulli tramite la visita dell'ArrayList figli. Alla fine si restituisce la lista con le informazione dei nodi.

```
    // Attraversare l'albero in ampiezza e restituire la lista delle informazioni
contenute nei nodi cos`ı incontrati (visita per livelli).
    public LinkedList<T> visitaLivelli(){
        LinkedList<T> livel = new LinkedList<T>();
        ArrayList<Nodo_m_ario<T>> coda = new ArrayList<Nodo_m_ario<T>>();
        if(nNodi!=0){
                coda.add(radice);
                while(!coda.isEmpty()){
                        Nodo_m_ario<T> temp=(Nodo_m_ario<T>) coda.remove(0);
                        livel.add(temp.getInfo());
                        for(int i=0; i<=m_ario-1;i++){
                                if(temp.getFigli().get(i)!=null){
                                        coda.add(temp.getFigli().get(i));
                                }
                        }
                }
        }else{
                System.out.print("L'ALbero non contiene nodi");
        }
        System.out.println("Visità per Livelli= " +livel);
        return livel;
    }
```

Il metodo aggiornaLivelli, è molto simile alla visitaLivelli, solamente che al posto di salvare in una lista l informazioni dei nodi, aggiorna i livelli dei nodi trovati d i volta in volta con una visita per livelli, partendo da un nodo dato come parametro,

```
    // Aggiorna Livelli nodi
    public void aggiornaLivelli(Nodo_m_ario<T> x){
        ArrayList<Nodo_m_ario<T>> coda = new ArrayList<Nodo_m_ario<T>>();
```

```
		if(nNodi!=0){
			coda.add(x);
			while(!coda.isEmpty()){
				Nodo_m_ario<T> temp=(Nodo_m_ario<T>) coda.remove(0);
				temp.setLivello(temp.getPadre().getLivello()+1);
				for(int i=0; i<=m_ario-1;i++){
					if(temp.getFigli().get(i)!=null){
						coda.add(temp.getFigli().get(i));
					}
				}
			}
		}else{
			System.out.print("L'ALbero non contiene nodi");
		}
	}
```

Il metodo visitaPosticipata, attraversa in maniera ricorsiva l'Albero, restituendo una lista con le informazioni dei nodi che ha visitato nel corso dell'attraversamento. In questo metodo viene richiamato il metodo principale dell'attraversamento ricorsivo, visitaRicorsivaPost.

```
	// Restituire la lista delle informazioni contenute nei nodi che si incontrano effettuando una visita posticipata dell'albero.
	public LinkedList<T> visitaPosticipata() {
		LinkedList<T> post = new LinkedList<T>();
		visitaRicorsivaPost(radice, post);
		System.out.println("Visità posticipata= " +post);
		return post;
	}
```

Il metodo visitaRicorsivaPost, ha come parametro un nodo x dal quale iniziare l'attraversamento e la lista post dove inserire le informazioni. Prima di tutto controlla se il nodo è uguale a null in tal caso fa ritornare null, altrimenti parte un ciclo for sui figli del nodo richiamando il metodo visitaRicorsivaPost. Alla fine assegna alla lista post l'informazione del nodo.

```
	// Visita Posticitata Ricorsiva
	private void visitaRicorsivaPost(Nodo_m_ario<T> x, LinkedList<T> post) {
		if (x == null) {
			return;
		}
		for (int i = 0; i <= m_ario-1; i++) {
			visitaRicorsivaPost(x.getFigli().get(i), post);
		}
		post.add(x.getInfo());
	}
```

Il metodo visitaSimmetrica, attraversa in maniera ricorsiva l'Albero, restituendo una lista con le informazioni dei nodi che ha visitato nel corso dell'attraversamento. In questo metodo viene richiamato il metodo principale dell'attraversamento ricorsivo, visitaRicorsivaSim.

```
	// Effettuare la visita simmetrica dell'albero e restituire la lista delle informazioni contenute nei nodi cos`ı incontrati, dove per visita simmetrica di un albero m-ario si intende la visita simmetrica dei sottoalberi dal primo all'bm 2 c-esimo, seguita dalla visita della radice, seguita dalla visita simmetrica dei sottoalberi dal dm 2 e-
```

esimo all'm-esimo (per visita di un nodo si intende l'accesso all'informazione ivi contenuta, per visita di un sottoalbero si intende la visita di tutti i nodi del sottoalbero, una ed una sola volta).

```
	public LinkedList<T> visitaSimmetrica() {
		LinkedList<T> sim = new LinkedList<T>();
		visitaRicorsivaSim(radice, sim);
		System.out.println("Visità simmetrica= " +sim);
		return sim;
	}
```

Il metodo visitaRicorsivaSim, ha come parametro un nodo x dal quale iniziare l'attraversamento e la lista post dove inserire le informazioni. Prima di tutto controlla se il nodo è uguale a null in tal caso fa ritornare null, altrimenti parte un ciclo for sui figli del nodo che si trovano dalla posizione 0 al m-aria/2 richiamando il metodo visitaRicorsivaPost. Poi assegna alla lista post l'informazione del nodo e successivamente fa partire un'altri ciclo for che va dalla posizione m-aria/2 fino alla m_aria richiamando il metodo visitaRicorsivaPost e saltando il primo ciclo for assegna alla lista post l'informazione del nodo.

```
	// Visita Simmetrica Ricorsiva
	private void visitaRicorsivaSim(Nodo_m_ario<T> x, LinkedList<T> sim) {
		if (x == null) {
			return;
		}
		for (int i = 0; i < (m_ario-1)/2 ; i++) {
			visitaRicorsivaSim(x.getFigli().get(i), sim);
		}
		sim.add(x.getInfo());
		for(int i=(m_ario-1)/2; i<=m_ario-1;i++){
			visitaRicorsivaSim(x.getFigli().get(i), sim);
		}
	}
```

Il metodo getnNodiInf, restituisce il contatore dei Nodi con informazione.

```
	// Restituisce contatore numero Nodi con informazione
	public int getnNodiInf() {
		return nNodiInf;
	}
```

Il metodo printnNodiInf, stampa il contatore dei Nodi con informazione.

```
	// Stampa contatore numero Nodi con informazione
	public void printnNodiInf(){
		System.out.println("Numero nodi con Informazione = "+ nNodiInf);
	}
```

Il metodo getnNodi, restituisce il contatore totale dei Nodi.

```
	// Restituisce contatore numero Nodi totali
	public int getnNodi() {
		return nNodi;
	}
```

Il metodo printnNodi, stampa il contatore totale dei Nodi.

```
	// Stampa contatore numero Nodi totali
	public void printnNodi(){
		System.out.println("Numero totale dei nodi = "+ nNodi);
	}
```

Il metodo getH, restituisce l'altezza dell'Albero.

```
	// Restituire l'altezza dell'albero
	public int getH() {
		return h;
	}
```

Il metodo printH, stampa l'altezza dell'Albero.

```
	// Stampa padre di un nodo
	public void printH(){
		System.out.println("L'altezza dell'ALbero è= "+getH());
	}
```

Il metodo isEmpty, controlla se l'Albero è vuoto o no.

```
	// Controllo se albero è vuoto, cioè senza radice
	public boolean isEmpty() {
		return radice==null;
	}
```

Il metod getarietà, restituisce il valore statico dell'arieta dell'Albero.

```
	// Restituisce arietà Albero
	public static int getarietà(){
		return m_ario;
	}
```

Il metodo printRadice, stampa la radice dell'Albero.

```
	// Stampa radice di un Albero
	public void printRadice(){
		System.out.println("La Radice dell'Albero è= "+radice.getInfo());
	}

}
```