|  |
| --- |
| Formulario di  Programmazione 2 |
|  |
| Rosario Terranova |
|  |
|  |
|  |

|  |
| --- |
| <https://rosarioterranova.github.io/> |

Sommario

[Eccezioni 2](#_Toc410122800)

[Interfacce 2](#_Toc410122801)

[Java I/O 3](#_Toc410122802)

[Analisi di Complessità 6](#_Toc410122803)

[Strutture dati 8](#_Toc410122804)

[Liste Concatenate 9](#_Toc410122805)

[Liste doppiamente concatenate 10](#_Toc410122806)

[Pila 12](#_Toc410122807)

[Coda 12](#_Toc410122808)

[Ricorsione 14](#_Toc410122809)

[Alberi 16](#_Toc410122810)

[Algoritmi di ordinamento 20](#_Toc410122811)

[Codici JAVA 23](#_Toc410122812)

[Eccezioni e interfacce 23](#_Toc410122813)

[Java I/O 25](#_Toc410122814)

[Liste concatenate 29](#_Toc410122815)

[Pila 34](#_Toc410122816)

[Coda 35](#_Toc410122817)

[Ricorsione 36](#_Toc410122818)

[BST 38](#_Toc410122819)

[Ordinamento semplice 43](#_Toc410122820)

[Ordinamento avanzato 44](#_Toc410122821)

|  |  |
| --- | --- |
| Eccezioni | |
| *Definizione* | Evento che si scatena durante l’esecuzione di un programma per indicare un errore nell’esecuzione del programma il quale è necessario gestirlo.  Le eccezioni causano l’**interruzione** del normale flusso delle operazioni.  Sono conseguenze a varie anomalie: il malfunzionamento fisico di un dispositivo di sistema, la mancata inizializzazione di oggetti particolari quali ad esempio connessioni verso basi dati, o semplicemente errori di programmazione come la divisione per zero di un intero. |
| *try … catch* | È possibile che venga intrapresa una serie di azioni speciali che consentano l’esecuzione del programma. È possibile catturare l’errore con il meccanismo **try-catch.**  La clausola **throws** (lancia) è una segnalazione all’utente per avvisarlo che si può verificare una particolare eccezione che, se non correttamente gestita, provoca l’arresto del programma |
| *Eccezioni non*  *dichiarabili* | Questi tipi di eccezioni non devono essere dichiarate, poiché andrebbero dichiarate ovunque nel programma, ma vengono gestite in automatico dalla JVM. Il programmatore deve comunque stare attento a non scatenarle. I tipi di eccezioni più comuni sono:   * **NullPointerException:** si sta cercando di accedere ai campi di un oggetto ancora nullo; * **IndexOutOfBoundsException:** si stanno cercando di superare il numero di indici di un array; * **ClassCastException:** errore di conversione (casting). |
| *Dichiarare una*  *eccezione* | Similmente al return, usiamo il **throw** (lancia):  *throw new Exception(“Errore!”);*  Non causa solo il ritorno al chiamante, ma a tutti i chiamati, fino a quando qualcuno non cattura l’eccezione. |
| Quando all’interno di un blocco di codice può essere sollevata un’eccezione, questa deve essere dichiarata nella firma del metodo e nei metodi richiamanti.  Il messaggio dell’eccezione sarà visibile insieme all’errore nella console |
| *Catturare una*  *eccezione* | Per catturare un ‘eccezione si usa: il blocco try che la segnala, mentre la cattura avviene con il blocco catch |
| Nella maggior parte dei casi, le eccezioni devono essere gestite nei programmi; in caso contrario  il programma non viene compilato.  Questo è il caso delle eccezioni di tipo **IOException** lanciate dai metodi di I/O, i quali metodi vengono di solito invocati all’interno di clausole try-catch. |
| Interfacce | |
| *Definizione* | Permettono di stabilire lo scheletro di una classe, e servono a dichiarare una classe e i suoi metodi senza definirli del tutto.  In altre parole, servono a fornire una forma, ma non un’implementazione (simili alle classi astratte). |
| *Implementazione* | Possono contenere solo dichiarazioni di metodi, ma anche variabili unicamente **final** o **static**.  I metodi dichiarati all’interno dell’interfaccia sono senza corpo, ma devono averlo nelle classi che implementano l’interfaccia. |
| *Differenza tra classi abstract e interfacce* | La differenza con le classi *abstract* è che quest’ultime risultano più versatili perché oltre a dichiarare variabili astratte, nel suo interno troviamo anche delle definizioni reali; le interfacce però possono essere ereditate da più oggetti. |
| È possibile combinare più interfacce (situazioni dove l’oggetto x è sia di tipo a, b, c non risolvibile con l’ereditarietà dato che Java non ammette quella multipla). |

|  |  |  |
| --- | --- | --- |
| Java I/O | | |
| *Definizione* | Il package java.io definisce i concetti base per gestire l'I/O da qualsiasi sorgente e verso qualsiasi destinazione (sia console che file). Per importare tutto il package dobbiamo scrivere all’inizio della nostra classe: import java.io.\*; | |
| *Stream* | Per ricevere in ingresso dei dati, un programma apre uno stream (flusso di dati) su una sorgente di informazioni (file, memoria, connessione di rete), e ne legge sequenzialmente le informazioni.  C:\Users\Saro\Desktop\Immagine.pngAnalogamente un programma può inviare informazioni ad un destinatario, aprendo uno stream verso di esso, e scrivendo sequenzialmente le informazioni in uscita.  Uno stream è un’astrazione che produce o consuma informazioni, ed è collegato a un device fisico. Tutti gli stream si comportano allo stesso modo, anche se il device fisico a cui sono collegati è diverso. In questo modo, con le stesse classi I/O, i metodi possono essere applicate ad ogni tipo di device. Per esempio, gli stessi metodi possono essere usati per scrivere nella console o un su un file di disco.  Gli Stream rappresentano flussi sequenziali di byte. Tutti gli Stream vengono gestiti con algoritmi del tipo rappresentato nella figura a destra. | |
| *Classi di*  *java.io* | Il package java.io distingue due serie di classi per la gestione degli stream: | |
| **Byte stream** (8 bit - byte)  Parliamo di I/O binario, viene usato in generale per i dati (Es. i bit di un immagine digitale o di un suono digitalizzato).  I flussi di byte sono suddivisi in 2 classi astratte:  **InputStream**: flusso di ingresso  **OutputStream**: flusso d’uscita | **Character stream** (16bit - char)  Flussi di caratteri Unicode a 16bit, parliamo quindi di I/O testuale (Es. i caratteri ascii).  Sono divisi in 2 classi astratte:  **Reader**: lettori  **Writer**: scrittori |
| *Buffer* | In italiano **memoria tampone**, **memoria di transito** o anche **memoria intermediaria**, è una zona di memoria usata temporaneamente per l'input o l'output dei dati, oppure per velocizzare l'esecuzione di alcune operazioni. | Il buffer è utilizzato per la comunicazione fra componenti che lavorano a velocità differenti. Ad esempio se la CPU, che lavora ad alta velocità, deve spedire alcuni dati alla stampante, la quale supporta una velocità molto minore, scriverà tali dati nel buffer di memoria, potendo così continuare a lavorare ad un altro processo mentre la stampante può stampare il dato leggendolo dal buffer e non interrompendo la CPU. |
| Un buffer può essere implementato sia con l'*hardware*, per mezzo di circuiti dedicati, sia con il *software*, riservando una parte della memoria ai dati da manipolare. |
| L'utilizzo di buffer nei software, li espone, se non adeguatamente protetti, ad attacchi che causano *buffer overflow* bloccando il programma o il sistema. |
| La memorizzazione in buffer avviene principalmente attraverso l'utilizzo di code (FIFO) o più raramente di stack (LIFO). Una struttura a coda permette l'ottimizzazione della sequenza di invio dei dati. |
| *Byte stream* | La classe base **InputStream** definisce il concetto  generale di "canale di input" che lavora a byte   * Il **costruttore** apre lo stream * **read()** legge uno o più byte * **close()** chiude lo stream   Attenzione: InputStream è una classe astratta, quindi il metodo read() dovrà essere realmente definito dalle classi derivate | La classe base **OutputStream** definisce il concetto  generale di "canale di output" che opera a byte  il costruttore apre lo stream (vedi open)   * **write()** scrive uno o più byte * **flush()** svuota il buffer di uscita * **close()** chiude lo stream   Attenzione: OutputStream è una classe astratta,  quindi il metodo write() dovrà essere realmente definito dalle classi derivate |
| Dalle classi base astratte si derivano varie classi concrete, specializzate per fungere da:   * sorgenti per input da file * dispositivi di output su file * stream di incapsulamento, cioè pensati per aggiungere nuove funzionalità a un altro stream (I/O bufferizzato, filtrato,… I/O di numeri, di oggetti,...) | C:\Users\Saro\Desktop\Immagine.png |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Byte stream*  *Input da file* | **FileInputStream** è la classe derivata che rappresenta il concetto di sorgente di byte agganciata a un file   * il nome del file da aprire è passato come parametro al costruttore di FileInputStream * in alternativa si può passare al costruttore un oggetto File costruito in precedenza | | | | | |
| Per aprire un file binario in lettura si crea un **oggetto di classe FileInputStream**, specificando il nome del file all'atto della creazione nell'argomento (apertura invocata in modo implicito) | | | Per leggere dal file si usa poi il **metodo read()** che permette di leggere uno o più byte   * restituisce il byte letto come intero fra 0 e 255 * se lo stream è finito, restituisce -1 * se non ci sono byte, ma lo stream non è finito, rimane in attesa dell'arrivo di un byte | | |
| Poiché le operazioni su stream possono fallire per varie cause, tutte le operazioni possono sollevare eccezioni necessità di try / catch | | | | | |
| *Byte stream*  *Output su file* | **FileOutputStream** è la classe derivata che rappresenta il concetto di dispositivo di uscita agganciato da un file   * il nome del file da aprire è passato come parametro al costruttore di FileOutputStream * in alternativa si può passare al costruttore un oggetto File costruito in precedenza | | | | | |
| Per aprire un file binario in scrittura si crea un  **oggetto di classe FileOutputStream**, specificando  il nome del file all'atto della creazione, e un secondo parametro opzionale, di tipo boolean, che  permette di chiedere l'apertura in modo append | | | | Per scrivere sul file si usa il **metodo write()**  che permette di scrivere uno o più byte  – scrive l'intero (0 255) passatogli come parametro  – non restituisce nulla | |
| *Stream di*  *incapsulamento* | Gli stream di incapsulamento hanno come scopo quello di avvolgere un altro stream per creare un’entità con funzionalità più evolute. | | C:\Users\Saro\Desktop\Immagine.png | | | Il loro costruttore ha quindi come parametro un InputStream o un OutputStream esistente |
| *Stream di incapsulamento*  *input* | BufferedInputStream | aggiunge un buffer e ridefinisce read() in modo da avere una lettura bufferizzata | | | | |
| DataInputStream | definisce metodi per leggere i tipi di dati standard in forma binaria: readInteger(), readFloat(), ... | | | | |
| ObjectInputStream | definisce un metodo per leggere oggetti "serializzati" (salvati) da uno stream; offre anche metodi per leggere i tipi primitivi e gli oggetti delle classi wrapper (Integer, etc.) di Java | | | | |
| *Stream di incapsulamento*  *output* | BufferedOutputStream | aggiunge un buffer e ridefinisce write() in modo da avere una scrittura bufferizzata | | | | |
| DataOutputStream | definisce metodi per scrivere i tipi di dati standard in forma binaria: writeInteger() | | | | |
| PrintStream | definisce metodi per stampare come stringa valori primitivi (con print()) e classi standard (con toString()) | | | | |
| ObjectOutputStream | definisce un metodo per scrivere oggetti "serializzati"; offre anche metodi per scrivere i tipi primitivi e gli oggetti delle classi wrapper (Integer, etc.) di Java | | | | |
| *Character Stream* | Le classi per l’I/O da stream di caratteri (Reader e Writer) sono più efficienti di quelle a byte, hanno nomi analoghi e struttura analoga, e convertono correttamente la codifica UNICODE di Java in quella locale. | | | C:\Users\Saro\Desktop\Immagine.png | | |
| Rispetto agli stream binari, il file di testo si apre costruendo un oggetto FileReader o FileWriter, rispettivamente read() e write() leggono/scrivono un int che rappresenta un carattere UNICODE.  Un carattere UNICODE è lungo due byte, read() restituisce -1 in caso di fine stream.  Occorre dunque un cast esplicito per convertire  il carattere UNICODE in int e viceversa. | | |
| *InputSteramReader*  *e*  *OutputStream-*  *Reader* | Gli stream di byte esistono da Java 1.0, quelli di caratteri esistono invece da Java 1.1. Varie classi esistenti fin da Java 1.0 usano quindi stream di byte anche quando dovrebbero usare in realtà stream di caratteri. La conseguenza è che i caratteri rischiano di non essere sempre trattati in modo coerente. | | | | | |
| Occorre dunque poter **reinterpretare** uno stream di byte come reader / writer quando opportuno.  Esistono due classi "incapsulanti" progettate proprio per questo scopo:   * *InputStreamReader* che reinterpreta un InputStream come un Reader; * *OutputStreamWriter* che reinterpreta un OutputStream come un Writer. | | | | | |
| *La classe System* | Video e tastiera sono rappresentati dai due oggetti statici *System.in* e *System.out*. Poiché esistono fin da Java 1.0 (quando Reader e Writer non esistevano), essi sono formalmente degli stream di byte, ma in realtà sono stream di caratteri. | | | | | |
| System.in può essere interpretato come un Reader incapsulandolo dentro a un InputStreamReader  *InputStreamReader tastiera = new InputStreamReader(System.in);* | | | | | |
| System.out può essere interpretato come un Writer incapsulandolo dentro a un OutputStreamWriter  *OutputStreamWriter video = new OutputStreamWriter(System.out);* | | | | | |
| L’oggetto reader legge singoli caratteri. La classe **BufferedReader** trasforma un reader in un lettore in grado di leggere intere righe. Il metodo **readLine** della classe BufferedReader consente di leggere una singola riga di testo da tastiera   * *InputStreamReader reader = new InputStreamReader(System.in);* * *BufferedReader console = new BufferedReader(reader);* * *System.out.println(“Inserisci una riga di testo”);* * *String str = console.readLine();* | | | | | |
| *La classe*  *ConsoleReader()* | La classe ConsoleReader() racchiude tutti i termini e i metodi della classe System per quanto riguarda l’input e l’output di dati. | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Analisi di Complessità | | | |
| *Problema* | Funzione tra l’insieme delle istanze del problema e l’insieme delle soluzioni  *Es*. ; | | |
| *Algoritmo* | Procedimento finito (descrivibile finitamente), non ambiguo (ad ogni istante è possibile determinare ciò che l’algoritmo sta eseguendo) e terminante (avente sempre fine) | | |
| *Valutazione della complessità di un algoritmo* | La complessità di un algoritmo non si valuta dal tempo di esecuzione di un programma che lo esegue (esso dipenderebbe dal tipo di implementazione, tipo di input o hardware utilizzato).  Il tempo di esecuzione complessivo di un algoritmo dipende dalla **dimensione dell’input**, ossia dalla quantità di memoria necessaria a rappresentare l’input. | | |
| *Funzione di complessità tempo* | Lega la dimensione dell’input al tempo che l’algoritmo impiega su di esso.  Dato un algoritmo , tipicamente indica la sua funzione di complessità. Se non ci sono ambiguità, la funzione di complessità si indica semplicemente con T.  *Es*. Data una costante e , consideriamo due input di dimensione e | | |
| *Approssimazione della funzione di complessità* | * n piccolo: prevale l’ultimo termine * n = 10: secondo e ultimo termine uguali * n = 100: primo e secondo termine uguali * n > 100: è il primo termine a prevalere   A seconda dei valori di n, scegliamo la migliore approssimazione per T(n). | | |
| *Notazione*  *O-grande* | se due costanti tali che . ( = dimensione input)   * g(n) limita superiormente T(n) * g(n) approssima asintoticamente T(n) dall’alto * g(n) è una buona approssimazione superiore per T(n) quando n è molto grande   *Es*. Il limite superiore per la ricerca in un array non ordinato è  *Es*. Provare che . Dobbiamo dimostrare che le due costanti positive c ed  N tali che . Basta scegliere e , infatti  *Es*. è vera o falsa? Chiaramente è falsa: n non può essere un limite superiore per | | |
| *Notazione* | se due costanti tali che ( omega)   * g(n) limita inferiormente T(n) * g(n) approssima asintoticamente T(n) dal basso * g(n) è una buona approssimazione inferiore per T(n) quando n è molto grande   *Es*. Il limite inferiore per la ricerca in un array non ordinato è  *Es*. Provare che . Dobbiamo dimostrare che le due costanti positive c ed  N tali che . Basta scegliere e , infatti  *Es*. è vera o falsa? Chiaramente è falsa: non può essere un limite inferiore per | | |
| *Notazione* | è se tre costanti ed N tali che ( theta)   * g(n) limita strettamente T(n) * g(n) approssima asintoticamente T(n) * g(n) è una buona approssimazione per T(n) quando n è molto grande   *Teorema*.  *Es.* Abbiamo dimostrato che .  Questo basta per concludere che | | |
| *Proprietà di* | Un polinomio con termine di grado massimo positivo si comporta asintoticamente come il monomio con potenza massima e coefficiente unitario. | | |
| *Complessità tipiche* | C:\Users\Saro\Desktop\Immagine.png | | |
| *Altre Proprietà* | * Se e , allora * Se e , allora (transitività) | | |
| *Complessità di un programma* | È la complessità dell’algoritmo implementato dal programma.  *Es.* Quanti enunciati di **assegnamento** fa l’algoritmo?  2 per l’inizializzazione, 2 per ciascuna iterazione del ciclo (i e sum;  n iterazioni)  Totale 2+2n assegnamenti  for (i = sum = 0; i<n; i++)  sum += a[i];  Ogni assegnamento prende tempo costante:  *Es*. 1 per l’inizializzazione  3 per ciascuna iterazione del ciclo esterno (i, j e sum; n iterazioni)  2 per ciascuna iterazioni del ciclo interno (i iterazioni  for (i = 0; i<n; i++)  for (j=1, sum=a[0]; j<=1;j++)  sum += a[j]; | | |
| *Complessità*  *vari casi* | Caso peggiore | l’algoritmo richiede tempo massimo | **complessità costante**  **complessità logaritmica**  **complessità lineare**  **complessità pseudolineare**  **complessità quadratica**  **complessità cubica**  **complessità polinomiale**  **complessità esponenziale**  *Es.*  🡪 complessità lineare  🡪 complessità quadratica  🡪 complessità polinomiale  🡪 complessità esponenziale  È molto utile ricordare i diversi ordini di infinito:  Questo significa che un algoritmo di complessità logaritmica è più efficiente di uno di complessità . |
| Caso medio | l’algoritmo richiede tempo medio |
| Caso migliore | l’algoritmo richiede tempo minimo |
| *Es.* Algoritmo A per ordinare in senso crescente un array di 5 interi  Nel **caso peggiore** l’array è in ordine decrescente, ed A impiega passi(input), nel **caso migliore** la complessità è costante  **Caso medio**: #medio passi di A= | |

|  |  |  |  |
| --- | --- | --- | --- |
| Strutture dati | | | |
| *Definizione* | In informatica una struttura dati è un'entità usata per **organizzare un insieme di dati** all'interno della memoria del computer.  La scelta delle strutture dati da utilizzare è strettamente legata a quella degli algoritmi; per questo, spesso essi vengono considerati insieme. Infatti, la scelta della struttura dati influisce inevitabilmente sull'efficienza degli algoritmi che la manipolano, a prescinde dai dati effettivamente contenuti.  Le strutture di dati si differenziano prima di tutto in base alle **operazioni** che si possono effettuare su di esse e alle prestazioni offerte. | | |
| *Strutture dati fisse* | **Array** | struttura dati omogenea, che contiene un numero **finito** di elementi dello stesso tipo. Questi elementi sono individuati attraverso un indice numerico, che tipicamente va da 0 al numero massimo di elementi meno uno. La dimensione del vettore deve essere dichiarata al momento della sua creazione. | |
| **Record** | struttura dati che può essere eterogenea o omogenea. Nel primo caso contiene una combinazione di elementi che possono essere di **diverso** tipo, ad esempio un intero, un numero in virgola mobile e un carattere testuale. Gli elementi che lo compongono sono detti anche campi, e sono identificati da un nome. | |
| **Classe** | costrutto tipico dei linguaggi orientati agli **oggetti**, e consiste in un record a cui sono associate anche delle operazioni o metodi. | |
| *Strutture dati dinamiche* | Le strutture dati dinamiche sono basate sull'uso di dati con i loro **riferimenti in** **memoria**, e sull'allocazione dinamica della memoria. Gli elementi possono essere **allocati** (e **deallocati**) man mano che servono, collegati tra loro in modi diversi, e questi collegamenti possono a loro volta mutare durante l'esecuzione del programma. Lo spazio di memoria necessario per allocare i puntatori, e le operazioni necessarie alla loro manutenzione costituiscono il costo aggiuntivo delle strutture dati dinamiche. | | |
| **Lista concatenata** | | insieme di "**nodi**" collegati linearmente. I nodi sono dei record che contengono un "carico utile" (**valore**) di dati, ed un puntatore all'elemento successivo della lista. Un nodo funge da testa della lista, e da questo è possibile accedere a tutti i nodi della lista. Il costo di accesso ad un nodo della lista cresce con la dimensione della lista. Conoscendo il nodo precedente ad un nodo N, è possibile rimuovere N dalla lista, o inserire un elemento prima di lui, in un tempo costante |
| **Lista doppiamente concatenate** | | in questo caso i nodi contengono un puntatore sia al nodo precedente che al successivo. Dato un nodo N il suo successore è N->succ, e il suo precedente è  N->prec. Deve sempre essere vero che N->succ->prec == N. |
| **Albero** | | ogni nodo contiene due (o più) riferimenti ad altri nodi che sono detti suoi "**figli**".  Ciascun nodo deve essere figlio di un solo padre. In molte implementazioni, ogni nodo ha un numero fissato di figli, ad esempio due o tre. Si parla in questo caso di **alberi binari o ternari**.  Ciascun nodo, oltre ai puntatori ai nodi figli, ha normalmente un "carico utile" (**valore**), ovvero un dato associato al nodo, utile per il problema applicativo da risolvere. |
| *Contenitori* | Le strutture dati sopra esposte possono essere utilizzate per realizzare alcuni tipi di contenitori di utilizzo frequente, che possono forzare una particolare modalità di accesso ai dati. | | |
| **Pila (o stack)** | | struttura dati di tipo **LIFO** (Last In First Out) con metodi **push** e **pull**.  Viene tipicamente realizzata con array o liste |
| **Coda (o queque)** | | struttura dati di tipo **FIFO** (First In First Out) con metodi **enqueque** o **dequeque**.  Viene tipicamente realizzata con array o liste. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liste Concatenate | | | | |
| *Vantaggi di una lista concatenata* | * gli elementi sono organizzati in un’unica struttura; * la dimensione del nostro insieme cresce e decresce in funzione delle reali necessità del processo in esecuzione. * è possibile mantenere un ordine nell’insieme; * è possibile inserire elementi nel centro della lista. | | | |
| *Struttura di una lista concatenata* | Una lista concatenata semplice è caratterizzata dal fatto che gli elementi vengono aggiunti **dinamicamente** solo quando è necessario.  Inoltre ogni elemento contiene un **riferimento** all’elemento successivo.  Serve anche un **identificatore** esterno (*aux*) per tener traccia della lista. | | | |
| C:\Users\Saro\Desktop\Immagine.png | | | |
| *Implementazione* | Per creare un nuovo elemento con valore 10 scriviamo:  Nodo elem = new Nodo(10);  Per creare un secondo elemento con valore 13:  elem.setNext(new Nodo(13));  Per creare un terzo elemento con valore 45:  (elem.getNext()).setNext( new Nodo(45)); | | | |
| Per ogni nuovo nodo creato dobbiamo comunque istanziare un nuovo oggetto della classe Nodo e dobbiamo scrivere il **riferimento** al nuovo nodo nel nodo precedente | | | |
| *Nodo aux* | Per accedere ad un qualunque nodo dobbiamo accedere prima al nodo precedente. Se ad esempio vogliamo settare il quarto nodo, per evitare di dover scrivere codice del genere:  ((head.getNext()).getNext).setNext(new Nodo(99));  utilizziamo un nodo che chiamiamo *aux* il quale verrà assegnato al nuovo nodo creato per gestire le sue informazioni. | | | Nodo head = new Nodo(13);  Nodo aux = head;  aux.setNext(new Nodo(16));  aux = aux.getNext();  aux.setNext(new Nodo(18)); |
| *Inserimento*  *in testa* | I passi da seguire per l’inserimento in testa sono:   * istanziare un nuovo elemento; * collegare la lista già esistente al nuovo nodo; * modificare il riferimento alla testa. | | C:\Users\Saro\Desktop\Immagine.png | |
| *Inserimento*  *in coda* | I passi da eseguire per l’inserimento in coda sono:   * scorrere la lista fino all’ultimo elemento; * istanziare un nuovo elemento * collegare l’ultimo elemento trovato con il nuovo elemento appena creato.   Se la lista è vuota bisogna però creare il primo  elemento (modificare il valore di head) | | C:\Users\Saro\Desktop\Immagine.png | |
| *Inserimento*  *ordinato* | L’inserimento ordinato comporta l’aggiunta di un elemento in una qualunque posizione all’interno della lista. Le operazioni da eseguire sono:   * scorrere la lista confrontando gli elementi già presenti con quello da inserire; * arrestarsi appena si trova un elemento maggiore (o minore) * collegare la parte con gli elementi minori al nuovo elemento * collegare la parte con gli elementi maggiori | | C:\Users\Saro\Desktop\Immagine.png | |
| *Ricerca di un*  *elemento* | Per la ricerca, dobbiamo distinguere due casi:   * la lista è ordinata; * la lista non è ordinata.   Nel primo caso possiamo arrestare la ricerca quando troviamo due elementi consecutivi rispettivamente maggiore e minore dell’elemento da cercare.  Nel secondo caso, se non troviamo prima l’elemento desiderato, dobbiamo arrivare fino al termine della lista. | | | |
| *Cancellazione* | Per la cancellazione abbiamo:   * cancellazione del primo elemento * cancellazione dell’ultimo elemento * cancellazione di un elemento specificato.   Nel terzo caso dobbiamo prima cercare l’elemento da cancellare. | | | |
| Liste doppiamente concatenate | | | | |
| *Differenze* | C:\Users\Saro\Desktop\Immagine.pngLe liste concatenate semplici consentono di scorrere gli elementi soltanto in una direzione, a partire dalla testa.  Per alcune applicazioni è utile poter scorrere la lista in entrambe le direzioni.  A tale scopo è possibile definire una lista con due riferimenti, uno all’oggetto precedente e l’altro al successivo. | | | |
| *Inserimento*  *in testa e*  *in coda* | I passi da seguire per l’inserimento in testa o in coda in una lista doppiamente concatenata sono:   * istanziamento di un nuovo elemento; * se la lista è vuota modificare la testa e la coda; * altrimenti si setta il riferimento al nodo successivo della nuova testa alla vecchia testa e alla vecchia testa il riferimento del precedente alla nuova testa * modifica del riferimento alla nuova testa. | C:\Users\Saro\Desktop\Immagine.pngC:\Users\Saro\Desktop\Immagine.png | | |

|  |  |  |
| --- | --- | --- |
| *Inserimento*  *ordinato* | Nel caso di inserimento ordinato bisogna distinguere vari casi:   * la lista è vuota * l’elemento da inserire è il primo * l’elemento da inserire è l’ultimo * l’elemento va inserito nel mezzo | C:\Users\Saro\Desktop\Immagine.png |
| *Ricerca di un*  *elemento* | Anche nel caso di liste doppiamente concatenate, nella ricerca, dobbiamo distinguere due casi:   * la lista è ordinata; * la lista non è ordinata.   La presenza del doppio riferimento non dà nessun vantaggio in termini di tempo di ricerca. | |
| *Cancellazione* | Per la cancellazione abbiamo:   * cancellazione del primo elemento * cancellazione dell’ultimo elemento * cancellazione di un elemento specifico   Il primo ed il secondo caso sono perfettamente  duali.  Nel terzo caso dobbiamo prima cercare l’elemento  da cancellare.  In tutti i casi dobbiamo verificare se è necessario  aggiornare le variabili head e tail. | Cancellazione di un elemento specifico  C:\Users\Saro\Desktop\Immagine.png  C:\Users\Saro\Desktop\Immagine.png |
| *Liste circolari* | Le liste circolari sono un’estensione delle liste tradizionali (semplici o doppiamente concatenate).  La loro caratteristica è data dalla presenza di un riferimento tra l’ultimo elemento ed il primo Nodo.  C:\Users\Saro\Desktop\nodo.png | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pila | | | | |
| *Definizione* | C:\Users\Saro\Desktop\Immagine.pngIn alcuni casi è utile disporre di strutture dati che hanno un solo punto di accesso per inserire e reperire i dati (ad esempio, una pila di libri). In una struttura di questo tipo i dati (i libri) vengono inseriti solo in cima e possono essere estratti solo dalla cima.  Un altro esempio può essere le persone che entrano in un ascensore.  Le strutture di questo tipo prendono il nome di Pile (Stack), le quali sono dei sistemi **LIFO** (Last In First Out). Esse hanno due metodi principali: *push* e *pop*. | | | |
| *Operazioni possibili* | * Verificare se è piena (**IsFull**) * Verificare se è vuota (**IsEmpty**) * Inserire un elemento (**Push**) * Togliere un elemento (**Pop**) * Far restituire il primo elemento, senza estrarlo (**TopElem**) * Cancellare tutti i dati (**Clear**) | In alcuni casi le strutture LIFO hanno una dimensione limitata, per cui è necessario definire un valore massimo di elementi inseribili. | | |
| *Implementazione* | Per implementare una pila servono:   * uno **spazio di memoria** ordinato, dove inserire gli elementi (come un array) * un **indice**, per sapere qual è l’ultimo elemento inserito (*top*)   C:\Users\Saro\Desktop\Immagine.png  L’indice deve tener conto di quanti elementi ci sono nella pila. Normalmente si utilizza un *array* per memorizzare gli elementi, e un numero *intero* che indica la prima posizione libera dello stack. | | | **top = 1**  Vuol dire che c’è un elemento nell’array in posizione 0 |
| *Push*  *(inserimento)* | Nella procedura di inserimento bisogna eseguire i seguenti passi:   * verificare che la pila non sia piena; * inserire l’elemento appena passato; * spostare di una posizione in alto l’indice top. | | | |
| *Pop*  *(estrazione)* | Nella procedura di estrazione bisogna eseguire i seguenti passi:   * verificare che la pila non sia vuota; * decrementare il valore dell’indice; * leggere l’oggetto che sta in cima alla pila. | | | |
| Coda | | | | |
| *Definizione* | Al contrario delle pile, le code (o *QUEUE*, si legge “chiù”) usano il sistema **FIFO** (First In First Out). La coda è spesso usata in molte situazioni della vita quotidiana (fila alla posta, prenotazione delle pizze, ecc.), ed è una sequenza di elementi che può essere “accorciata” da un lato e allungata da un altro lato tramite le seguenti due operazioni:   * **Enqueue**: operazione che corrisponde all’inserimento di un elemento in coda * **Dequeue**: operazione che corrisponde all’estrazione dell’elemento testa dalla coda   C:\Users\Saro\Desktop\Immagine.png | | | |
| *Operazioni*  *possibili* | * inserire un elemento x in coda (**EnQueue**(x)); * togliere un elemento dalla coda (**DeQueue**()); * verificare se la coda è vuota (**IsEmpty**()); * cancellare tutti i dati (**ClearQueue**()); * leggere (senza toglierlo dalla coda) il primo elemento in attesa (**readHead**()); * nel caso in cui si preveda una coda con capacità massima limitata, verificare se la coda ha raggiunto la sua massima capacità: (**IsFull**()). | | Normalmente anche le code hanno una dimensione  limitata, oltre la quale non vengono più accettati  inserimenti. | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Schema*  *di una coda* | Per gestire una sequenza di dati il primo supporto da adoperare che viene in mente sono gli array. Essi purtroppo presentano numerosi problemi nel momento in cui debbono gestire le operazioni tipiche di una coda.  Possiamo però utilizzare un array di dimensioni fisse che fornirà lo spazio di memoria dove vengono messi in sequenza gli elementi della coda.   * L’**estrazion**e dall’inizio della coda prevede che si estragga l’elemento di indice 0 e che tutti gli altri elementi successivi vengano “scivolati” avanti di un indice. * C:\Users\Saro\Desktop\Immagine.pngL’**inserimento** avviene semplicemente inserendo un elemento nel primo indice libero in fondo all’array. Dovrò pertanto mantenere in una variabile il valore di questo indice e aggiornarlo sia all’inserimento che alla estrazione. | | | | |
| *Operazioni*  *Modulari* | Sia A l’array di supporto e sia **DIM=A.length**;  Una operazione di Dequeue aggiorna l’indice di testa come segue:  *indiceTesta=(indiceTesta+1)%DIM;*  Una operazione di Enqueue aggiorna l’indice di coda come segue:  *indiceCoda=(indiceCoda+1)%DIM* | | Es. Se nella coda abbiamo 2|3| | |  con tail=2 e head=0  Enque di 6  tail = 3%4=3 2|3|6| |  Deque del 2  head = 1%4=1 3|6| | | | | |
| *Code circolari* | Eseguire uno shift di tutti gli elementi dopo ogni estrazione è troppo oneroso. Per tale motivo è stata pensata una struttura, denominata *coda circolare*, nella quale esistono due indici, **head** e **tail**, che indicano il primo elemento e l’ultimo. | | | | |
| C:\Users\Saro\Desktop\Immagine.png | C:\Users\Saro\Desktop\Immagine.png  Dequeue | | C:\Users\Saro\Desktop\Immagine.png  Enqueue | |
| C:\Users\Saro\Desktop\Immagine.pngIl vantaggio di una simile struttura logica è che non è necessario effettuare shift per ogni inserimento, ma basta una sola assegnazione (più la modifica della variabile head).  Ogni operazione di Enqueue o Dequeue comporta l’avanzamento di uno degli indici (tail per Enqueue, head per Dequeue).  Gli indici head e tail vengono sempre incrementati, rispettivamente con le operazioni di DeQueue e EnQueue. | | | | |
| *Limiti della*  *coda circolare* | Chiaramente il valore dell’indice tail potrà raggiungere ma non superare il valore dell’indice head (a seguito di operazioni di Enqueue, riempimento  della coda) come nella figura a destra. | | | | C:\Users\Saro\Desktop\Immagine.png |
| Analogamente head non potrà superare tail (dopo operazioni di Dequeue, svuotamento della coda ) come nella figura a destra.  *La figura a destra è un errore!* | | | | C:\Users\Saro\Desktop\Immagine.png |
| Se però i due puntatori coincidono, dobbiamo poter distinguere le condizioni  di coda vuota (prima figura) o coda con un solo elemento (seconda figura). | | | | C:\Users\Saro\Desktop\Immagine.png |
| *Implementazione* | Riassumendo, per definire una coda servono:   * uno **spazio di memoria ordinato**, dove inserire gli elementi * **due indici**, per sapere quali sono il primo e l’ultimo elemento. | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ricorsione | | | | |
| *Definizione* | La funziona fattoriale, dato un numero intero non negativo n, è così definita:  Il quale significa:  0!=1  1!=1(1-1)!=10!=1  2!=2(2-1)!=21!=2=2  3!=3(3-1)!=3!=321=6  4!=4(4-1)!=43!=4321=24  …  Quindi per ogni n intero positivo, il fattoriale di n è il prodotto dei primi n numeri interi positivi. | Il fattoriale è un esempio di funzione con ricorsione; essa si basa si basa sul **principio di induzione**: per dimostrare una preposizione si dimostra il *caso base*, e poi bisogna dimostrare se la proprietà vale per n, e se vale anche per n+1 essa vale per qualsiasi n. Una funzione si dice ricorsiva se è definita in termini di se stessa.  Ogni definizione ricorsiva è caratterizzata dal:   * **Caso base (condizione di terminazione):** la condizione per cui la funzione termina, cioè smette di richiamare se stessa. Se non ci fosse, la funzione sarebbe un loop infinito. * **Passo ricorsivo (chiamata ricorsiva):** la soluzione ottenuta viene combinata con altra informazione per produrre la soluzione al problema originale. | | |
| *Ricorsione in Java* | In Java la ricorsione è come un ciclo che richiama se stesso più volte fino al caso base, per poi ritornare qualcosa dall’ultima chiamata alla penultima, alla terz’ultima, ecc. Abbiamo 2 possibilità per scrivere il fattoriale in java: utilizzando un metodo iterativo o utilizzando un metodo ricorsivo. | | Invocare un metodo mentre si esegue il metodo stesso è un paradigma di programmazione che si chiama *ricorsione,* ed il metodo che ne fa uso si chiama *metodo ricorsivo*. La ricorsione è uno strumento molto potente per realizzare alcuni algoritmi, ma può essere anche causa di molti errori di difficile diagnosi. | |
| *Pila di esecuzione* | Per capire come utilizzare correttamente la ricorsione, vediamo innanzitutto come funziona. In generale, per gestire la chiamata di metodi all’interno del corpo di altri metodi, la macchina virtuale Java fa uso di una **pila di esecuzione** (run-time stack). | | | |  | | --- | | Parametri e variabili locali | | Connessione dinamica | | Indirizzo di ritorno | | Valore restituito | |
| Gli elementi della pila sono record della forma dello schema a destra, e sono detti *Documenti di attivazione* (o **Activation Record**): | | |
| C:\Users\Saro\Desktop\Immagine.png | | Quando un metodo ricorsivo invoca se stesso, la  macchina virtuale Java esegue le stesse azioni che  vengono eseguite quando viene invocato un  metodo qualsiasi  1. sospende l’ esecuzione del metodo invocante  2. esegue il metodo invocato fino alla sua  terminazione  3. riprende l’ esecuzione del metodo invocante  dal punto in cui era stata sospesa (con un eventuale valore di ritorno). | |
| Vediamo la sequenza usata per calcolare 3!  si invoca factorial(3)  factorial(3) invoca factorial(2)  factorial(2) invoca factorial(1)  factorial(1) invoca factorial(0)  factorial(0) restituisce 1  factorial(1) restituisce 1  factorial(2) restituisce 2  factorial(3) restituisce 6  Come vediamo il metodo invoca se stesso ma con numeri più piccoli di quello preso in input, fino ad arrivare al caso base, dopo il quale comincerà la cascata dei return.  Si crea quindi una lista di metodi in attesa, che si allunga e poi si accorcia fino ad estinguersi. | | Esistono due regole ben definite che vanno utilizzate per scrivere metodi ricorsivi:  1. Caso base: il metodo ricorsivo deve fornire la soluzione del problema in almeno un caso particolare, senza ricorrere ad una chiamata ricorsiva  2. Passo induttivo: il metodo ricorsivo deve effettuare la chiamata ricorsiva semplificando il  problema (ossia avvicinandosi al caso base) | |
| Si potrebbe pensare che le chiamate ricorsive si  possano succedere una dopo l’altra, all’infinito.  Invece ad ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base; la soluzione del caso base non richiede ricorsione.  Quindi la soluzione ricorsiva di un problema è un  *algoritmo effettivo* in quanto arriva a conclusione  in un numero **finito** di passi. | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Pila di esecuzione*  *al dettaglio* | * Consideriamo l’istruzione int n=fact(3); * fact cerca di restituire 3\*fact(2), ma fact(2) deve essere calcolato; * fact(2) manda di nuovo in esecuzione la funzione con l’argomento 2, fact(2) cerca di restituire il valore 2\*fact(1), ma fact(1) deve essere calcolato, * fact(1) manda di nuovo in esecuzione la funzione con l’argomento 1, fact(1) cerca di restituire il valore 1\*fact(0), ma fact(0) deve essere calcolato, * fact(0) manda di nuovo in esecuzione la funzione con l’argomento 0, finalmente siamo arrivati al caso base, fact(0) può essere calcolato e vale 1 | | * A questo punto il calcolo 1\*fact(0) può essere completato 1\*1=1 * Ora Il calcolo 2\*fact(1) può essere completato, restituendo 2 a fact(2) * Ora il calcolo 3\*fact(2) può essere completato, restituendo 6 a fact(3) | |
| *Ricorsione infinita* | Se manca il caso base, o ad ogni passo ricorsivo la soluzione non si semplifica, il metodo continua a richiamare se stesso all’infinito dando lungo ad una ricorsione infinita.  Solitamente il programma termina con l’errore **StackOverflowError** generato perché si è esaurita la memoria disponibile per tenere traccia delle chiamate. | | | |
| *Ricorsione in coda* | Esistono diversi tipi di ricorsione, quello visto nel caso del metodo “factorial” si chiama *ricorsione di coda* (o tail recursion).  Nella ricorsione in coda il metodo ricorsivo esegue una sola invocazione ricorsiva (ossia se stesso), e tale invocazione è l’ultima azione del metodo. | Allora, a cosa serve la ricorsione in coda?   * Non è necessaria ma rende il codice più leggibile * E’ utile quando la soluzione del problema è esplicitamente ricorsiva (es. fattoriale) * In ogni caso, la ricorsione in coda è meno efficiente del ciclo equivalente perché il sistema deve gestire le invocazioni sospese | | |
| *Esempio* | scrivere in Java un metodo per calcolare la funzione esponenziale: | | | |
| *Ricorsione*  *non in coda* | È relativa a metodi ricorsivi in cui la chiamata al metodo stesso non è l’ultima azione compiuta.  La ricorsione non in coda non può essere eliminata facilmente, però è possibile dimostrare che essa è sempre eliminabile. | | | |
| *Ricorsione*  *multipla* | Si parla di ricorsione multipla quando un metodo richiama se stesso *più volte* durante la sua esecuzione. Un esempio di ricorsione multipla è la **funzione di Fibonacci**  Tale funziona genera i numeri 0,1,1,2,3,5,8,13,21,34,55,89… (i primi due numeri sono 0 ed 1, mentre gli altri sono la somma dei due predecessori nella sequenza prodotta). | | | La ricorsione multipla va usata con estrema cautela perché può portare a programmi molto inefficienti.  Eseguendo il calcolo della funzione di Fibonacci per un intero relativamente grande si può osservare che il tempo di elaborazione cresce molto rapidamente. |
| *Metodo di lavoro* | Per risolvere un algoritmo con la ricorsione, non bisogna esprimere la soluzione in termini di se stessa, ma su di un problema più piccolo.  Se il problema diventa sempre più piccolo, alla fine sarà così piccolo che potrà essere risolto direttamente. | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Alberi | | | |
| *Motivazioni* | Tutte le strutture dati viste fino ad ora sono strutture dati *lineari*, con le quali è difficile rappresentare una gerarchia di dati. Una rappresentazione gerarchica dei dati è invece possibile da realizzare con una struttura dati chiamata *albero*.  Essa è una figura inspirata all’omonimo concetto esistente in natura, solo che in informatica tipicamente sono disegnati a testa in giù. | C:\Users\Saro\Desktop\Immagine.png | |
| Lo scopo di una rappresentazione gerarchica dei dati è comprendibile a livelli di efficienza: le operazioni di ricerca, inserimento e cancellazione dei dati sono enormemente più veloci rispetto alla rappresentazione lineare.  Es. Consideriamo l’insieme , con  La rappresentazione tramite lista concatenata è la seguente:  C:\Users\Saro\Desktop\Immagine.png  Con la struttura gerarchia ad albero invece è:  C:\Users\Saro\Desktop\Immagine.png  In termini di complessità possiamo notare che, se ad esempio dobbiamo ricercare il 31, la ricerca lineare dovrà scorrere valore per valore per trovare il numero, mentre nella ricerca gerarchia possiamo già escludere le gerarchie dei due estremi e ricercare il valore solo nel centro. | | |
| *Definizioni* | Un albero è un insieme (anche vuoto) di **nodi** (o archi) connessi medianti **rami** (o vertici). Ogni nodo (eccetto il nodo **radice**) è connesso tramite un ramo a un altro nodo che ne è il **padre**, e di cui rappresenta un **figlio**. Quindi la radice è l’unico nodo privo di padre. | | |
| Se sono alberi (non vuoti) privi di nodi comuni, e X è un nodo, allora inserendo tutti come sotto-alberi di X, si ottiene un unico albero. | | |
| Ogni nodo può contenere informazioni di vario tipo, rappresentate dalla sua **etichetta** (o chiave). Spesso si indica un nodo mediante la sua etichetta. | | |
| *Cammino* | Un cammino è una sequenza di nodi e rami tali che ciascun nodo (tranne l’ultimo) sia padre del successivo. Esso è individuato dalla sequenza delle etichette dei suoi nodi.  C:\Users\Saro\Desktop\Immagine.png  Es. In questo albero possiamo trovare i cammini (A,B,E), (A,C,F),(A,C,G),(A,D). | | La **lunghezza** di un cammino è il numero di nodi coinvolti in un cammino. Ogni nodo costituisce un cammino di lunghezza zero.  Es. Nell’albero accanto, il cammino (C,F) ha lunghezza 1, mentre il cammino (A,B,E) ha lunghezza 2. In generale, un cammino con K nodi ha lunghezza K-1. |
| Il **cammino** **caratteristico** è un cammino che esiste per ciascun nodo, il quale lo collega alla radice.  Es. Nell’albero accanto, (A,B,E) è il cammino caratteristico del nodo E, mentre (A,C,G) è il cammino caratteristico del nodo G. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Livelli e tipi*  *di nodi* | Il **livello di un nodo** è la lunghezza del camino caratteristico di un nodo.  Es. Nell’albero sopra, il nodo A ha livello 0; il nodo C ha livello 1; il nodo E ha livello 2. | | * Un nodo senza figli si chiama **foglia**; * Un nodo con almeno un figlio si chiama **nodo interno**; * Il nodo interno senza padre si chiama **radice**; * Due o più nodi con lo stesso padre si chiamano **fratelli**.   Es. Nell’albero sopra abbiamo:   * + D,F,G,E sono delle foglie   + A,B,C sono nodi interni   + A è la radice   + B,C,D sono fratelli di padre A; e F,G sono fratelli di padre C | | |
| Se il cammino caratteristico di un nodo Y contiene un nodo X, diciamo che X è **antenato** di Y e Y è **discendente** di X.  Es. Nell’albero sopra, per trovare gli antenati di F dobbiamo prima trovare il suo cammino caratteristico che è (A,C,F), e poi possiamo dunque dichiarare A e C come antenati di F, mentre F è il loro discendente. I discendenti di A sono tutti nodi che partono da esso. | |
| *Altezza e*  *sotto-alberi* | Altezza di un nodo  La lunghezza del più lungo cammino da un nodo ad una foglia si chiama altezza del nodo.  Tutte le foglie hanno altezza 0.  Es. Nell’albero sopra l’altezza di C è 1. | Altezza di un albero  Si definisce come l’altezza della sua radice, o il massimo livello delle sue foglie.  Es. L’albero sopra ha altezza 2. | | | Sotto-albero  Se T è un albero e X è un suo nodo, l’insieme dei nodi di T contenente X e tutti i suoi discendenti si chiama sotto-albero di T .  X si chiama radice del sotto-albero.  Es. Nell’albero sopra, se prendiamo il sotto-albero (C,F,G), C sarà la radice del sotto-albero, |
| *Attraversamento di un albero*  *(visita)* | Per attraversamento o visita di un albero si intende  l’ispezione dei nodi dell’albero in modo che tutti i nodi vengano ispezionati una ed una sola volta.  Quindi, un attraversamento di un albero definisce un  ordinamento totale tra i nodi dell’albero  • in base alla loro **posizione** nell’albero,  • NON in base alle loro etichette (che possono essere  non ordinabili)  Trattandosi di ordinamento totale, ogni nodo ha un  precedente e un successivo all’interno di un  attraversamento. | | | Ad esempio nell’albero di sopra i possibili attraversamenti sono:  A,B,C,D,E,F,G  B,C,D,E,F,G,A  E,F,G,B,C,D,A  …  Ogni permutazione dei nodi definisce un diverso attraversamento.  Certi attraversamenti procedono a salti e in pratica sono poco utili | |
| *Albero binario* | Un albero si dice binario se ogni nodo ha massimo due figli (detti rispettivamente **figlio sinistro** e **figlio destro**). | | C:\Users\Saro\Desktop\Immagine.png | | |
| In un albero binario, un nodo avente due figli si dice **pieno**. | |
| Un albero binario di altezza h si dice **completo** se tutti i nodi di livello minore di h sono pieni. | |
| *Teoremi sugli alberi binari completo* | Consideriamo un albero binario completo avente altezza h ed n nodi.  Possiamo esprimere n in funzione di h con il Teorema 1, e possiamo esprimere h in funzione di n con il 2. | | | | |
| Teorema 1. | | Teorema 2. | | |
| *Attraversamento dell’albero binario*  *PREORDER* | Visitare la radice, attraversare ricorsivamente il sotto-albero sinistro della radice ed infine il sotto-albero destro della radice | | Es. Dall’albero binario sopra  Attraversamento preorder:  A, B, D, E, C, F, G | | |
| *Attraversamento dell’albero binario*  *INORDER* | Attraversare ricorsivamente il sotto-albero  sinistro della radice, visitare la radice, e attraversare ricorsivamente il sotto-albero destro della radice | | Es. Dall’albero binario sopra  Attraversamento inorder:  D, B, E, A, F, C, G | | |
| *Attraversamento dell’albero binario*  *POSTORDER* | Attraversare ricorsivamente il sotto-albero  sinistro della radice, il sotto-albero destro della  radice, e infine visitare la radice. | | Es. Dall’albero binario sopra  Attraversamento postorder:  D, E, B, F, G, C, A | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Albero binario bilanciato* | Un albero binario si dice bilanciato (in altezza) se, per ogni nodo, la differenza di altezza dei due sottoalberi è zero o uno.  Un albero binario completo è bilanciato. | | | C:\Users\Saro\Desktop\Immagine.png  Bilanciato | | | C:\Users\Saro\Desktop\Immagine.png  Bilanciato | |
| C:\Users\Saro\Desktop\Immagine.png  Sbilanciato | | | C:\Users\Saro\Desktop\Immagine.png  Massimo sbilanciamento:  Struttura lineare | |
| *Albero binario di ricerca (BST)* | Un albero binario si dice di *ricerca* se, per ogni nodo, tutte le etichette del sottoalbero sinistro sono minori dell’etichetta del nodo, e tutte le etichette del sottoalbero destro sono maggiori dell’etichetta del nodo. | | | | | | | Esempio  C:\Users\Saro\Desktop\Immagine.png |
| *Predecessore e successore* | Dato un insieme totalmente ordinato, ha senso parlare di predecessore e di successore nell’insieme di un elemento dell’insieme.  Es. Dato l’insieme  e il suo elemento t:   * r è il predecessore in A di t; * v è il successore in A di t;   Il predecessore di b e il successore di y non sono definiti. | | | | Dato un BST e una sua chiave che si trova in un nodo avente sotto-albero sinistro non vuoto, il  **predecessore** della chiave (nell’insieme di etichette  del BST) si trova nel nodo più a destra del sotto-albero sinistro del nodo contenente la chiave. | | | |
| Dato un BST e una sua chiave che si trova in un nodo avente sotto-albero destro non vuoto, il **successore** della chiave (nell’insieme di etichette del BST) si trova nel nodo più a sinistra del sotto-albero destro del nodo contenente la chiave. | | | |
| C:\Users\Saro\Desktop\Immagine.png | * Predecessore di 10 è 9 * Predecessore di 13 è 10 | | | | Quindi, predecessore di una chiave è la chiave **massima** nel sotto-albero sinistro | | |
| * Successore di 13 è 20 * Successore di 25 è31 | | | | Quindi, successore di una chiave è la  chiave **minima** nel sotto-albero destro | | |
| *Implementazione statica di BST* | Se il numero dei nodi di un BST è fissato a priori, allora lo si può rappresentare con un array, dunque una struttura statica.  Ciascuna cella dell’array rappresenta un nodo.  Ciascuna cella è una struttura con 3 campi:  1. L’**etichetta** del nodo che la cella rappresenta  2. L’**indice** della cella dell’array che rappresenta il  nodo **figlio sinistro**  3. L’**indice** della cella dell’array che rappresenta il  nodo **figlio destro** | | | | Es. Si consideri il BST  Esso può essere implementato staticamente mediante il seguente array  C:\Users\Saro\Desktop\Immagine.pngC:\Users\Saro\Desktop\Immagine.png  la radice sta nella prima cella e –1 indica il figlio nullo. | | | |
| *Implementazione dinamica si BST* | Si consideri il BST di sopra.  Esso può essere implementato  dinamicamente mediante una  struttura in cui ogni nodo ha  due riferimenti ai nodi figli. | | C:\Users\Saro\Desktop\Immagine.png | | | | | |
| *Implementazione*  *dell’ attraversamento di un albero binario* | Abbiamo visto 3 attraversamenti per un albero binario:   * Preorder * Inorder * Postorder | | | | | | | |
| *Ricerca su un BST* | L’algoritmo per decidere se una chiave si trova in un BST avviene attraverso i seguenti controlli:   * Se il BST è vuoto restituisce null * Se la chiave coincide con l’etichetta della radice, si restituisce la radice * Se la chiave è *minore* dell’etichetta della radice, si esegue l’algoritmo sul sotto-albero *sinistro* della radice * Se la chiave è *maggiore* dell’etichetta della radice, si esegue l’algoritmo sul sotto-albero *destro* della radice | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Complessità della ricerca su BST* | | Se la chiave si trova al livello 0, abbiamo 0 assegnamenti, a livello 1 abbiamo un assegnamento, a livello 2 abbiamo 2 assegnamenti, e così via; quindi il numero di assegnamenti dipende dalla posizione in cui si trova la chiave cercata. | | | * Il numero **massimo** di assegnamenti è uguale   all’altezza del BST, quindi   * Il numero **medio** di assegnamenti è proporzionale   all’altezza del BST, quindi   * Se il BST è completo avremo * Se il il BST è bilanciato avremo * Se il BST è sbilanciato avremo | | | | | | | | | |
| Pertanto, più l’albero è sbilanciato, più la complessità della ricerca si avvicina a quella della ricerca su una  lista, ossia lineare. Più l’albero è bilanciato, più la complessità della ricerca si avvicina a quella logaritmica. | | | | | | | | | | | | |
| *Inserimento in un BST* | | * Se il BST è *vuoto*, si inserisca la chiave in un nuovo nodo che sarà radice * Si *ricerchi* la chiave. Se si trova, si restituisca false * Si determini la *foglia* che può essere *padre* del nuovo nodo * Si innesti il nuovo nodo come *figlio* della foglia trovata | | | | | | | | | | | | La complessità asintotica dell’inserimento è analoga a quella della ricerca  L’algoritmo di inserimento visto non mantiene il bilanciamento del BST |
| *Cancellazione da un BST* | | Per cancellare da un BST una certa chiave, mantenendo le proprietà di BST, abbiamo 4 casi:  0. Si ricerca la chiave, se non si trova si restituisce false | 1. Se la chiave si trova in una **foglia**, si metta a *null* il riferimento del nodo padre alla foglia  C:\Users\Saro\Desktop\Immagine.png | | | | | | | | 2. Se la chiave si trova in un nodo avente un solo **sotto-albero**, si faccia puntare al *figlio* il riferimento del nodo padre al nodo  C:\Users\Saro\Desktop\Immagine.png | | | |
| 3. Se la chiave si trova in un nodo con due **sotto-alberi**, si esegua la *fusione* dei due sotto-alberi o la **sostituzione** della chiave.  *Es*. cancellare 20 dall’albero sopra | | | | | | | | | | | | |
| *Fusione di due sottoalberi* | | Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi. Il procedimento è il seguente: (a destra) | | 1. Si innesti il sotto-albero destro alla destra del nodo contenente il predecessore della chiave da  cancellare  C:\Users\Saro\Desktop\Immagine.png | | | | | | | | | 2. Si innesti il sotto-albero sinistro modificato come da (1) al posto del nodo con la chiave da cancellare  C:\Users\Saro\Desktop\Immagine.png | |
| *Sostituzione di una chiave* | | 1. Si sostituisca la chiave col suo predecessore  C:\Users\Saro\Desktop\Immagine.png | | | | | | 2. Se il nodo che conteneva il predecessore è una *foglia*, si esegua il passo 1 dell’algoritmo di cancellazione  C:\Users\Saro\Desktop\Immagine.png. | | | | | | |
| 3. Se il nodo che conteneva il predecessore ha un sotto-albero, si esegua il passo 2 dell’algoritmo di cancellazione  C:\Users\Saro\Desktop\Immagine.png | | | | | La cancellazione per sostituzione diminuisce l’altezza del BST.  La complessità asintotica della  cancellazione è analoga a quella della ricerca. | | | | | Ripetute cancellazioni per sostituzione inframmezzate da inserimenti nel sotto-albero destro aumentano lo sbilanciamento. Si può ovviare al problema rendendo l’algoritmo  simmetrico, ossia alternando una sostituzione col  predecessore ad una sostituzione col successore. | | |
| Algoritmi di ordinamento | | | | | | | | | | | | | | | |
| *Definizioni* | | Gli ordinamenti costituiscono una classe molto importante tra gli algoritmi di uso comune. Essi consentono di riordinare una grande quantità di dati in base ad alcune chiavi (per esempio numeri o stringhe). | | | | | | | | | Per ordinare un insieme di dati:  primo passo: *scelta dei criteri di ordinamento*  · Es. ordinamento crescente o decrescente per  insieme di chiavi numeriche  · Es. ordinamento lessicografico per chiavi  alfanumeriche  secondo passo: *scelta dell’ algoritmo* basata sul tipo di chiavi e sulla dimensione dei dati | | | | |
| Il confronto tra la **complessità** degli algoritmi di ordinamento avviene valutando 2 importanti proprietà (indipendenti dalla macchina):   * il **numero dei confronti** tra le chiavi * il **numero di spostamenti** di dati   Il #confronti e #spostamenti possono non coincidere.  Poiché la complessità dello stesso algoritmo può  variare a seconda della proprietà che si considera, la scelta dell’algoritmo da usare va valutata in relazione alla dimensione dei dati ed al tipo di chiavi. | | | | | | | | |
| Tipi di ordinamento  C:\Users\Saro\Desktop\Immagine.png | | | | |
| *Selection sort* | | L’algoritmo seleziona di volta in volta il record con  chiave minima (oppure massima), spostandolo nella posizione corretta:  passo 1: il record con chiave più bassa viene selezionato e scambiato con l’elemento nella prima posizione  passo 2: tra i record rimanenti, si cerca quello di chiave minima e si scambia con il record in seconda posizione  passo i: tra i record dalla posizione i alla posizione n-1, si cerca quello di chiave minima e si scambia con il record in posizione i | | | | | | | | C:\Users\Saro\Desktop\Immagine.png | | | | | |
| Poiché ogni elemento è spostato massimo una volta, questo algoritmo è da preferire quando si devono ordinare insiemi di record molto grandi con chiavi piccole. | | | | | | | | | | | | | |
| *Insertion sort* | | Viene considerato un elemento alla volta, inserendolo in un sottogruppo che viene costruito già ordinato.  C:\Users\Saro\Desktop\Immagine.png  Per ogni nuovo elemento viene ricercata la posizione all'interno della parte ordinata, slittando gli elementi per creare uno spazio libero.  C:\Users\Saro\Desktop\Immagine.png | | | | | | | | passo 1: si confrontano i primi 2 elementi e si spostano se in ordine inverso  passo i: dal terzo elemento in poi, si seleziona l’elemento di posto i e si inserisce nella giusta posizione nel sottoarray ordinato dal posto 0 al posto i-1 slittando gli elementi per creare uno spazio libero.  C:\Users\Saro\Desktop\Immagine.png | | | | | |
| Il *vantaggio* dell’ordinamento per inserimento è che l’array viene ordinato solo quando è realmente necessario.  Lo *svantaggio* è che l’algoritmo non si accorge degli elementi che sono nella posizione corretta questi potrebbero venire spostati dalle loro posizioni per poi ritornarvi successivamente e questo può dar luogo a spostamenti ridondanti. | | | | | | | | | | | | | |
| *Bubble sort* | | Il bubble sort confronta elementi consecutivi (j e j-1) iniziando da destra, scambiandoli se non li trova in ordine. Al termine del primo ciclo viene così trovato il minimo, che galleggia in cima all’array.  Al i-esimo passaggio viene trovato il i-esimo elemento più piccolo, posizionandolo all’ i-esimo posto. | | | | | C:\Users\Saro\Desktop\Immagine.png | | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Complessità degli algoritmi di ordinamento semplici* | |  |  |  |  | | --- | --- | --- | --- | |  | **Selection** | **Insertion** | **Bubble** | | #Confronti | | | | | *Migliore* |  |  |  | | *Medio* |  |  |  | | *Peggiore* |  |  |  | | #Spostamenti | | | | | *Migliore* |  |  |  | | *Medio* |  |  |  | | *peggiore* |  |  |  | | | | Ricordando che definiamo in generale:   * **complessità costante** * **complessità lineare** * **complessità quadratica** * Caso migliore: quando l’array è già ordinato * Caso peggiore: quando l’array è in ordine inverso * Caso medio: elementi in ordine sparso | | |
| *Shell sort* | È un miglioramento dell’Insertion Sort. Esso parte dal principio che è più efficiente ordinare prima porzioni del sotto-array, e poi l’intero array originale   * **data** = array originale * **datai** = sotto-array * **h** = numero di sotto-array | | | | | |
| Scelta di h   * h *piccolo*, datai troppo grandi, quindi algoritmi di ordinamento ugualmente inefficienti * h *grande*, troppi datai, quindi l’ordine globale di data non risulta sostanzialmente modificato * h viene scelto non fisso, ma in modo incrementale | Calcolo di data  Fissato , data è diviso in sotto-array per    Esempio:  , diviso in , ,  dobbiamo dunque creare 3 sotto-array con elementi presi a 3 a 3:  [0]=data[0]=10; [1]=data[3]=2; [2]=data[6]=4;  [0]=data[1]=5; [1]=data[4]=9;  [0]=data[2]=7; [1]=data[5]=3 | | | | |
| *Quick sort* | Il quicksort è un algoritmo di ordinamento molto utilizzato nella pratica, perché mediamente risulta più efficiente degli altri. Lo svantaggio principale deriva dalla sua ricorsività.  Viene preso un elemento di riferimento, detto **pivot**, utilizzato per il confronto con gli altri elementi. L’array viene suddiviso in 2 sottoarray:   * il primo contiene elementi minori (o uguali) del pivot * il secondo contiene elementi maggiori (o uguali) al pivot.   Nella posizione centrale viene posto l’elemento pivot.  Si procede quindi in maniera ricorsiva a riordinare i due  sottoarray. | | | | | C:\Users\Saro\Desktop\Immagine.png |
| Scegliere il pivot  Per non causare sbilanciamento, il pivot andrebbe scelto in modo da suddividere l’array in sottoarray di dimensione circa *uguale*. Una delle possibili scelte è prendere come pivot il primo o ultimo elemento dell’array:  Viene preso come pivot il primo elemento a[0].  L’array viene scandito procedendo dai due estremi verso il centro, scambiando le eventuali coppie di elementi che si trovano in posizione errata.  Il ciclo termina quando gli indici si raggiungono.  Nella posizione centrale viene posto l’elemento a[0].  A questo punto l’array avrà la parte sinistra con gli elementi più piccoli di a[0], e la parte destra con quelli maggiori.  Si procede quindi in maniera ricorsiva sulle due parti rimanenti dell’array. | | | | C:\Users\Saro\Desktop\Immagine.png | |
| *Merge sort* | E’ un algoritmo che permette il riordino di un array  mediante la chiamata *ricorsiva* della procedura su due  metà dell’array da riordinare, seguite da un algoritmo  di merging.  Passo base: se l’array ha 2 elementi, con un confronto  ed un eventuale scambio si ottiene l’array ordinato.  Passo induttivo: se l’array ha più di 2 elementi, lo si  divide in 2 array, ciascuno contenente la metà degli  elementi.  I due array vengono prima ordinati (richiamando la stessa procedura su ciascuno di essi), e poi **fusi** in un unico array ordinato. | | | | C:\Users\Saro\Desktop\Immagine.png | |
| *Complessità degli algoritmi di ordinamento avanzati* | |  |  |  | | --- | --- | --- | |  | **Quick** | **Merge** | | #Confronti | | | | *Migliore* |  |  | | *Medio* |  |  | | *Peggiore* |  |  | | #Spostamenti | | | | *Migliore* |  |  | | *Medio* |  |  | | *peggiore* |  |  | | | Non esiste una stima precisa della complessità dello Shell Sort perché dipende dal valore ottimale per il passo di decremento .  Il Quick Sort non fa spostamenti dato che è ricorsivo  **complessità logaritmica/pseudolineare** | | | |

# Codici JAVA

|  |
| --- |
| Eccezioni e interfacce |
| /\*//////////////////////////  MECCANISMO TRY ... CATCH  \*//////////////////////////  // PseudoCodice  try{  fai qualcosa;  } catch (tipo-eccezione nome-eccezione){  fai qualcosa per correggere o segnalare l'eccezzione;  }  //////////ESEMPIO//////////////////////////////////////  public int f1(int[] a, int n){  throws ArrayIndexOutOfBoundsException;  return n\*a[n+2]  }  //f1() ritorna l'elemento dell'array a[n+1]\*n  public void f2(){  int[]a={1,2,3,4,5};  try{  for (int i=0;i<a.lentgh;i++)  System.outprint(f1(a,i)+" ");  } catch(ArrayIndexOutOfBoundsException e){  System.out.println(”Eccezione catturata in f2()”);  throw e;  }  }  /\* f2() crea un array di 5 elementi e stampa il metodo f1(a,i)  ma arrivato ad i=3 non potrà stampare a[5] perché non rientra nell'array.  Dunque il controllo del corpo del catch si accorge che è stato generata  un'eccezione "ArrayIndexOutOfBoundsException" e la chiama "e", stampa il messaggio  che l'eccessione è stata catturata (quindi non la risolve) e lo manda ad f1()  \*/  /\*//////////////////////////  THROW E CATTURA DELLE ECCEZIONI  \*//////////////////////////  class Eccezioni{  void f() throws Exception{  g();  }    void g() throws Exception{  h();  }    void h()throws Exception {  int x = 0;  if (x==0)  throw new Exception(“divisione per 0”);  int y=2/x;  }  }    /\* i metodi si richiamano a cascata indicando nei costruttori che potrebbero  esserci eccezioni. In questo caso solamente col throw stiamo lanciando l'eccezione  ma non la stiamo catturando, quindi essa si propaga e interrompe il programma  \*/    class Eccezioni{  void f() throws Exception{  g();  }    void g()throws Exception {  h();  }    void h()throws Exception {  int x =0;  try{  int y= 2/x;  }catch(ArithmeticException ae){  System.out.print(“errore!”)  }  }  }  /\* in questo caso l'eccezzione è catturata dal catch, il quale stamperà a  console l'errore ma non lo farà propagare, e il programma continua a compilare  \*/    /\*//////////////////////////  Esempio di interfaccia  \*//////////////////////////  interface Nuotatore{  public void nuota();  }  class Acquario{  Nuotatore[] elementi = new Nuotatore[10];  //metodi della classe  }  class Pesce implements Nuotatore{  public void nuota(){  …  }  }  class Crostaceo{  …  }  class Granchio extends Crostaceo implements Nuotatore{  public void nuota(){  …  }  }  /\* Le interfacce non si possono istanziare. Prendiamo l’esempio sopra:  Nuotatore n = new Nuotatore(); NO  Posso però assegnare i tipi che sono stati implementati  Nuotatore p = new Pesce(); SI  \*/ |

|  |
| --- |
| Java I/O |
| /\* FILEINPUSTREAM  Lettura da un file binario con il FileInputStream  \*/  import java.io.\*;  public class LetturaDaFileBinario {  public static void main(String args[]){  FileInputStream file = null;  try {  file = new FileInputStream(args[0]); //args[0] potrebbe essere un percorso di un file nel PC  } catch(FileNotFoundException e){  System.out.println("File non trovato");  System.exit(1);  }  try { //Inizio fase di lettura e stampa su console del carattere letto  int x;  int n = 0;  while ((x = file.read())>=0){ //quando lo stream termina, read() restituisce -1 ed esce dal while  System.out.print(" " + x);  n++;  }  System.out.println("Totale byte: " + n);  } catch(IOException ex){  System.out.println("Errore di input");  System.exit(2);  }  }  }  /\* FILEOUTPUTSTREAM  Scrittura su un file binario con il FileOutputStream  \*/  import java.io.\*;  public class ScritturaSuFileBinario {  public static void main(String args[]){  FileOutputStream file = null;  try {  file = new FileOutputStream(args[0]);  }catch(FileNotFoundException e){  System.out.println("Imposs. aprire file");  System.exit(1);  }  try { //fase di scrittura  for (int x=0; x<10; x+=3) {  System.out.println("Scrittura di " + x);  file.write(x);  }  } catch(IOException ex){  System.out.println("Errore di output");  System.exit(2);  }  }  }  /\* DATAINPUTSTREAM  FileInpuStreamReader legge solo byte, dobbiamo incapsularlo dentro lo steam DataInputStream  per leggere float, int, double, boolean  \*/  import java.io.\*;  public class LetturaTipi {  public static void main(String args[]){  FileInputStream fin = null;  try {  fin = new FileInputStream("Prova.dat");  }catch(FileNotFoundException e){  System.out.println("File non trovato");  System.exit(3);  }  DataInputStream is = new DataInputStream(fin); //Creazione del DataInputStream con argomento il FileInputStream  float f2; char c2; boolean b2; double d2; int i2;  try {  f2 = is.readFloat(); b2 = is.readBoolean();  d2 = is.readDouble(); c2 = is.readChar();  i2 = is.readInt();  is.close();  System.out.println(f2 + ", " + b2 + ", "+ d2 + ", " + c2 + ", " + i2);  } catch (IOException e){  System.out.println("Errore di input");  System.exit(4);  }  }  /\* DATAOUTPUTSTREAM  Stessa procedura dell'input per l'incapsulamento  \*/  import java.io.\*;  public class Scrittura Tipi {  public static void main(String args[]){  FileOutputStream fs = null;  try {  fs = new FileOutputStream("Prova.dat");  }catch(IOException e){  System.out.println("Apertura fallita");  System.exit(1);  }  DataOutputStream os = new DataOutputStream(fs);  float f1 = 3.1415F; char c1 = 'X';  boolean b1 = true; double d1 = 1.4142;  try {  os.writeFloat(f1); os.writeBoolean(b1);  os.writeDouble(d1); os.writeChar(c1);  os.writeInt(12); os.close();  } catch (IOException e){  System.out.println("Scrittura fallita");  System.exit(2);  }  }  }  /\* CHARACTERSTREAM  Lettura da testo UNICODE  \*/  import java.io.\*;  public class LetturaDaFileDiTesto {  public static void main(String args[]){  FileReader r = null;  try {  r = new FileReader(args[0]);  }catch(FileNotFoundException e){  System.out.println("File non trovato");  System.exit(1);  }  try {  int n=0, x;  while ((x = r.read())>=0) {  char ch = (char) x;  System.out.print(" " + ch); n++; //Cast esplicito da int a char - Ma solo se è stato davvero letto un carattere (cioè se non è stato letto -1)  }  System.out.println("\nTotale caratteri: " + n);  } catch(IOException ex){  System.out.println("Errore di input");  System.exit(2);  }  }  }  /\* SYSTEM.IN  Essa è interpretata come un Reader incapsulato dentro un InputStreamReader  il quale a sa volta è incapsulato dentro un lettore Bufferizzato  \*/  import java.io.\*;  public class SistemDentro {  public static void main(String[] args){  BufferedReader console = new BufferedReader (new InputStreamReader (System.in));  System.out.println("Inserisci una riga di testo");  try{  String str = console.readLine();  } catch (IOException e){  System.out.println(e);  System.exit(1);  }  }  }  /\* CONSOLEREADER  Racchiude tutti i termini e i metodi della classe System per quanto riguarda l’input e l’output di dati  \*/  import java.io.\*;  public class LettoreC{  private BufferedReader reader;  public LettoreC(){  reader=new BufferedReader(new InputStreamReader (System.in));  }    public String readLine(){ //leggi una stringa  String inputLine="";  try{  inputLine = reader.readLine();  }catch(IOException e){  System.out.println(e);  System.exit(1);  }  return inputLine;  }    public int readInt (){  String inputString = readLine();  int n = Integer.parseInt(inputString);  return n;  }  public double readDouble (){  String inputString = readLine();  double x = Double.parseDouble(inputString);  return x;  }    public static void main (String [] args){  LettoreC lettore = new LettoreC();  System.out.println("come ti chiami?");  String x = lettore.readLine();  System.out.println("ciao "+x+", quanti anni hai?");  int eta = lettore.readInt();  System.out.println(eta <10? "solo "+eta+" anni? che piccolo!":eta+" anni? sei grande!");  }  }  /\* LETTURA E SCRITTURA SU FILE  Insieme delle classi di Java.IO per leggere e scrivere dati su file  \*/  public class LetturaScritturaFile{  public static void main (String [] args){  int n;  ConsoleReader console = new ConsoleReader();  System.out.print("inserisci il nome del file di input : ");  String FileIn = console.readLine();  System.out.print("inserisci il nome del file di output : ");  String FileOut = console.readLine();  try {  FileReader lettore = new FileReader(FileIn);  FileWriter scrittore = new FileWriter(FileOut);  while ((n=lettore.read()) !=-1){  scrittore.write((char)n);  }  lettore.close();  scrittore.close();  }  catch(FileNotFoundException e){  System.out.println(e);  System.exit(1);  }  catch(IOException e){  System.out.println(e);  System.exit(1);  }  }  } |

|  |
| --- |
| Liste concatenate |
| /\* NODO  Struttura di un Nodo della lista semplice  \*/  class Nodo{  private int info; //valore del nodo  private Nodo next; //riferimento al prossimo nodo  public Nodo(int val){  this(val, null);  }    public Nodo (int val, Nodo n){  info = val;  next = n;  }  public void setInfo(int val){  info = val;  }  public int getInfo(){  return info;  }  public void setNext(Nodo n){  next = n;  }  public Nodo getNext(){  return next;  }  }  /\* NODO AUSILIARIO  Per evitare di scrivere  ((head.getNext()).getNext).setNext(new Nodo(99));  si crea un nodo ausiliaro a cui viene assegnato un nodo a noi interessato  \*/  Nodo head = new Nodo(13);  Nodo aux = head;  aux.setNext(new Nodo(16));  aux = aux.getNext();  aux.setNext(new Nodo(18));  /\* LISTA CONCATENATA  Gli elementi vengono aggiunti dinamicamente solo quando è necessario.  Ogni elemento contiene un riferimento all’elemento successivo.  Serve anche un identificatore esterno (aux) per tener traccia della lista.  \*/  class Lista{  private Nodo head;  public Lista(){  head = null; //in caso di lista vuota  }  public void InsertHead(int val){  head = new Nodo(val,head);  }    public boolean IsEmpty(){ //controlla se la lista è vuota  return (head == null ? true : false);  }  public void InsertTail(int val){  if (IsEmpty())  head = new Nodo(val); //se la lista è vuota inserisci il Nodo in testa  else{  Nodo aux = head;  for( ; aux.next != null; aux = aux.next); //scorrimento lista, porta aux all'ultimo nodo  aux.next = new Nodo(val);  }  }    public void InsertOrdered(int val){  if (IsEmpty())  head = new Nodo(val);  else  if( head.info > val) //se il valore della testa è maggiore di val...  head = new Nodo(val,head); //..il Nodo con val diventa la nuova testa  else{  Nodo aux = head;  for(;(aux.next!=null) && (aux.next.info<val);aux=aux.next); //scorri fino a quando la lista finisce && valore di aux è minore di val  aux.next = new Nodo(val,aux.next);  }  }    public Nodo SearchOrd(int key){  Nodo aux = head;  for( ; (aux != null) && (aux.info < key); aux = aux.next); //ricerca la lista è in ordine crescente, per il decrescente basta mettere >  if((aux != null) && (aux.info == key))  return aux;  return null;  }    public Nodo Search(int key){  Nodo aux = head;  for(; (aux != null) && (aux.info != key); aux = aux.next); //la lista scorre fino a quando aux.info == key, o aux.info == null, in quest'ultimo caso la chiave non esiste nella lista  return aux;  }    public int DeleteHead(){  if(IsEmpty())  return 0;  else{  Nodo aux = head;  head = head.next; //la nuova testa sarà il vecchio secondo nodo della lista  return aux.info;  }  }    public int DeleteTail(){  if (IsEmpty())  return 0;  else{  Nodo aux = head;  Nodo prev = null; //è un altro ausiliare ma per il nodo precedente  for( ; aux.next != null; prev = aux, aux = aux.next); //scorre la lista assegnando 2 ausiliari  if(prev == null) //se prev == null vuol dire che abbiamo solo un elemento nella lista  head = null;  else  prev.next = null; //dato che prev è il penultimo, prev.next è l'ultimo è viene cancellato  return aux.info;  }  }    public Nodo DeleteKey(int key){  if (IsEmpty())  return null;  else{  Nodo aux = head;  Nodo prev = null;  for(; (aux != null) && (aux.info != key); prev = aux, aux = aux.next);  if(aux != null)  if(prev == null)  head = head.next;  else  prev.next = aux.next; //unisci il riferimento del precedente del nodo A al successivo del nodo A  return aux;  }  }  }  /\* NODO DOPPIO  Struttura di un Nodo della lista doppiamente concatenata  \*/  public class NodoDbl{  public int info;  public NodoDbl next, prev; //riferimento al successivo e al precedente  public NodoDbl(int val){  this(val, null, null);  }  public NodoDbl(int val, NodoDbl n, NodoDbl p){  info = val;  next = n;  prev = p;  }  }  /\* LISTA DOPPIAMENTE CONCATENATA  Lista che permette di scorre sia dalla testa che dalla coda.  A tale scopo sono necessari due riferimenti, uno all’oggetto precedente e l’altro al successivo.  \*/  public class ListaDbl{  private NodoDbl head;  private NodoDbl tail; //tail è usato per poter scorrere la lista anche al contrario  public ListaDbl(){  head = tail = null;  }  public boolean IsEmpty(){  return head == null;  }    public void InsertHead(int val){  if ( IsEmpty() )  head = tail = new NodoDbl(val);  else{  head = new NodoDbl(val,head,null);  head.next.prev = head; //sistemazione del riferimento al nodo precedente della vecchia testa all'attuale testa  }  }  public void InsertTail(int val){  if ( IsEmpty() )  head = tail = new NodoDbl(val);  else{  tail = new NodoDbl(val,null,tail);  tail.prev.next = tail; //sistemazione del riferimento al nodo successivo della vecchia coda all'attuale coda  }  }  public void InsertOrdered(int val){  if (IsEmpty())  head = tail = new NodoDbl(val);  else  if(head.info >= val) //se l'elemento da inserire è il primo  InsertHead(val);  else{  NodoDbl aux= head;  for(; (aux!=null) && (aux.info<val); aux=aux.next); //scorri la lista  if (aux == null)  InsertTail(val); //se l'elemento da inserire è l'ultimo  else{ //se l'elemento da inserire è nel mezzo...  aux.prev = new NodoDbl(val,aux,aux.prev); //...inserisci un nuovo nodo col riferimento al nodo successivo della lista e a quello precedente  aux.prev.prev.next = aux.prev; //sistemazione del riferimento al nodo successivo del nodo precedente al precedente del nodo da inserire al nodo da inserire  }  }  }  public NodoDbl SearchOrd(int key){  NodoDbl aux = head;  for(; (aux != null) && (aux.info < key); aux = aux.next);  if((aux != null) && (aux.info == key))  return aux;  return null;  }  public NodoDbl Search(int key){  NodoDbl aux = head;  for(; (aux != null) && (aux.info != key); aux = aux.next);  return aux;  }    public int DeleteHead(){  if(IsEmpty())  return 0;  else{  NodoDbl aux = head;  if (head == tail)  head = tail = null;  else{  head = head.next;  head.prev = null;  }  return aux.info;  }  }  public int DeleteTail(){  if(IsEmpty())  return 0;  else{  NodoDbl aux = tail;  if (head == tail)  head = tail = null;  else{  tail = tail.prev;  tail.next = null;  }  return aux.info;  }  }    public NodoDbl DeleteKey(int key){  NodoDbl aux = head;  for(;(aux!=null) && (aux.info!=key); aux=aux.next);  if (aux != null) // è stato trovato un elemento  if(aux.prev == null){ // è la testa  if (head == tail) // un solo elemento nella lista  head = tail = null;  else{  head = head.next;  head.prev = null;  }  }  else{  if (aux == tail){  tail = tail.prev; // è la coda  tail.next = null;  }  else{ // è un elemento nel mezzo  aux.prev.next = aux.next;  aux.next.prev = aux.prev;  }  }  return aux;  }  } |

|  |
| --- |
| Pila |
| public class Pila{  private int top; //indice per l'ultimo elem inserito  private final int MAX; //massimo contenuto di una pila  private int elem[]; //array degli elementi della pila  private static final int MAXDEFAULT = 10;  public Pila(){  this(MAXDEFAULT);  }  public Pila(int max){  top = 0;  MAX = max;  elem = new int[MAX];  }  public boolean IsFull(){  return (top == MAX);  }  public boolean IsEmpty(){  return (top == 0);  }    public void Clear(){  top = 0;  }  public boolean Push(int val){ //inserimento  if (IsFull())  return false;  elem[top++] = val;  return true;  }    public int Pop(){ //estrazione  if (IsEmpty())  return 0;  return elem[--top];  }    public int TopElem(){ //restituisce il valore dell'elemento in cima  if (IsEmpty())  return 0;  return elem[top-1]; //ritorna top-1 poiché l'array elem[] (come tutti gli array) inizia dall'indice 0, dove c'è il primo elemento inserito nella pila  }  } |

|  |
| --- |
| Coda |
| public class Queue{  private int head, tail;  private final int MAX;  private int elem[];  private static final int MAXDEFAULT = 10;  public Queue(){  this(MAXDEFAULT);  }  public Queue(int max){  head = tail = 0;  MAX = max;  elem = new int[MAX];  }  public boolean IsFull(){  return (head == (tail+1) % MAX);  }    public boolean IsEmpty(){  return ( head == tail );  }    public void ClearQueue(){  head = tail = 0;  }    public int getFirstElem(){  if(IsEmpty())  return 0;  return elem[head];  }    public boolean EnQueue(int val){ //inserimento in coda  if(IsFull())  return false;  elem[tail] = val;  tail = ++tail % MAX;  return true;  }  public int DeQueue(){ //estrazione della testa  if(IsEmpty())  return 0;  int val = elem[head];  head = ++head % MAX;  return val;  }  } |

|  |
| --- |
| Ricorsione |
| /\* FATTORIALE  metodi iterativi e ricorsivi per il calcolo del fattoriale  \*/  public static int iterFactorial(int n){  if(n<0)  throw new IllegalArgumentException();  else if (n==0) return 1;  else{  int p=1;  for (int i=2; i<=n; i++)  p=p\*i;  return p;  }  }  public static int ricFactorial(int n){  if(n<0)  throw new IllegalArgumentException();  else if (n==0) return 1;  else  return n \* factorial(n-1);  }  /\* ESPONENZIALE  metodi iterativi e ricorsivi per il calcolo dell'esponenziale  \*/  public int ricEsp(int x,int y){  if (y==0) return 1;  else return x\*esp(x,y-1);  }  public double iterEsp(int x, int y){  if(y<0)  throw new IllegalArgumentException();  else{  if(y==0) return 1;  else{  int ris=x;  while (y>1){  ris=ris\*x;  y--;  }  return ris;  }    /\* INVERTI STRINGA  Inversione di una stringa presa in input  (Esempio di ricorsione non in coda, dove la chiamata al metodo stesso non è l’ultima azione compiuta)  \*/  public void reverse(){  char ch = (char)System.in.read();  if (ch != ‘\n’){  reverse();  System.out.print(ch);  }  }  /\* FIBONACCI  metodi iterativi e ricorsivi per il calcolo della successione di Fibonacci  (Esempio di ricorsione multipla)  \*/  public static int ricFibonacci(int n){  if(n<0)  throw new IllegalArgumentException();  else if (n<2) return n;  else  return fibonacci(n-2) + fibonacci(n-1);  }  public static int iterFibonacci(int n){  if (n<2) return n;  else{  int i=2, tmp, current=1, last=0;  for (;i<=n; ++i){  tmp = current;  current += last;  last = tmp;  }  return current;  }  /\* RICORSIONI VARIE  Altri esempi di ricorsione  \*/  //Massimo comune divisore  public static int MCD (int x, int y){  if (x==y) return x;  else if (x>y) return MCD(x-y,y);  else return MCD (x,y-x);  }    //Minimo comune multiplo  public static int mcm(int x, int y){  if (x==0 && y==0) return 0;  else return (x\*y)/(MCD(x,y));  }    //Soluzione dela torre di Hanoi  public class TowersOfHanoi{  private int totalDisks;    public TowersOfHanoi(int disks){  totalDisks = disks;  }    public void solve(){  moveTower(totalDisks, 1, 3, 2);  }    private void moveTower(int numDisks, int start, int end, int temp){  if (numDisks == 1)  move OneDisk(start, end);  else{  moveTower(numDisks-1, start, temp, end);  moveOneDisk(start, end);  moveTower(numDisks-1, temp, end, start);  }  }    private void moveOneDisk(int start, int end){  System.out.println("Sposta un disco da "+start+" a "+end);  }  } |

|  |
| --- |
| BST |
| /\* IMPLEMENTAZIONE DI UN BST  Un BST può essere implementato dinamicamente mediante una struttura  in cui ogni nodo ha due riferimenti ai nodi figli  \*/  //NODO di un BST di interi  public class NodoBST{  protected int key;  protected NodoBST left, right; //figlio sinisto e figlio destro  public NodoBST(){  left = right = null;  }    public NodoBST(int val){  this(val, null, null);  public NodoBST(int val, NodoBST sinistro, NodoBST destro){  key = val; left = sinistro; right = destro; //riferimenti ai figli  }    public int visit(){ //questo metodo è da gestire, può ritornare la key, stamparla o altro a seconda dell'uso necessario  return key;  }  }  //BST di interi  public class BST{  protected NodoBST root;  public BST(){  root = null;  }  }  /\* ATTRAVERSAMENTO PREORDER  - visitare la radice  - attraversare ricorsivamente il sotto-albero sinistro della radice  - attraversare ricorsivamente il sotto-albero destro della radice  \*/  protected void ricPreorder(NodoBST p){  if (p != null){  p.visit();  preorder(p.left);  preorder(p.right);  }  }  protected void iterPreorder(){  NodoBST p = root;  Pila aiuto = new Pila();  if (p != null){  aiuto.push(p);  while (!aiuto.isEmpty()){  p = (NodoBST) aiuto.pop();  p.visit();  if (p.right != null) aiuto.push(p.right);  if (p.left != null) aiuto.push(p.left);  }  }  }  /\* ATTRAVERSAMENTO INORDER  - attraversare ricorsivamente il sotto-albero sinistro della radice  - visitare la radice  - attraversare ricorsivamente il sotto-albero destro della radice  \*/  protected void ricInorder(NodoBST p){  if (p != null){  inorder(p.left);  p.visit();  inorder(p.right);  }  }  protected void iterInorder(){  NodoBST p = root;  Pila aiuto = new Pila();  while (p != null){  while (p != null){  if (p.right != null) aiuto.push(p.right); // impila figlio dx se esiste, e il nodo stesso procedendo verso sx  aiuto.push(p);  p = p.left;  }  p = (NodoBST) aiuto.pop(); // estrai un nodo senza figlio sinistro  while (!aiuto.isEmpty() && p.right == null){ //visita nodo e tutti quelli senza figlio dx  p.visit();  p = (NodoBST) aiuto.pop();  }  p.visit(); // visita anche il primo nodo con un figlio dx  if (!aiuto.isEmpty())  p = (NodoBST) aiuto.pop();  else p = null;  }  }  /\* ATTRAVERSAMENTO POSTORDER  - attraversare ricorsivamente il sotto-albero sinistro della radice  - attraversare ricorsivamente il sotto-albero destro della radice  - visitare la radice  \*/  protected void ricPostorder(NodoBST p){  if (p != null){  postorder(p.left);  postorder(p.right);  p.visit();  }  }  protected void iterPostorder(){  NodoBST p = root, q = root;  Pila aiuto = new Pila();  while (p != null){  for (; p.left != null; p = p.left) aiuto.push(p);  while (p != null && (p.right == null || p.right == q)){  p.visit();  q = p;  if (aiuto.isEmpty()) return;  p = (NodoBST) aiuto.pop();  }  aiuto.push(p);  p = p.right;  }  }  /\* RICERCA SU UN BST  L’algoritmo per decidere se una chiave si trova in un BST avviene  attraverso i seguenti controlli  \*/  public NodoBST ricerca (NodoBST p, int val){  while (p != null) //Se il BST è vuoto restituisce null  if (val == p.key) return p; //Se la chiave coincide con l’etichetta della radice, si restituisce la radice  else if (val < p.key) p = p.left; //Se la chiave è minore dell’etichetta della radice, si esegue l’algoritmo sul sotto-albero sinistro della radice  else p = p.right; //Se la chiave è maggiore dell’etichetta della radice, si esegue l’algoritmo sul sotto-albero destro della radice  return null;  }  /\* INSERIMENTO IN UN BST  Inserisce un nuovo nodo in base al valore di esso  \*/  public boolean inserisci (int val){  if (root == null)  root = new NodoBST(val); //Se il BST è vuoto, la chiave diventa la radice  else{  NodoBST p = root, prev = null;  while (p != null){ //Ricerca della chiave, scorrimento dell'albero  if (val == p.key)  return false; //Ritorna false se la chiave è già presente  prev = p;  if (val < p.key)  p = p.left;  else  p = p.right;  }  if (val < prev.key) //Si determina la foglia che può essere il padre del nuovo nodo  prev.left = new NodoBST(val); //Si innesta il nuovo nodo come figlio della foglia trovata  else  prev.right = new NodoBST(val);  }  return true;  }  /\* CANCELLAZIONE DA UN BST  Per cancellare da un BST una certa chiave, mantenendo le proprietà di BST, abbiamo 4 casi:  0. Si ricerca la chiave, se non si trova si restituisce false  1. Se la chiave si trova in una foglia, si metta a null il riferimento del nodo padre alla foglia  2. Se la chiave si trova in un nodo avente un solo sotto-albero, si faccia puntare al figlio il riferimento del nodo padre al nodo  3. Se la chiave si trova in un nodo con due sotto-alberi, si esegua la fusione dei due sotto-alberi o la sostituzione della chiave.  \*/  public int cancella (int val){  NodoBST nodo, p = root, prev = null;  while (p != null && p.key != val){  prev = p;  if (val < p.key) p = p.left;  else p = p.right;  }  nodo = p;  if (p != null && p.key == val){  if (nodo.right == null) nodo = nodo.left; //passi 1 e 2  else if (nodo.left == null) nodo = nodo.right;  else fondiSottoAlberi() //passo 3  sostituisciChiave() //in alternativa al precedente  if (p == root) root = nodo; //continua  else if (prev.left == p) prev.left = nodo; //passi 1 e 2  else prev.right = nodo;  return 0; //cancellazione effettuata  }  else if (root != null) //casi limite  return –1; //chiave non presente nel BST  else  return –2; //BST vuoto  }  //FUZIONE DI DUE SOTTOALBERI  {NodoBST tmp = nodo.left;  while (tmp.right != null)  tmp = tmp.right;  tmp.right = nodo.right;  nodo = nodo.left;  }  //SOSTITUZIONE DI DUE SOTTOALBERI  { NodoBST tmp = nodo.left;  NodoBST previous = nodo;  while (tmp.right != null)  { previous = tmp;  tmp = tmp.right;  }  nodo.key = tmp.key;  if (previous == nodo)  previous.left = tmp.left;  else  previous.right = tmp.left;  }  /\* OPERAZIONI VARIE \*/  protected void LetturaLivelli(){  NodoBST p = root;  Coda aiuto = new Coda();  if (p != null){  aiuto.Enqueue(p);  while (!aiuto.isEmpty()){  p = (IntBSTNodo) aiuto.Dequeue();  p.visit();  if (p.left != null) aiuto.Enqueue(p.left);  if (p.right != null) aiuto.Enqueue(p.right);  }  }  } |

|  |
| --- |
| Ordinamento semplice |
| /\* SWAP  Per scambiare il valore di due elementi in un array  serve un indice temporaneo  \*/  void swap(int array[], int precedente, int successivo) {  int tmp = array[precedente];  array[precedente] = array[successivo];  array[successivo] = tmp;  }  /\* SELECTION SORT  L’algoritmo seleziona di volta in volta il record con  chiave minima (oppure massima), spostandolo nella posizione corretta  \*/  public void SelectionSort(int [] data) {  int i,j,minimo;  for (i = 0; i < data.length-1; i++) {  for (j=i+1, minimo=i; j<data.length; j++)  if (data[j]<data[minimo])  minimo = j;  swap(data,minimo,i);  }  }  //complessità #confronti = migliore O(n^2), medio O(n^2), peggiore O(n^2)  //complessità #spostamenti = migliore O(1), medio O(n), peggiore O(n)  /\* INSERTION SORT  Ogni elemento dell’array viene spostato in un sotto-array che viene ordinato,  facendo poi shiftare il processo al numero successivo  \*/  public void InsertionSort(int[] data) {  int i, j,tmp;  for (i = 1; i < data.length; i++) {  tmp = data[i];  for (j=i; (j>0)&&(tmp<data[j-1]); j--)  data[j] = data[j-1];  data[j] = tmp;  }  }  //complessità #confronti = migliore O(n), medio O(n^2), peggiore O(n^2)  //complessità #spostamenti = migliore O(n), medio O(n^2), peggiore O(n^2)  /\* BUBBLE SORT  Confronta elementi consecutivi (j e j-1) iniziando da destra,  scambiandoli se non li trova in ordine. Al termine del primo  ciclo viene così trovato il minimo, che galleggia in cima all’array.  Al i-esimo passaggio viene trovato il i-esimo elemento più piccolo,  posizionandolo all’ i-esimo posto.  \*/  public void BubbleSort(int [] data){  for (int pass = 0; pass < data.length-1; pass++)  for (i = 1; i<data.length;i++)  if (data[i]<data[i-1])  swap(data,i,i-1);  }    //complessità #confronti = migliore O(n^2), medio O(n^2), peggiore O(n^2)  //complessità #spostamenti = migliore O(1), medio O(n^2), peggiore O(n^2) |

|  |
| --- |
| Ordinamento avanzato |
| /\* SHELL SORT  Ordina prima porzioni del sotto-array, e poi l'intero array originale  \*/  void ShellSort (int [] data) {  int i, j, k, h, hContatore, tmp, incrementi[] = new int[20];  for (h = 1, i = 0; h < data.length; i++) { // crea il numero corretto di incrementi h in base alla formula generale  incrementi[i] = h;  h = 3\*h + 1;  }  for (i--; i >= 0; i--) { // itera per il numero dei diversi incrementi h  h = incrementi[i];  for (hContatore = h; hContatore < 2\*h; hContatore++) { // itera per il numero di sottoarray ordinati-h nel passo i-mo  for (j = hContatore; j < data.length; ) { // ordina per insertion sort il sottoarray contenente ogni h-mo elemento dell’ array “data”  tmp = data[j];  k = j;  while ((k-h>=0) && (tmp<data[k-h])){  data[k] = data[k-h];  k -= h;  }  data[k] = tmp;  j += h;  }  }  }  }  /\* QUICK SORT  Viene preso un elemento di riferimento, detto pivot, utilizzato per il confronto con gli altri elementi.  L’array viene suddiviso in 2 sottoarray:  - il primo contiene elementi minori (o uguali) del pivot  - il secondo contiene elementi maggiori (o uguali) al pivot.  Nella posizione centrale viene posto l’elemento pivot.  Si procede quindi in maniera ricorsiva a riordinare i due sottoarray.  \*/  public void QuickSort(int[] data, int first, int last) {  int lower = first + 1, upper = last, pivot = data[first];  while (lower <= upper) {  while (data[lower]<pivot)  lower++;  while (pivot<data[upper])  upper--;  if (lower < upper)  swap(data,lower++,upper--);  else lower++;  }  swap(data,upper,first);  if (first < upper-1)  QuickSort(data,first,upper-1);  if (upper+1 < last)  QuickSort(data,upper+1,last);  }  //complessità #confronti = migliore O(n lg n), medio O(n lg n), peggiore O(n^2)  /\* MERGE SORT  E’ un algoritmo che permette il riordino di un array  mediante la chiamata ricorsiva della procedura su due  metà dell’array da riordinare, seguite da un algoritmo  di merging.  \*/  void MergeSort(int [] data, int first, int last) {  if (first < last) {  int mid = (first + last) / 2;  MergeSort(data, first, mid);  MergeSort(data, mid+1, last);  merge(data, first, last);  }  }  int[] temp; // usato da merge();  void merge(int[] data, int first, int last) {  int mid = (first + last) / 2;  int i1 = 0, i2 = first, i3 = mid + 1;  while (i2 <= mid && i3 <= last)  if (data[i2]<data[i3])  temp[i1++] = data[i2++];  else temp[i1++] = data[i3++];  while (i2 <= mid)  temp[i1++] = data[i2++];  while (i3 <= last)  temp[i1++] = data[i3++];  for (i1 = 0, i2 = first; i2 <= last;  data[i2++] = temp[i1++]);  }  void MergeSort(int[] data) { // per occultare i limiti dell'array  temp = new int [data.length];  MergeSort(data,0,data.length-1);  }  //complessità #confronti = migliore O(n lg n), medio O(n lg n), peggiore O(n lg n)  //complessità #spostamenti = migliore O(n lg n), medio O(n lg n), peggiore O(n lg n) |