

Esempi di strutture dati e algoritmi

Strutture dati e algoritmi

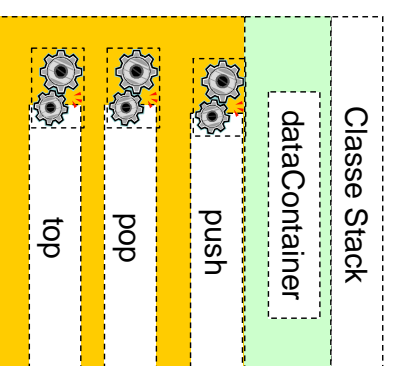
- Esistono delle *strutture dati* e degli *algoritmi* che sono usate frequentemente in informatica
- Le strutture dati includono *insiemi, pile, code, liste, alberi, grafi,...*
- Gli algoritmi includono l'*ordinamento, la visita dei grafi, la ricerca di dati in vettori, ...*
- Nel seguito studieremo alcune di queste strutture e algoritmi

Insiemi

- Vi sono molti modi di implementare gli insiemi che differiscono a seconda del modo in cui si possono inserire e rimuovere gli elementi, l'efficienza delle operazioni, l'occupazione di memoria, :
 - **Pile**: gli elementi si possono rimuovere solo in ordine inverso a quello di inserimento
 - **Code**: gli elementi si possono rimuovere solo nello stesso ordine in cui sono stati inseriti
 - **Vettori**: gli elementi possono essere rimossi e inseriti in qualsiasi ordine. Occorre conoscere fin dall'inizio il numero di elementi dell'insieme
 - **Liste**: gli elementi possono essere rimossi e inseriti in qualsiasi ordine. La gestione delle liste è più complessa.

La pila (stack)

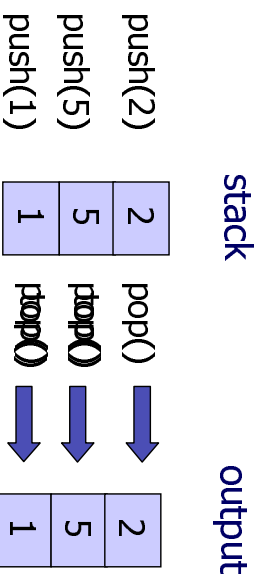
- La pila simula un contenitore in cui gli elementi si trovano uno sopra l'altro
- La pila è un oggetto con tre metodi
 - **push**: inserisce un elemento nella pila
 - **pop**: estrae un elemento dalla pila
 - **top**: restituisce l'elemento in testa alla pila senza rimuoverlo



- La pila implementa una strategia **FILO** (First In, Last Output)

La pila: un esempio

```
Stack myStack=new Stack(100);
myStack.push(1);
myStack.push(5);
myStack.push(2);
int a=myStack.pop();
System.out.print(a);
System.out.print(myStack.top());
System.out.print(myStack.pop());
System.out.print(myStack.top());
System.out.print(myStack.pop());
```



output → 25511

Implementazione di una pila

- Per implementare una pila si utilizza un array che funge da contenitore e un indice che indica dove si trova l'elemento in testa alla pila
- Il costruttore permette di inizializzare il contenitore con un numero di elementi voluto

```
class Stack{
    private int dataContainer[];
    private int head;
    Stack(int length) {
        dataContainer=new int[length];
        head=-1;
    }
    // continua nella prossima slide
```

Implementazione di una pila: pop, push e top

```
// continua dalla slide precedente
public int top() {
    return dataContainer[head];
}
public int pop() {
    head--;
    return dataContainer[head+1];
}
public void push(int in) {
    head++;
    dataContainer[head]=in;
}
}
```

Quando si usano le pile?

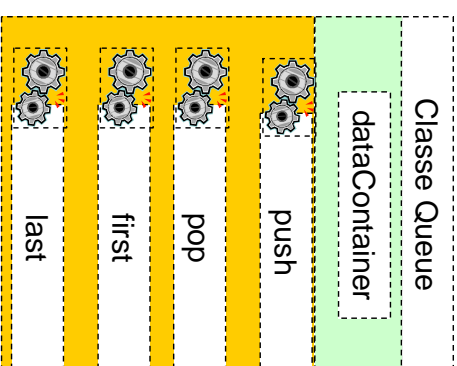
- Si usano in tipicamente nei compilatori e in vari problemi in informatica in cui l'ultimo elemento inserito deve estratto per primo

Esempio

- Un sistema operativo è in grado di gestire processi a priorità diversa (ad es. da priorità 1 a priorità 10). Se mentre si sta eseguendo un processo a ne arriva uno a priorità più alta, quello in esecuzione viene inserito in una pila e si passa all'esecuzione di quello a priorità più alta. Eventualmente successivamente ne può arrivare uno a priorità ancora per cui quelle precedentemente subentrato viene messo in testa alla pila. In seguito, quando il processo a priorità maggiore sarà terminato, gli processi verranno ripresi in ordine di priorità, cioè inverso a come sono stati fermati.

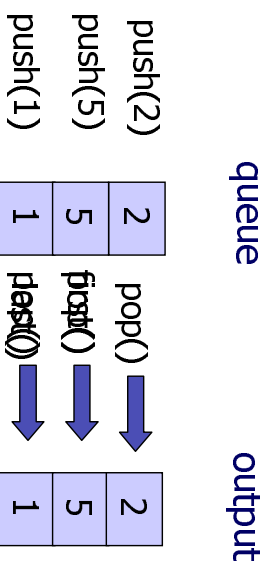
La coda (queue)

- La coda simula un contenitore in cui gli elementi si trovano in fila, uno dietro l'altro
- La coda ha metodi simili alla pila
 - push**: inserisce un elemento nella coda
 - pop**: estrae un elemento dalla coda
 - first**: restituisce l'elemento in testa alla coda senza rimuoverlo
 - last**: restituisce l'elemento in coda alla coda senza rimuoverlo
- La coda implementa una strategia **FIFO** (First In, First Output)



La coda: un esempio

```
Queue myQueue=new Queue(100);
myQueue.push(1);
myQueue.push(5);
System.out.print(myQueue.first());
System.out.print(myQueue.last());
System.out.print(myQueue.pop());
myQueue.push(2);
System.out.print(myQueue.pop());
System.out.print(myQueue.pop());
```

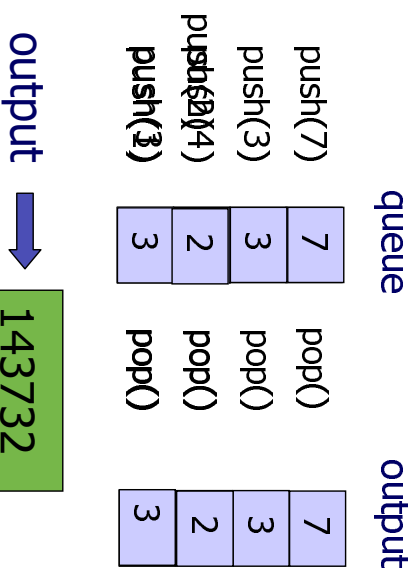


output → 15152

La coda circolare.... come funziona

- Per poter memorizzare gli elementi una coda in array si usa un'implementazione circolare
- Si supponga di avere una coda di 4 elementi

```
myQueue.push(1);
myQueue.push(4);
myQueue.push(3);
System.out.println(myQueue.pop());
myQueue.push(7);
System.out.println(myQueue.pop());
myQueue.push(2);
System.out.println(myQueue.pop());
System.out.println(myQueue.pop());
System.out.println(myQueue.pop());
System.out.println(myQueue.pop());
```



Implementazione di una coda

- Per implementare una coda si utilizza un array che funge da contenitore
- Si utilizzano inoltre due indici, che indicano dove si trovano il primo elemento che è stato inserito nella coda e l'ultimo elemento
- L'implementazioni di first() e last() sono immediate

```
class Queue{
    private int dataContainer[];
    private int first, last;

    Queue(int length) {
        dataContainer=new int[length];
        first=0;
        last=length-1;
    }
    public int first() {
        return dataContainer[first];
    }
    public int last() {
        return dataContainer[last];
    }
}
// continua nella prossima slide
```

Implementazione di una coda:

pop() e push()

- Per implementare push() e pop(), si usa la coda in maniera circolare
 - Quando first arrivano alla testa dell'array si riparte dal fondo ...
 - Quando last arriva al fondo dell'array si riparte dall'inizio...

```
// continua dalla slide precedente
...
public void push(int in) {
    last++;
    if (last==dataContainer.length){
        last=0;
    }
    dataContainer[last]=in;
}

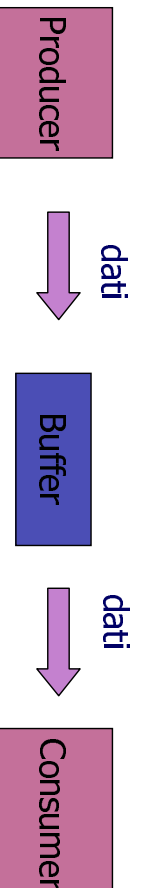
public int pop() {
    int result=dataContainer[first];
    first++;
    if (first==dataContainer.length){
        first=0;
    }
    return result;
}
}
```

Quando si usano le code?

- Si usano, ad esempio, per realizzare buffer o canali di comunicazione

Esempio

- Due servizi, due processi, due periferiche, due risorse comunicano fra loro. Un servizio (producer) produce dei dati che vengono ricevuti e elaborati dall'altro (consumer). Poiché i due servizi possono viaggiare a velocità diverse occorre avere un buffer (implementato con una coda) che contenga temporaneamente i dati se il consumer non riesce ad elaborarli in tempo.



I vettori (array)

- I vettori sono **strutture dati predefinite** in Java
- I vettori permettono di implementare altre strutture dati: ad esempio, pile e code
- I vettori permettono di implementare anche insiemi in cui si inserisce e si rimuove gli elementi in qualsiasi ordine:
 - se agli elementi da inserire è possibile associare un numero unico (allocazione per chiave, ad accesso diretto), allora si mette l'elemento nella posizione corrispondente all'indice
 - altrimenti si può mettere ogni oggetto in un posto qualsiasi (allocazione sequenziale)

La biblioteca: allocazione per chiave

- Un libro è definito da un titolo, gli autori e un numero di catalogo. Il numero di catalogo è un numero assegnato ai libri man mano che arrivano in biblioteca
- Per l'implementazione si usa un array di libri. In posizione n si mette il libro avente il numero di catalogo (chiave) n.

```
class Libro{
    String titolo, autori;
    int num;

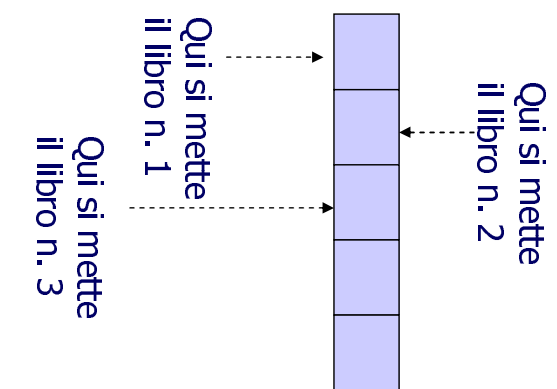
    Libro(String t, String a, int n){
        titolo=t;
        autori=a;
        num=n;
    }
}
```

```
class biblioteca{
    private libro insiemelibri[];

    public biblioteca(int n){
        insiemelibri= new libro[n];
    }
    // continua nella slide successiva
}
```

Allocazione per chiave II

- Poiché ogni libro può occupare solo una posizione dell'array, la rimozione e l'inserimento sono banali



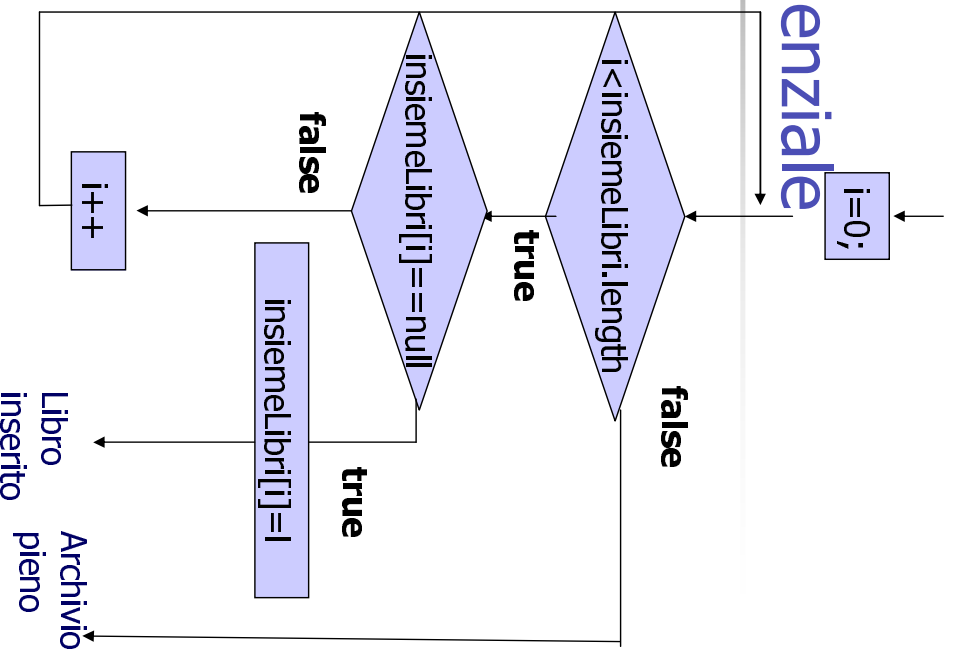
```
// continua dalla slide precedente

public void insert(Libro l){
    insiemLibri[l.num]=l;
}

public void remove(int num){
    insiemLibri[num]=null;
}
}
```

Allocazione sequenziale

- In una seconda implementazione della biblioteca ogni libro può assumere un qualsiasi posto
- Per l'inserimento si cerca la prima posizione libera (in cui c'è null) e ci si inserisce il libro



Allocazione sequenziale II

// i costruttori e le variabili sono gli stessi di prima...

```
public void insert(Libro l){  
    int i;  
    for(i=0;i<insiemelibri.length;i++){  
        if(insiemelibri[i]==null) {  
            insiemelibri[i]=l;  
            break;  
        }  
    }  
    if(i==insiemelibri.length) System.out.println("Archivio pieno...");  
}
```

// continua nella slide successiva

Allocazione sequenziale III

Per la rimozione occorre cercare l'elemento da eliminare

// continua dalla slide precedente

```
public void remove(Libro l){  
    int i;  
    for(i=0;i<insiemelibri.length;i++){  
        if(insiemelibri[i].equals(l)) {  
            insiemelibri[i]=null;  
            break;  
        }  
    }  
    if(i==insiemelibri.length)  
        System.out.println("Libro non trovato...");  
}
```

metodo che
controlla se
due oggetti
sono uguali

Il metodo equals

- Il metodo equals serve a definire quando due oggetti sono uguali
 - per oggetti generici (fanno eccezione le stringhe e pochi altri oggetti predefiniti) due oggetti sono uguali solo se hanno lo stesso handler
 - l'utente può ridefinire il concetto di uguaglianza ridefinendo esplicitamente il metodo equals

```
class Libro{
    String titolo, autori;
    int num;

    Libro(String t, String a, int n){
        titolo=t;
        autori=a;
        num=n;
    }

    boolean equals(Libro l){
        return (titolo.equals(l.titolo)
            && autori.equals(l.autori)
            && num==l.num);
    }
}
```

Allocazione sequenziale: un trucco

- Nell'implementazione precedente, quando si rimuove un elemento si può lasciare un buco fra posizioni occupate. Per questo motivo, nell'inserimento occorre scorrere tutto l'array per vedere se si trova un posto vuoto.
- In alternativa, si può tenere traccia dell'ultima posizione occupata e fare in modo che non rimangano posizioni vuote fra quelle occupate.
 - In fase di rimozione si sposta l'oggetto nell'ultima posizione al posto di quello occupato
 - In fase di inserimento si inserisce sempre dopo l'ultima posizione occupata

Allocazione sequenziale: un trucco I

```
// i costruttori e le variabili sono gli stessi di prima...
int head;

public void insert(Libro l){
    if(l!=insiemelibri.length){
        insiemelibri[head]=l;
        head++; }
    else
        System.out.println("Archivio pieno...");
}
```

// continua nella slide successiva

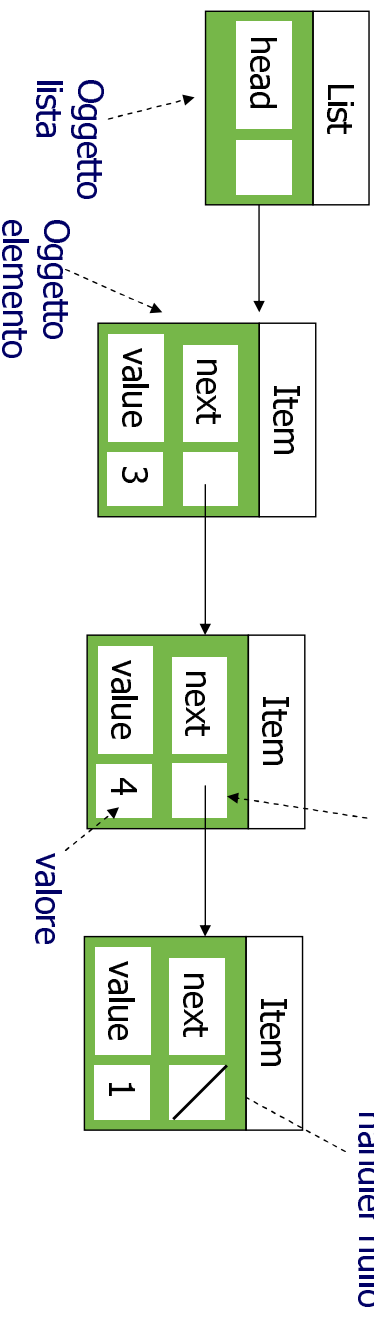
Allocazione sequenziale: un trucco II

```
// continua dalla slide precedente
public void remove( int num){
    int i;
    for(i=0;i<head;i++){
        if(insiemelibri[i].equals(l)) {
            insiemelibri[i]=insiemelibri[head];
            head--;
            break;
        }
    }
    if(i==insiemelibri.length)
        System.out.println("Libro non trovato...");
}
```

Le liste

- La lista è una sequenza di elementi concatenati
- Ogni elemento contiene
 - Il valore del contenuto
 - Il puntatore (handler) all'elemento successivo

Esempio: una lista di 3 interi



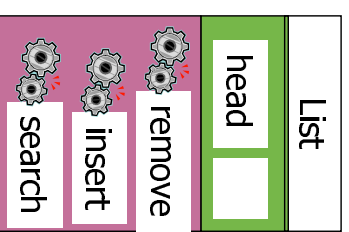
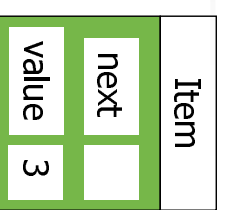
Implementazione di una lista

Oggetti e liste

- Ogni elemento della lista viene implementato con un oggetto
- L'oggetto elemento contiene il valore e (l'handler al) successivo
- Anche la lista è un oggetto che contiene (l'handler del) primo elemento

```
class Item{
    int value;
    Item next;
    Item(int v, Item n){
        value=v;
        next=n;}
}

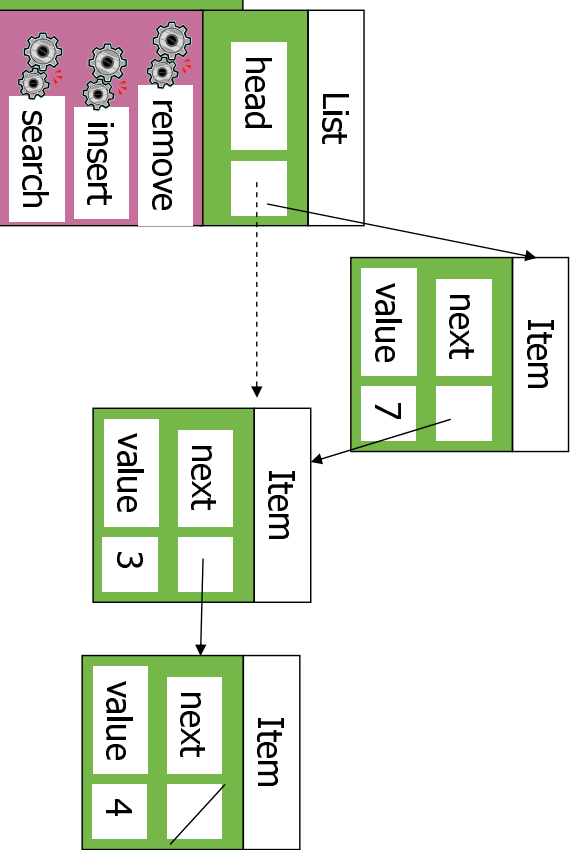
class List{
    Item head;
    void insert(int v){
        ....
    }
    ....
}
```



Implementazione di una lista: inserimento

- L'inserimento avviene aggiungendo l'elemento (ad esempio in test) con opportune modifiche degli handler

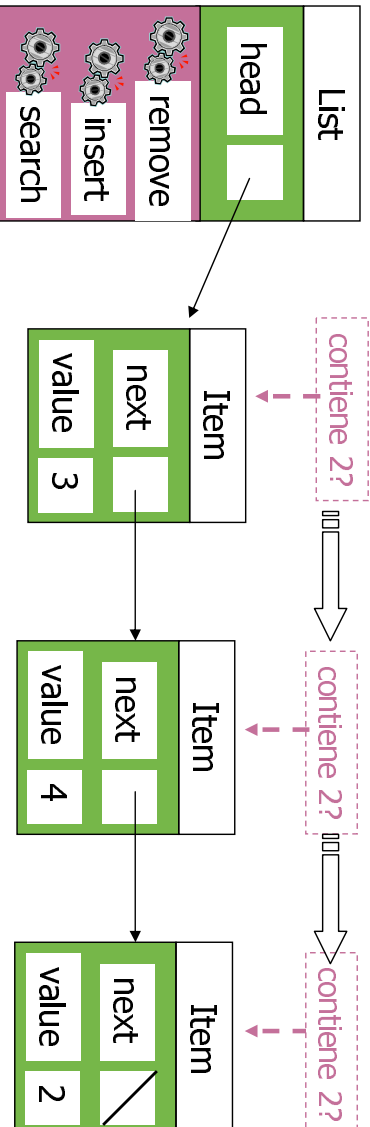
```
class List{  
private Item head;  
public void insert(int v){  
head = new Item(v,head)  
}  
// continua nelle slide successive
```



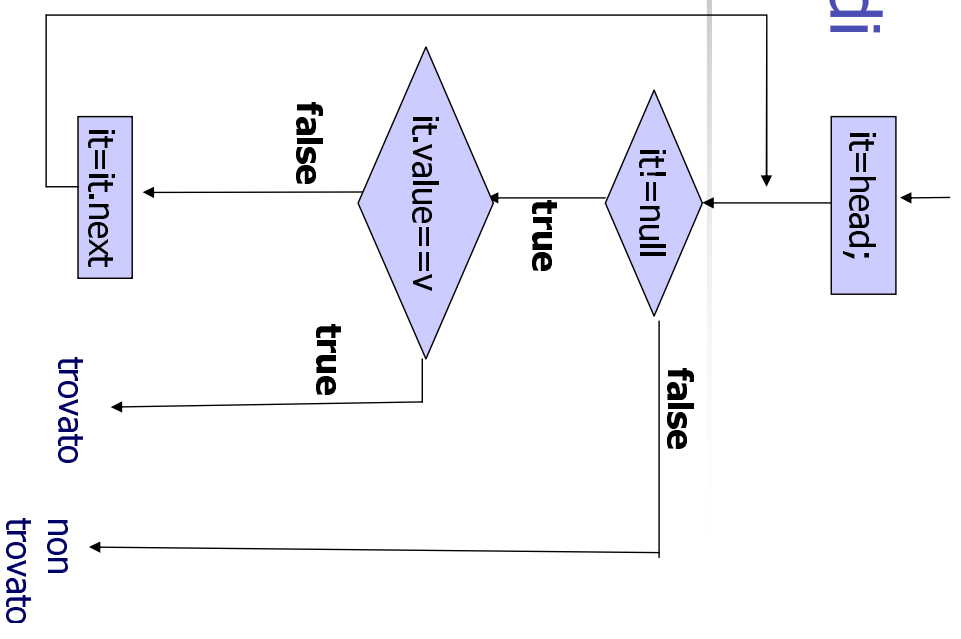
Implementazione di una lista: ricerca

- Per cercare un valore occorre scorrere tutta la lista partendo da head
- Se si trova l'elemento che contiene il valore si restituisce l'handler altrimenti si restituisce il valore null

Esempio..... cercando 2



Implementazione di una lista: ricerca

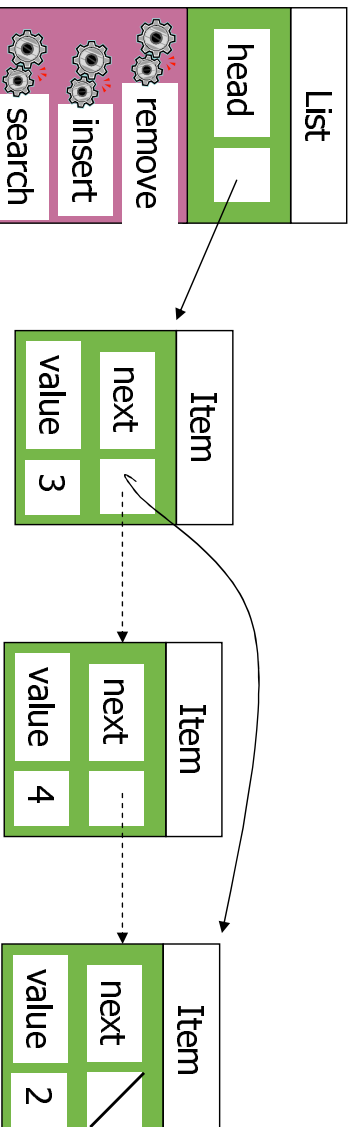


```
//continua dalla precedente slide  
public Item search(int v){  
    Item it;  
    for(it=head;it!=null;it=it.next){  
        if(it.value==v) break;  
    }  
    return it;  
}  
//continua nelle slide successive
```

Implementazione di una lista: rimozione

- Per rimuovere un elemento occorre prima scorrere tutta la lista per individuarlo, poi si elimina aggiornando gli handler
- Occorre mantenere due handler mentre si scorre la lista

Rimozione di 4



Implementazione di una lista: rimozione

```
//continua dalle slide precedenti

public void remove(int v){
    if (head != null){
        if(head.value==v) {head=head.next;}
        else {Item it;
              for(it=head;it.next != null;it=it.next){
                  if(it.next.value==v){
                      it.next=it.next.next;
                      break;}
              }
        }
    }
}
```

Se head è nullo
la ricerca è subito terminata

Rimozione nel caso in cui
l'elemento
da rimuovere è il primo

Ricerca nella lista

Rimozione

Osservazioni

- Vettori e liste permettono di implementare insieme in cui non si conosce l'ordine con cui saranno inseriti/eliminati gli elementi: ad esempio, una biblioteca, una videoteca, una rubrica ...
- **Vettori rispetto alle liste: velocità di accesso**
 - Nelle liste per accedere ad un elemento occorre scorrere la lista fino a quando non si trova l'elemento desiderato
 - con vettori basati su chiave (allocazione per chiave), si può accedere direttamente all'elemento che si desidera
 - Con i vettori basati su allocazione sequenziale occorre scorrere tutto il vettore per accedere un elemento



Osservazioni II

- Vettori rispetto alle liste: occupazione di memoria
 - Con i vettori occorre conoscere fin dall'inizio il numero (massimo) di elementi da memorizzare: si spreca spazio e/o si rischia che lo spazio non basti
 - Con le liste non serve sapere quanti elementi occorre allocare, ma nelle liste occorre memorizzare anche gli handler che possono occupare un po' di memoria in più.
- A cosa serve il metodo ricerca per valore che abbiamo visto nelle liste ?
 - Ogni elemento della lista potrebbe contenere un oggetto complesso di cui il valore è la chiave
 - Ad esempio, una lista potrebbe contenere degli studenti nella quale si ricerca uno studente per numero di matricola



Alberi

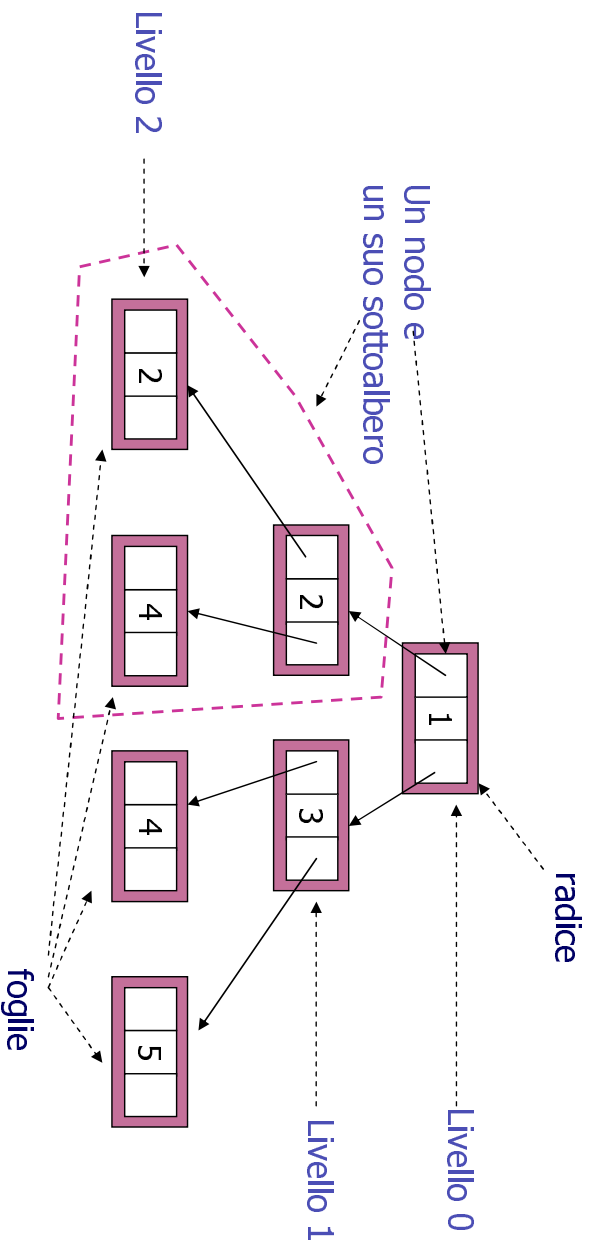
- Gli alberi sono una struttura dati fatta di nodi: ogni nodo contiene
 - un valore
 - gli handler ad alcuni sottoalberi (figli)

Definizioni

- Gli alberi si dicono **n-ari** se hanno n figli
 - Una classe importante di alberi sono quelli **binari** (due figli)
- Esiste un solo nodo (detto **radice**) da cui si può raggiungere tutti gli altri
- I nodi senza figli sono detti **foglie**
- I **livelli di un albero (profondità)** sono il numero di nodi compresi fra la radice e una delle foglie
- Un albero si dice **bilanciato** se ogni foglia è allo stesso livello.

Alberi II

Un albero binario bilanciato



Implementazione degli alberi binari

- L'implementazione degli alberi binari usa un oggetto **nodo** che contiene il **valore** e i due **sottoalberi** destro e sinistro
- Gli alberi mettono di solito a disposizione dei metodi per **inserire**, **rimuovere** ed, eventualmente, **cercare un valore**

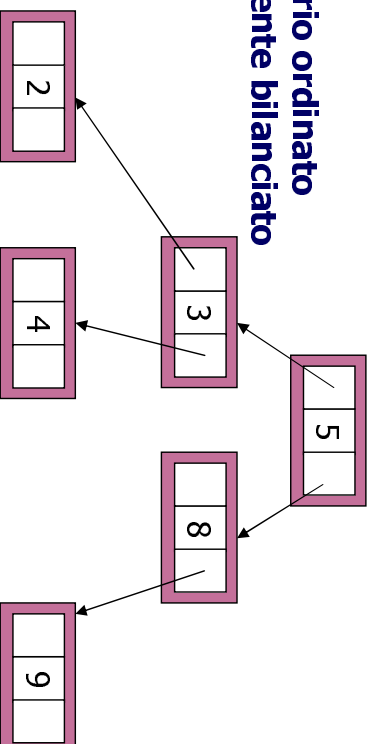
```
class Node{
    int value;
    Node left, right;
    Node(int v, Node l, Node r){
        value=v;
        left=l;
        right=r;}
}
```

```
class Tree{
    Node root;
    Tree(Node root, Tree left, Tree right){
        this.root= root;
        root.left= left;
        root.right= right;
    }
    // continua nella slide successiva
}
```

Alberi binari ordinati

- Un albero binario è **ordinato** se per ogni nodo n e il suo valore V_n
 - Il sottoalbero sinistro contiene solo valori minori di V_n
 - Il sottoalbero destro contiene solo valori maggiori di V_n

Un albero binario ordinato
non perfettamente bilanciato

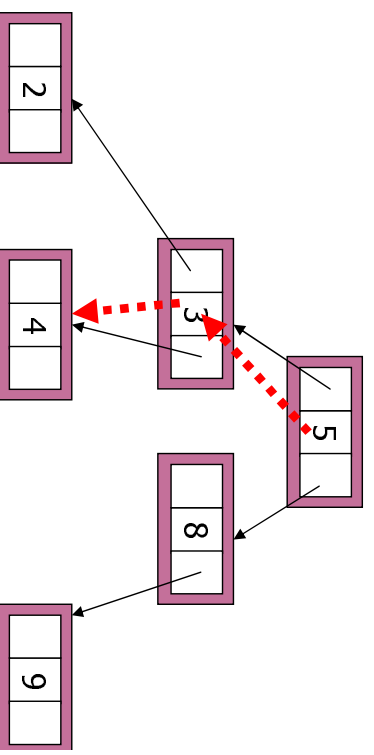


Alberi ordinati

- Gli alberi ordinati sono molto importanti e usati in informatica per realizzare insiemi di grandi dimensioni
- Per accedere ad un elemento non occorre visitare tutto l'albero, ma è sufficiente scendere in un ramo dell'albero partendo dalla radice

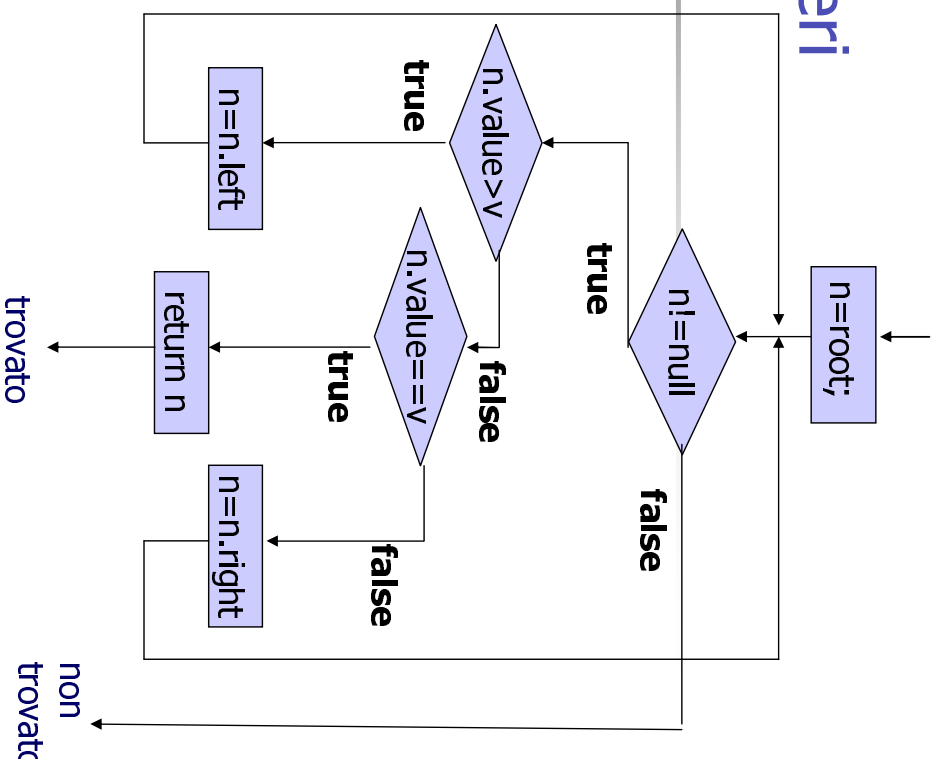
Per cercare 4

- si parte dalla radice
- si confronta 4 con 5: si va a sinistra
- si confronta 4 con 3: si va a destra
- si confronta 4 con 4: trovato!



Ricerca in alberi binari ordinati

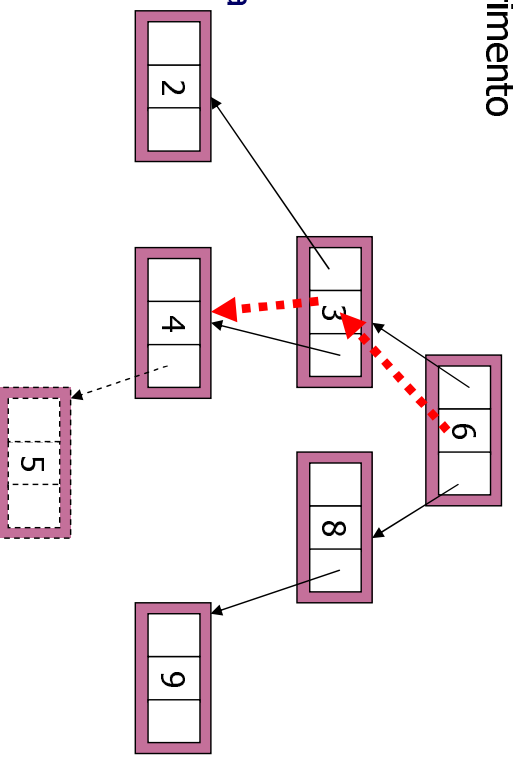
```
Node search(int v){
Node n=root;
while(n!=null){
if(v<n.value){
n=n.left;}
else {
if(n.value==v){
return n;}
else {
n=n.right;}
}
}
return null;
}
```



Inserimento in alberi ordinati

- Ogni nuovo elemento viene inserito in una foglia o in nodo parzialmente pieno
- La strategia ricorda la ricerca: prima si cerca la foglia dove inserire un elemento, poi si effettua l'inserimento

- Per inserire 5
- si parte dalla radice
 - si confronta 5 con 6: si va a sinistra
 - si confronta 5 con 3: si va a destra
 - si confronta 5 con 4:



Inserimento in alberi binari ordinati

```
void insert(int v){
    Node n=root;
    while(true){
        if(n.value > v) {
            if(n.left==null){
                n.left=new Node(v);
                break;}
            else {n.left;}
        }
        else {
            if(n.right==null){
                n.right=new Node(v);
                break;}
            else {n.right;}
        }
    }
}
```

ciclo infinito: si esce con il break
a seconda del risultato del confronto si considerai figlio sinistro o quello destro
se non esiste il figlio sinistro, si inserisce il nuovo nodo...
se esiste il figlio sinistro, si va a sinistra
se non esiste il figlio destro, si inserisce il nuovo nodo...
se esiste il figlio destro, si va a destra

Importanti proprietà degli alberi

Alberi bilanciati

- Un albero n-ario bilanciato di con l livelli contiene
 - n^l foglie e

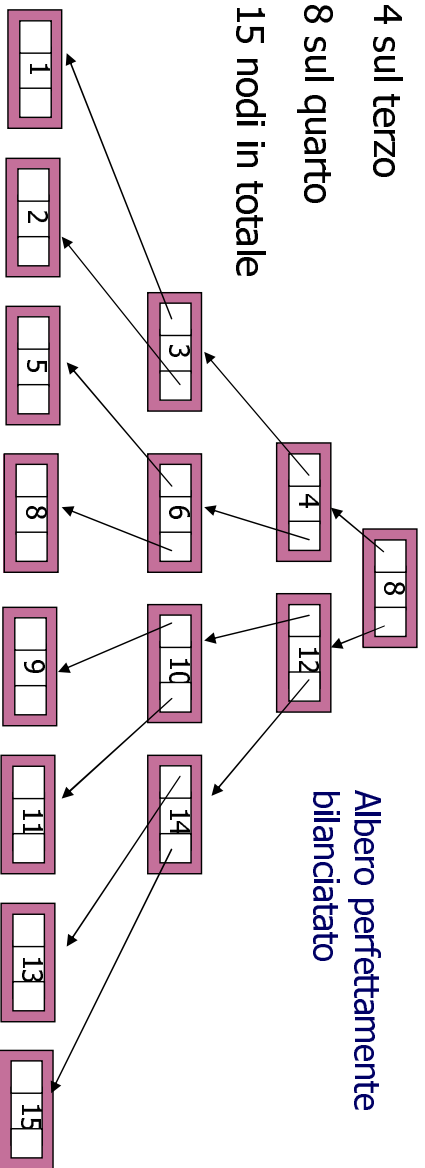
- $$\frac{n^{l+1}-1}{n-1} = \sum_{i=0}^l n^i \text{ nodi}$$

- Vicversa, si può calcolare il numero dei livelli l a partire dal numero di foglie f : $l = \log_n f$
- Inoltre, se N è il numero d nodi, allora $l = \log(N(N-1)+1) - 1 \approx \log(N)$

Importanti proprietà degli alberi

Eempio: un albero binario con 3 livelli

- 1 nodo sul livello 0
- 2 nodi sul secondo
- 4 sul terzo
- 8 sul quarto
- 15 nodi in totale

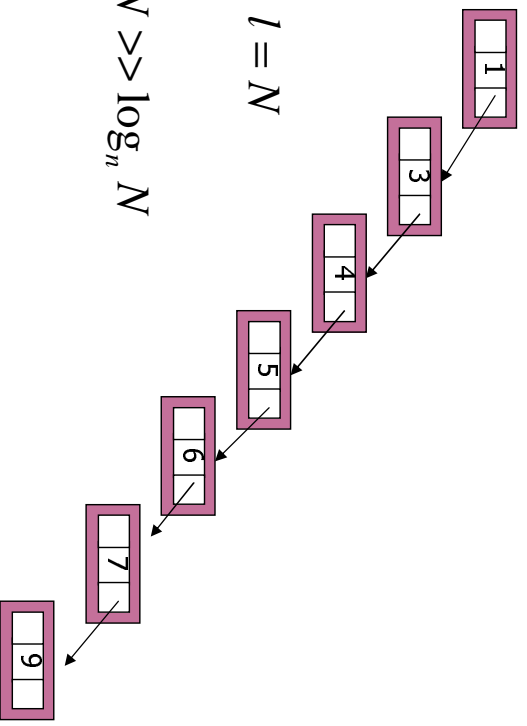


Importanti proprietà degli alberi

Alberi sbilanciati

- Un albero n-ario sbilanciato al massimo contiene con l livelli contiene
 - una foglia e
 - l nodi

- Se N è il numero d nodi, allora $l = N$
- Si osservi che se N è grande, $N \gg \log_n N$





Bilanciamento e tempi di accesso

- Il bilanciamento è fondamentale negli alberi ordinati perché i tempi di ricerca sono proporzionali alla profondità
 - La ricerca in un albero binario perfettamente bilanciato di n elementi costa circa $\log_2 n$ passi
 - In un completamente sbilanciato di costa n passi
- L'effetto bilanciamento si avverte maggiormente con molti elementi
- Esempio:
 - un albero con un miliardo (circa 2^{30}) elementi
 - Perfettamente bilanciato: profondità 30
 - Completamente sbilanciato: profondità 2^{30}



Oltre gli alberi binari... e fondamenti di informatica

- Esistono algoritmi che permettono di tenere sempre bilanciati gli alberi ordinati
- Si usano anche alberi n-ari ordinati: ad esempio i B-Tree
 - I B-tree vengono usati nei database per creare indici e accedere velocemente ai dati
 - Nei B-tree n può essere anche molto grande per cui l'albero risulta essere molto poco profondo anche con un grande numero di elementi
- Esempio: un B-tree che realizza un albero con arietà 500
 - Se la profondità è 5, le sole foglie sono $500^5 = 31\ 250$ miliardi!!

Oltre gli alberi binari... e fondamenti di informatica

I B-tree

- In un B-tree ogni nodo ha n figli e $n-1$ valori
- Il sottoalbero i -esimo contiene i valori compresi fra il valore i -esimo e $i+1$ -esimo del nodo
- L'arietà n viene scelta in modo che ogni nodo stia esattamente in settore del disco: per prendere un nodo dobbiamo fare un accesso
- Un B-tree può avere fino a metà delle celle vuote, ma è sempre perfettamente bilanciato

