Strutture dati elementari

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2022/2023

Introduzione

- Struttura dati:
  - Definisce come i dati sono logicamente organizzati
  - Definisce le operazioni per accedere e modificare i dati
- Descrive come i dati sono organizzati non quali dati sono memorizzati
  - Esempio: la struttura lista può contenere interi oppure stringhe
- Vedremo quattro tipologie di strutture dati elementari
  - Liste concatenate (Linked List)
  - Pile (Stack)
  - Code (Queue)
  - Alberi (Tree)

Preliminari: prototipo vs implementazione

- Prototipo
  - Descrive possibili valori ed operazioni di una struttura dati
  - Nasconde i dettagli implementativi
  - Permette al programmatore di implementare la struttura dati
  - Permette all'utente di capire come usare la struttura dati
- Implementazione
  - Realizzazione di una struttura dati con qualche linguaggio di programmazione
  - Non visibile all'utente
  - Può avere un forte impatto sui tempi di esecuzione

Preliminari: classi di strutture dati

- Alcune classi di strutture dati
  - Lineari: dati in ordine sequenziale (primo elemento, secondo, ...)
  - Non-lineari: nessun ordine sequenziale

  - Statiche: numero di elementi costante
  - Dinamiche: il numero di elementi può variare dinamicamente

  - Omogenee: un solo tipo di dato memorizzabile (numeri, stringhe, ..)
  - Eterogenee: differenti tipi di dato memorizzabili

Esempio: struttura dati Dizionario

- Struttura dati generica per memorizzare oggetti
  - Contiene un insieme di chiavi univoche
  - Ogni chiave è associata ad un valore
  - I valori posso essere duplicati, le chiavi sono uniche
- Un Dizionario è un insieme dinamico
  - Il suo contenuto può crescere, contrarsi ed essere modificato
- Operazioni basilari di un Dizionario (prototipo):
  - SEARCH(Key k): cerca l'oggetto associato alla chiave k
    - Ritorna NIL se la chiave non è nel dizionario
  - INSERT(Key k, Data d): aggiunge la coppia (k, d) al Dizionario
    - Se (k, d') è nel dizionario, sostituisce solo d' con d
  - DELETE(Key k): elmina la coppia (k, d) dal Dizionario
    - Non fa nulla se k non è nel dizionario

Java (asdlab.libreria.StruttureElem.Dizionario)

Interfaccia per la struttura dati Dizionario

```java
public interface Dizionario {

        // Aggiunge al dizionario la coppia (e,k)
        public void insert(Object e, Comparable k);


        // Rimuove dal dizionario l'elemento con chiave k
        public void delete(Comparable k);


        // Restituisce l'elemento e con chiave k
        public Object search(Comparable k);

}
```

N.B. Non possiamo imporre nell'interfaccia che le chiavi siano univoche

Dizionario: implementazione su array

- Idea: usiamo un array per memorizzare le coppie (Key,Data)
  - L'ordinamento delle chiavi nell'array è *casuale*
- search(Key k)
  - Cerca la chiave k tramite ricerca lineare sull'array
  - Ritorna i dati associati a k o NIL
- insert(Key k, Data d)
  - Verifica con ricerca lineare se k è presente nell'array
  - Se k è nell'array, sostituisce i dati
  - Altrimenti, inserisce la coppia (k, d) nella prima posizione libera
- delete(Key k)
  - Cerca k tramite ricerca lineare
  - Se k è nell'array, rimuove la coppia (k, d) dall'array
  - Sposta di una posizione a sinistra tutte le coppie dopo k

SEARCH SU ARRAY: PSEUDOCODICE

Semplice ricerca lineare su array

```
function SEARCH(DICT D, KEY k) → DATA
    i = LINSEARCH(D.A, D.size, k)
    if i ≠ −1 then
        return D.A[i].data
    else
        return NIL

function LINSEARCH(ARRAY A[1, ··· , m], INT n, KEY k) → INT
    for i = 1, ··· , n do
        if A[i].key == k then
            return i
    return −1                                  ▷ key k not found
```

- $D.A$ = array nel dizionario, $D.size$ = elementi nel dizionario
- Costo pessimo: $\Theta(n)$ (chiave k non trovata o in ultima posizione)
- Costo medio: $\Theta(n)$ (costo medio ricerca lineare)
- Costo ottimo: $O(1)$ (chiave k nella prima posizione dell'array)
- Nota: n è il numero di elementi nell'array non la sua lunghezza

INSERT SU ARRAY: PSEUDOCODICE

- INSERT su array non ordinato
  1. Verifica che la chiave non sia già nell'array
  2. Se la chiave è nell'array, sostituisce i dati
  3. Altrimenti, inserisce la coppia (k, d) in coda

```
function INSERT(DICT D, KEY k, DATA d)
    i = LINSEARCH(D.A, D.size, k)
    if i == -1 then
        D.size = D.size + 1          ▷ We add one more element
        i = D.size                   ▷ Possible overflow if i > m
    D.A[i].key = k
    D.A[i].data = d
```

- Caso pessimo/medio: $\Theta(n)$
  - Costo medio e pessimo di LINSEARCH: $\Theta(n)$
  - L'inserimento ha un costo costante
- Caso ottimo: $O(1)$
  - Caso ottimo di LINSEARCH: $O(1)$ (e.g., chiave in testa)

DELETE SU ARRAY: PSEUDOCODICE

- DELETE su array non ordinato
  1. Ricerca lineare della chiave k e, se trovata,
  2. Shift a sinistra di tutte le coppie che seguono quella con chiave k

```
function DELETE(DICT D, KEY k)
    i = LINSEARCH(D.A, D.size, k)
    if i ≠ −1 then
        LEFTSHIFT(D.A, D.size, i)
        D.size = D.size − 1

function LEFTSHIFT(ARRAY A[1, ··· , m], INT n, INT i)
    for j = i, ··· , n − 1 do
        A[j] = A[j + 1]
```

- Caso ottimo, medio e pessimo: $\Theta(n)$
  - Se LINSEARCH ritorna $-1 \Rightarrow \Theta(n)$ (chiave non trovata)
  - Se LINSEARCH ritorna $i \neq -1$, LEFTSHIFT sposta $n - i$ coppie
  - Il costo di LINSEARCH + costo di LEFTSHIFT è quindi

$$\Theta(i) + \Theta(n - i) = \Theta(n)$$

Costo delle operazioni su array non ordinato

- SEARCH(Key k)
  - Ricerca lineare su array non ordinato
  - Costo pessimo e medio: $O(n)$
- INSERT(Key k, Data d)
  - Ricerca lineare su array non ordinato + inserimento
  - Costo pessimo e medio: $O(n) + O(1) = O(n)$
- DELETE(Key k)
  - Ricerca lineare su array non ordinato + shift
  - Costo pessimo e medio: $\Theta(i) + \Theta(n - i) = \Theta(n)$

I costi sono dominati dalla ricerca lineare, sempre necessaria

Dizionario: implementazione su array ordinato

- Idea: usiamo un array per salvare le coppie (Key,Data) e lo manteniamo ordinato rispetto a Key dopo inserimento e rimozione
- SEARCH(KEY k)
  - Cerca la chiave k con ricerca binaria sull'array ordinato
  - Ritorna i dati associati a k o NIL
- INSERT(KEY k, DATA d)
  - Cerca con ricerca binaria (modificata) la posizione per k
  - Se k è nell'array, sostituisce i dati, altrimenti
    - sposta di un passo a destra tutte le coppie con chiave $> k$
    - inserisce la coppia (k, d) nello *spazio* aperto con lo shift
- DELETE(KEY k)
  - Cerca la chiave k con ricerca binaria e, se questa è presente
  - Sposta di un passo a sinistra tutte le coppie con chiave $> k$

SEARCH SU ARRAY ORDINATO: PSEUDOCODICE

Semplice ricerca binaria su array ordinato

```
function SEARCH(DICT D, KEY k) → INT
    i = BINSEARCH(D.A, D.size, k)
    if i ≠ −1 then
        return D.A[i].data
    else
        return NIL

function BINSEARCH(ARRAY A[1, ··· , m], INT n, KEY k) → INT
    i = 1, j = n
    while i ≤ j do
        M = (i + j)/2
        if A[M].key == k then return M
        else if A[M].key < k then i = M + 1
        else j = M − 1
    return −1
```

- Costo pessimo/medio: $O(\log n)$ (caso pessimo/medio di BINSEARCH)
- Costo ottimo: $O(1)$ (caso ottimo di BINSEARCH)

INSERT SU ARRAY ORDINATO: PSEUDOCODICE

- Inserimento su array ordinato
  1. Ricerca binaria della posizione in cui inserire la chiave k
  2. Se k non è presente, sposta a destra le coppie con chiave $> k$
  3. Inserisce la coppia (k, d) nello *spazio* aperto dallo shit

```
function INSERT(DICT D, KEY k, DATA d)
    i = BINSEARCHPOS(D.A, D.size, k)
    if D.A[i].key ≠ k then
        RIGHTSHIFT(D.A, D.size, i)
        D.size = D.size + 1
    D.A[i].key = k
    D.A[i].data = d

function RIGHTSHIFT(ARRAY A[1, ··· , m], INT n, INT i)
    for j = i, ··· , n + 1 do
        A[j + 1] = A[j]                          ▷ Overflow if n = m
```

BINSEARCHPOS: PSEUDOCODE

- Come cercare la posizione in cui inserire k in un array ordinato?
  - Se la chiave è già presente, ritorna la posizione della chiave
  - Altrimenti, ritorna la posizione in cui dovrebbe essere inserita

1: **function** BINSEARCHPOS(ARRAY $A[1, \cdots, m]$, INT n, KEY k) $\rightarrow$ INT
2: $\quad i = 1, j = n$
3: $\quad$ **while** $i \leq j$ **do**
4: $\qquad M = (i + j)/2$
5: $\qquad$ **if** $A[M].key == k$ **then return** M
6: $\qquad$ **else if** $A[M].key < k$ **then** $i = M + 1$
7: $\qquad$ **else** $j = M - 1$
8: $\quad$ **return** i

- L'unica differenza con BINSEARCH è a riga 8
- Il ciclo while termina quando $j < i \Rightarrow j = i - 1$
  - L'unica possibilità è che al passo precedente $i = j$
- Se $i = j$ allora $M = i = j$
  - Se $A[M] < k$ allora $i = M + 1$ è la posizione di inserimento
  - Se $A[M] > k$ allora $i = M$ è la posizione di inserimento

INSERT SU ARRAY ORDINATI: ANALISI

- Costo pessimo: $\Theta(n)$
  - Caso pessimo di BINSEARCHPOS: $\Theta(\log n)$
  - Caso pessimo di RIGHTSHIFT: $\Theta(n)$ (primo elemento rimosso)
- Costo ottimo: $O(\log n)$
  - Nel caso ottimo, la posizione di inserimento è in fondo a destra
  - Se BINSEARCHPOS esegue i passi, la posizione più a destra è $n/2^i$
  - Il costo totale è quindi
    $O(i)$ (BINSEARCHPOS) $+ O(n/2^i)$ (RIGHTSHIFT) $= O(i + n/2^i)$
    con $i \in [1, \log_2 n]$
  - Se la chiave k è in fondo all'array $\Rightarrow i = \log_2 n \Rightarrow O(\log n)$
- Costo medio: $O(n)$
  - Assumiamo che ogni posizione di inserimento sia equiprobabile
  - Costo dominato dallo shift: in media spostiamo $n/2$ elementi

DELETE SU ARRAY ORDINATO: PSEUDOCODICE

- Delete su array ordinato
  1. Ricerca binaria della chiave k e, se trovata,
  2. Shift a sinistra di una posizione di tutte le coppie con chiave $> k$

```
function DELETE(DICT D, KEY k)
    i = BINSEARCH(D.A, D.size, k)
    if i ≠ −1 then
        LEFTSHIFT(D.A, D.size, i)
        D.size = D.size − 1
```

- Costo pessimo: $\Theta(n)$
  - Caso pessimo di BINSEARCH: $\Theta(\log n)$
  - Caso pessimo di LEFTSHIFT: $\Theta(n)$ (primo elemento rimosso)
- Costo ottimo: $O(\log n)$
  - Come caso ottimo di INSERT
- Costo medio: $O(n)$
  - Se la chiave esiste, costo medio di LEFTSHIFT $\Theta(n)$
  - Se la chiave non esiste, costo medio di BINSEARCH $O(\log n)$

Conclusioni: Array ordinati vs non ordinati

- Siamo partiti con un prototipo per la struttura dati Dizionario

- Due strategie implementative differenti portano a prestazioni differenti

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

- Otteniamo qualche miglioramento con array ordinati
  - Miglioramento dovuto allo speed-up della procedura di ricerca

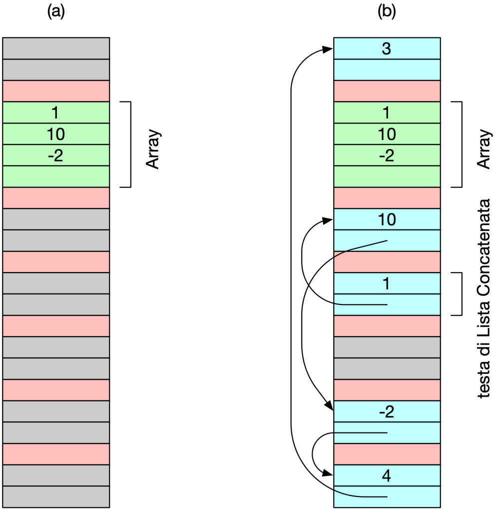
- Abbiamo comunque un dizionario a capienza limitata

- Dobbiamo inventare altre strategie per implementare dizionari che abbiano una capienza illimitata

Strutture dati elementari: Lista

- Una Lista è una struttura dati in cui tutti gli elementi sono organizzati in ordine sequenziale (primo elemento, secondo, $\cdots$)
- Una lista supporta almeno tre operazioni basilari:
  - Ricerca, Inserimento, Rimozione
- Implementazione con Array
  - L'ordine sequenziale è determinato dagli indici dell'array
  - Lo spazio per gli elementi è allocato staticamente
  - Spazio limitato ma accesso veloce agli elementi
- Implementazione con Liste concatenate (Linked Lists)
  - L'ordinamento è determinato da una catena di puntatori
  - Lo spazio per gli elementi è allocato dinamicamente su richiesta
  - Costo di accesso dipende dalla posizione ma dimensione illimitata
- Ci concentriamo su diversi tipi di Liste concatenate

Array vs Lista concatenata



(a) Non c'è abbastanza spazio libero contiguo per un array di 5 elementi

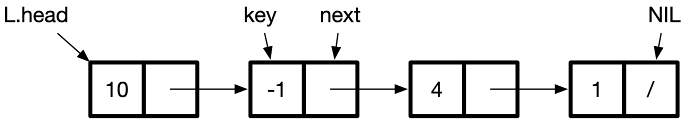
(b) Una lista concatenata non necessita di spazio contiguo

Lista concatenata semplice

- Ogni nodo x di una Lista concatenata semplice contiene
  - $x.key$: un valore chiave (non necessariamente unico)
  - $x.next$: un puntatore al nodo successivo nella lista

- Se $x.next = NIL$ allora x è l'ultimo nodo nella lista

- Un nodo può contenere altri dati oltre alla chiave (Es. $x.data$)



- Può essere visitata in un'unica direzione (dalla testa verso la coda)

- In inglese nota come Singly Linked List

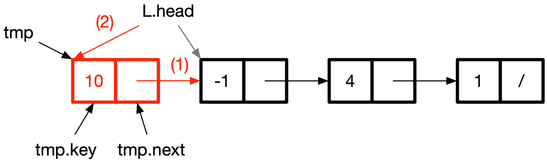
search sul Lista concatenata semplice

```
function SEARCH(SLList L, Key k) → Node
    tmp = L.head
    while tmp ≠ NIL do
        if tmp.key == k then
            return tmp
        tmp = tmp.next
    return NIL
```

- Ritorna un riferimento alla prima occorrenza della chiave k o NIL
- Se $n =$ numero di nodi nella lista
  - Costo pessimo: $\Theta(n)$ (chiave non trovata o in fondo alla lista)
  - Costo medio: $\Theta(n)$ (è una ricerca lineare)
  - Costo ottimo: $O(1)$ (chiave nel nodo in testa)

head insert su Lista concatenata semplice

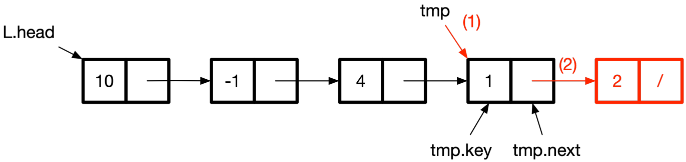
```
function HEAD_INSERT(SLList L, Key k)
    tmp = new Node(k)
    tmp.next = L.head          ▷ Update (1)
    L.head = tmp               ▷ Update (2)
```



- Inserisce un nodo in testa alla lista
  1. Crea un nuovo nodo e lo fa puntare alla testa della lista: $O(1)$
  2. Aggiorna la testa della lista: $O(1)$
- Costo ottimo, medio e pessimo: $O(1)$
  - Operazioni costanti che non dipendono dalla lunghezza della lista

TAIL INSERT SU LISTA CONCATENATA SEMPLICE

```
function TAIL_INSERT(SLList L, Key k)
    if L.head == NIL then
        L.head = NEW Node(k)
    else
        tmp = L.head
        while tmp.next ≠ NIL do
            tmp = tmp.next
        tmp.next = NEW Node(k)
```

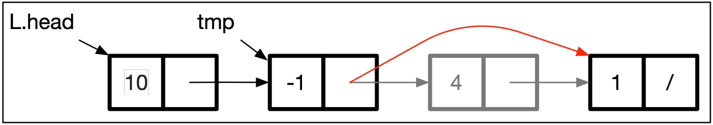


- Inserisce un nodo in coda alla lista
  1. Cerca l'ultimo nodo nella lista: $\Theta(n)$
  2. Fa puntare l'ultimo nodo al nuovo nodo: $O(1)$
- Costo ottimo, medio e pessimo: $\Theta(n)$

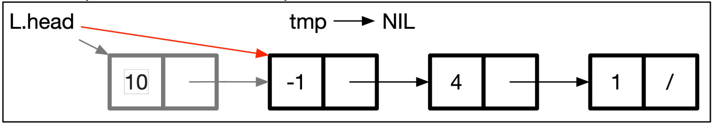
DELETE SU LISTA CONCATENATA SEMPLICE

function $\textsc{delete}(\textsc{SLList}\ L, \textsc{Key}\ k)$
    $tmp = \textsc{search\_prev}(L, k)$
    if $tmp \neq \text{NIL}$ **then**
        $tmp.next = tmp.next.next$ ▷ *Case 1*
    else if $L.head \neq \text{NIL}$ **and** $L.head.key == k$ **then**
        $L.head = L.head.next$ ▷ *Case 2 (head delete)*

Caso 1 (rimozione di nodo intermedio)



Caso 2 (rimozione della testa)



- Assumiamo un garbage collector
- Caso 1: dobbiamo cercare il nodo che precede quello da rimuovere
- Caso 2: il nodo che precede quello da rimuovere (nodo in testa)

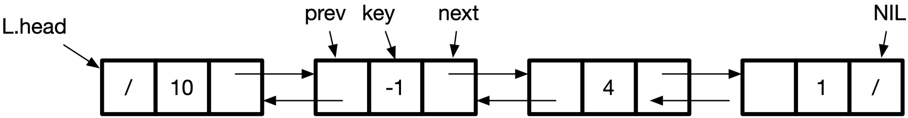
SEARCH_PREV SU LISTA CONCATENATA SEMPLICE

```
function SEARCH_PREV(SLList L, Key k)
    prev = NIL
    curr = L.head
    while curr ≠ NIL do:
        if curr.key == key then
            return prev
        prev = curr
        curr = curr.next
    return NIL
```

- Ritorna un riferimento al nodo che precede il nodo con chiave k
- Ritorna NIL se
  - non c'è un nodo con chiave k, oppure
  - il nodo in testa ha chiave k
- Il costo di DELETE dipende dal costo di SEARCH_PREV
  - Caso medio e pessimo: $\Theta(n)$ (come per ricerca lineare)
  - Caso ottimo: $O(1)$ (nodo in testa con chiave k)

VARIANTI: LISTA DOPPIAMENTE CONCATENATA

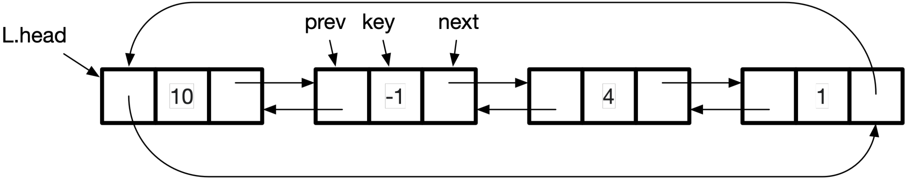
- Sono Liste concatenate semplici in cui ogni nodo contiene anche
  - $x.prev$: un puntatore al nodo precedente nella lista
- Se $x.prev = NIL$ allora x è il primo nodo nella lista



- Può essere visitata in entrambe le direzioni
- Stesso costo delle liste concatenate semplici per tutte le operazioni
- Non necessita ricerca del nodo precedente per la rimozione
  - In ogni nodo abbiamo visibilità sia in avanti che indietro
- In inglese nota come Doubly Linked List

Varianti: Lista concatenata circolare

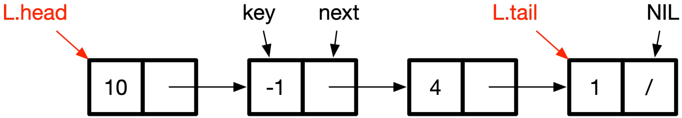
- Solo Liste doppiamente concatenate in cui
  - Il campo *next* dell'ultimo nodo punta al primo nodo
  - Il campo *prev* del primo nodo punta all'ultimo nodo



- Può essere visitata in entrambe le direzioni
- L'accesso alla testa dalla coda è veloce (cosi come il contrario)
  - Diventa più complesso visitare la lista (nessun puntatore a NIL)
- Stessi costi delle precedenti tranne che per l'inserimento in coda: $O(1)$
- In inglese nota come Circular Linked List

Varianti: Lista con puntatori a testa e coda

- Lista concatenata semplice o doppiamente concatenata in cui
  - Manteniamo puntatori per il nodo in testa e in coda



- L'accesso alla testa e alla coda è veloce
  - Più semplice da gestire rispetto alle liste circolari
- Stessi costi delle liste concatenate circolari

Liste concatenate: riassunto dei costi

Type	SEARCH	INSERT (testa)	INSERT (coda)	DELETE
Concatenata semplice	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Doppiamente concatenata	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Circolare	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Puntatori testa e coda	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Tutti i costi si riferiscono al caso medio e pessimo

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Classe per una lista concatenata circolare

```
// Implementazione basata su lista circolare
public class StrutturaCollegata implements Dizionario {
        private Record list = null;

        private final class Record { ... }

        public void insert(Object e, Comparable k)
        { ... }

        public void delete(Comparable k)
        { ... }

        public Object search(Comparable k)
        { ... }

}
```

L'implementazione di insert non controlla che la chiave sia già presente

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Classe per definire un nodo della lista

```
private final class Record {
        public Object     elem;
        public Comparable chiave;
        public Record     next;
        public Record     prev;

         public Record(Object e, Comparable k) {
                elem = e;
                chiave = k;
                next = prev = null;
        }
}
```

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Metodo per l'inserimento

```java
public void insert(Object e, Comparable k) {
        Record p = new Record(e, k);
        if (list == null)
                list = p.prev = p.next = p;
        else {
                p.next = list.next;
                list.next.prev = p;
                list.next = p;
                p.prev = list;
        }
}
```

N.B. list punta all'ultimo nodo nella Lista concatenata

Dizionario con liste concatenate

```
function SEARCH(DICT D, KEY k)
    tmp = LLSEARCH(D.LL, k)                    ▷ Search on Linked List
    if tmp == NIL then
        return NIL
    else
        return tmp.data

function INSERT(DICT D, KEY k, DATA d)
    tmp = LLSEARCH(D.LL, k)                    ▷ Search on Linked List
    if tmp ≠ NIL then
        tmp.data = d
    else
        LLINSERT(D.LL, k, d)                   ▷ Head insert on Linked List

function DELETE(DICT D, KEY k)
    LLDELETE(D.LL, k)                          ▷ Delete on Linked List
```

- *D.LL* è una struttura dati di tipo Lista concatenata
- Quale rappresentazione dovremmo scegliere?
  - Concatenata semplice, Doppiamente Concatenata, ..?

Dizionario: riassunto dei costi

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

- Nessun miglioramento rispetto agli array ordinati
  - L'unica operazione che domina i costi è comunque solo la ricerca
    - Con gli array l'operazione costosa è lo shift
  - Avrebbe senso mantenere la Lista concatenata ordinata?
- In ogni caso, con le Liste concatenata abbiamo una struttura illimitata

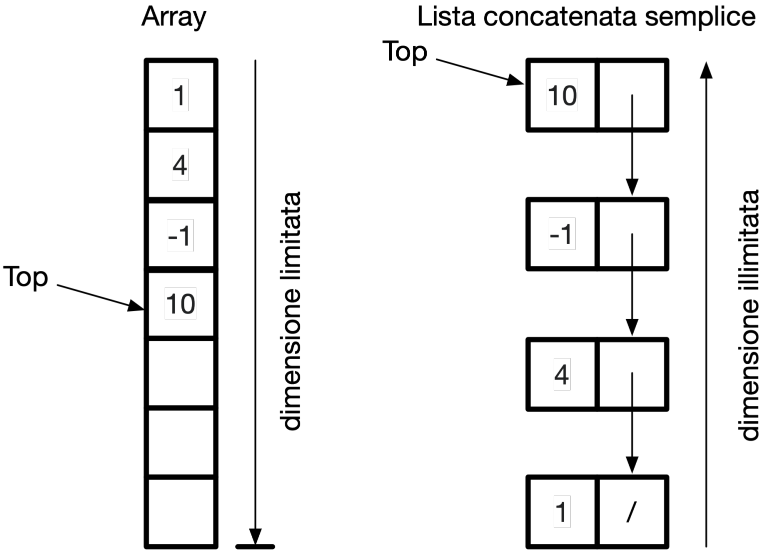
Strutture dati elementari: Pila (Stack)

- Una Pila è una struttura dati che supporta due operazioni principali
  - PUSH: aggiunge un nuovo elemento alla struttura
  - POP: rimuove l'elemento aggiunto più di recente
- Intuitivamente, una pila di elementi uno un cima all'altro
  - Ad esempio, una pila di piatti
  - Modalità LIFO (Last In First Out)
- Applicazioni delle pile
  - Gestione di record di attivazione (chiamate a funzione)
  - Linguaggi stack-oriented (PostScript, BibTex, ...)
  - Numerose applicazioni in algoritmi
    - Editor di testo (operazioni undo e redo)
    - Syntax parsing (parentesi bilanciate)
    - ...

Implementazione della struttura dati Pila

- Una Pila è una Lista che supporta un numero limitato di operazioni
- Implementazione con Liste concatenate semplici (esercizio)
  - POP: rimuove la testa della lista
  - PUSH: inserisce l'elemento in testa alla lista
  - Pro: dimensione illimitata
  - Con: piccolo overhead di memoria (valore+puntatore)
  - Domanda: perchè non usare Liste doppiamente concatenate?
- Implementazione con Array
  - POP: rimuove l'ultimo elemento nell'array
  - PUSH: inserisce l'elemento nella prima posizione libera
  - Pro: nessun overhead di memoria (memorizza solo il valore)
  - Con: dimensione limitata
- In entrambi i casi POP and PUSH costano $O(1)$

Implementazione della struttura dati Pila

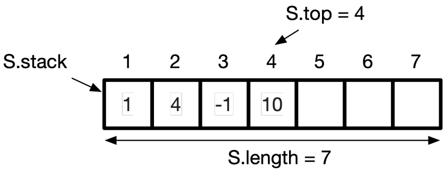


PUSH AND POP CON ARRAY STATICO

```
function PUSH(STACK S, INT x)
    if S.top == S.length then
        error "overflow"
    else
        S.top = S.top + 1
        S.stack[S.top] = x
```

```
function POP(STACK S) → INT
    if S.top == 0 then
        error "underflow"
    else
        e = S.stack[S.top]
        S.top = S.top − 1
        return e
```

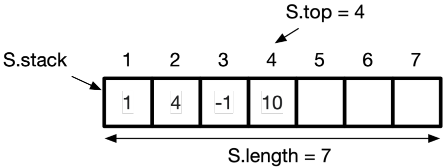
- Entrambe le funzioni costano: ?
- Stack underflow causato da un uso poco attento di POP
- Stack overflow causato da mancanza di spazio
- Come implementare una Pila con array dinamico?



```
function PUSH(STACK S, INT x)
    if S.top == S.length then
        error "overflow"
    else
        S.top = S.top + 1
        S.stack[S.top] = x
```

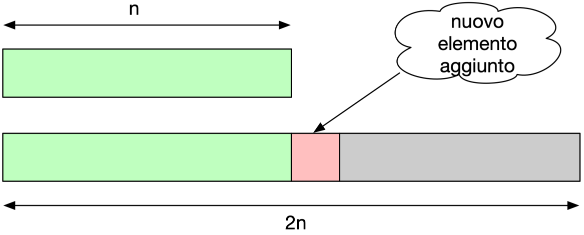
```
function POP(STACK S) → INT
    if S.top == 0 then
        error "underflow"
    else
        e = S.stack[S.top]
        S.top = S.top − 1
        return e
```

- Entrambe le funzioni costano (pessimo e ottimo) $O(1)$
- Stack underflow causato da un uso poco attento di POP
- Stack overflow causato da mancanza di spazio
- Come implementare una Pila con array dinamico?

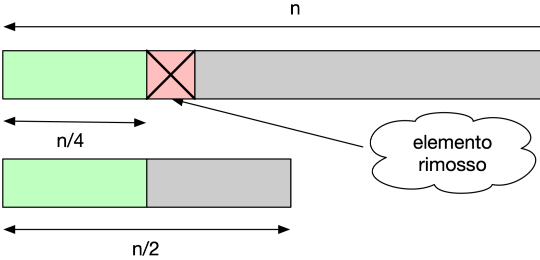


Strategia con array dinamico

- Raddoppiamo la dimensione dell'array quando non c'è spazio libero



- Dimezziamo la dimensione dell'array quando l'occupazione è di $1/4$



POP CON ARRAY DINAMICO

```
1:  function POP(STACK S) → INT
2:      if S.top == 0 then
3:          error "underflow"
4:      else
5:          e = S.stack[S.top]
6:          S.top = S.top − 1
7:          if S.top ≤ ⌊S.length/4⌋ then
8:              n = S.length
9:              LET T[1, ··· , ⌈n/2⌉] BE A NEW ARRAY
10:             for i = 1, ··· , ⌊n/4⌋ do
11:                 T[i] = S.stack[i]
12:             S.stack = T
13:             S.length = ⌈n/2⌉
14:         return e
```

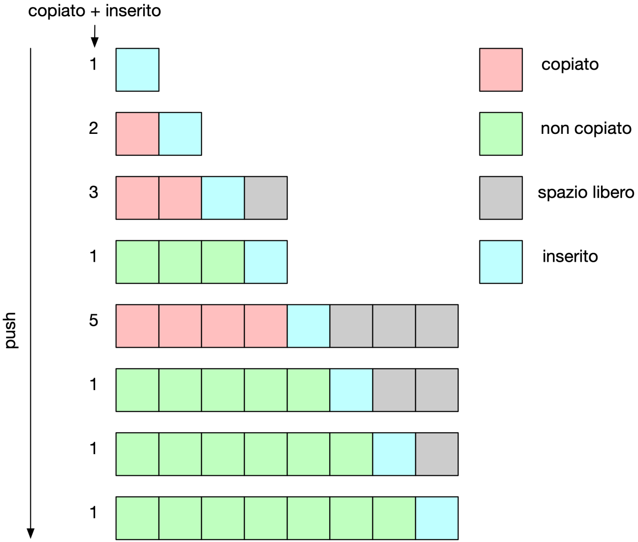
- Costo nel caso pessimo: $O(n)$ (copia dell'array, linee $10 - 11$)
- Costo nel caso ottimo: $O(1)$ (array utilizzato per più di $1/4$)

PUSH CON ARRAY DINAMICO

```
 1: function PUSH(STACK S, INT x)
 2:     if S.top == S.length then
 3:         n = S.length
 4:         LET T[1, ··· , 2n] BE A NEW ARRAY
 5:         for i = 1, ··· , n do
 6:             T[i] = S.stack[i]
 7:         S.stack = T
 8:         S.length = 2n
 9:     S.top = S.top + 1
10:     S.stack[S.top] = x
```

- Costo nel caso pessimo: $O(n)$ (copia dell'array, linee $5 - 6$)
- Costo nel caso ottimo: $O(1)$ (array non pieno)

Analisi di push



Analisi ammortizzata di PUSH

- Costo nel caso pessimo $O(n)$, costo nel caso ottimo $O(1)$

- Qual è il costo di n PUSH partendo da una Pila vuota?
  - Il costo nel caso pessimo è $O(n^2)$ se usiamo l'upper bound $O(n)$
  - $O(n^2)$ non è una stima accurata: non raddoppiamo spesso l'array
  - Il costo del'i-esima PUSH è infatti

$$c_i = \begin{cases} i & \text{se } i-1 \text{ è una potenza esatta di } 2 \\ 1 & \text{altrimenti} \end{cases}$$

  - Metodo degli aggregati: il costo totale di n PUSH è

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log_2 n} 2^j = n + \frac{2^{\log_2 n+1} - 1}{2-1} = 3n - 1 = O(n)$$

- Il costo ammortizzato di n PUSH è dunque $\dfrac{O(n)}{n} = O(1)$

Analisi ammortizzata di PUSH e POP

- Cosi come abbiamo fatto per PUSH possiamo dimostrare che il costo ammortizzato di n POP partendo da una Pila piena è $O(1)$
  - Non abbiamo una caratterizzazione semplice come per PUSH
  - Dobbiamo utilizzare il metodo degli accantonamenti
    - Costo ammortizzato: 2€ (1€ rimozione+1€ credito per copia)

- Quanto costa una generica sequenza di n PUSH e POP?
  - Di nuovo, metodo degli accantonamenti (non aggregati)
  - Possiamo dimostrare che il costo di ogni sequenza di n PUSH e POP su array dinamico inizialmente vuoto è al peggio $O(n)$
    $\Rightarrow$ costo ammortizzato di $O(1)$ per entrambe PUSH e POP

- N.B. Tali costi ammortizzati valgono solo per la nostra strategia di espansione/contrazione ma non per ogni possibile strategia
  - Ad esempio, dimezzare l'array quando è pieno per metà porta a costi ammortizzati di $O(n)$ per operazione

Java (asdlab.libreria.StruttureElem.Pila)

Interfaccia Pila

```
public interface Pila {
        // Verifica se la pila è vuota.
        public boolean isEmpty();

        // Aggiunge l'elemento in cima
        public void push(Object e);

        // Restituisce l'elemento in cima
        public Object top();

        // Cancella l'elemento in cima
        public Object pop();
}
```

Java (asdlab.libreria.StruttureElem.PilaArray)

Implementazione con array dinamico

```
public class PilaArray implements Pila {
        private Object[] S = new Object[1];
        private int n = 0;

        public boolean isEmpty()
        {...}

        public void push(Object e)
        { ... }

        public Object top()
        { ... }

        public Object pop()
        { ... }
}
```

L'array ha dimensione iniziale 1

Java (asdlab.libreria.StruttureElem.PilaArray)

Metodi push e top

```
public void push(Object e) {
        if (n == S.length) {
                Object[] temp = new Object[2 * S.length];
                for (int i = 0; i < n; i++) temp[i] = S[i];
                S = temp;
        }
        S[n] = e;
        n = n + 1;
}

public Object top() {
    if (this.isEmpty())
      throw new EccezioneStrutturaVuota("Pila vuota");
    return S[n - 1];
  }
```

Il caso ”pila vuota” viene gestito lanciando un’eccezione

Java (asdlab.libreria.StruttureElem.PilaArray)

Metodi pop e isEmpty

```java
public Object pop() {
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    n = n - 1;
    Object e = S[n];
    if (n > 1 && n == S.length / 4) {
        Object[] temp = new Object[S.length / 2];
        for (int i = 0; i < n; i++) temp[i] = S[i];
        S = temp;
    }
    return e;
}

public boolean isEmpty() {
    return n == 0;
}
```

Il caso "pila vuota" viene gestito lanciando un'eccezione

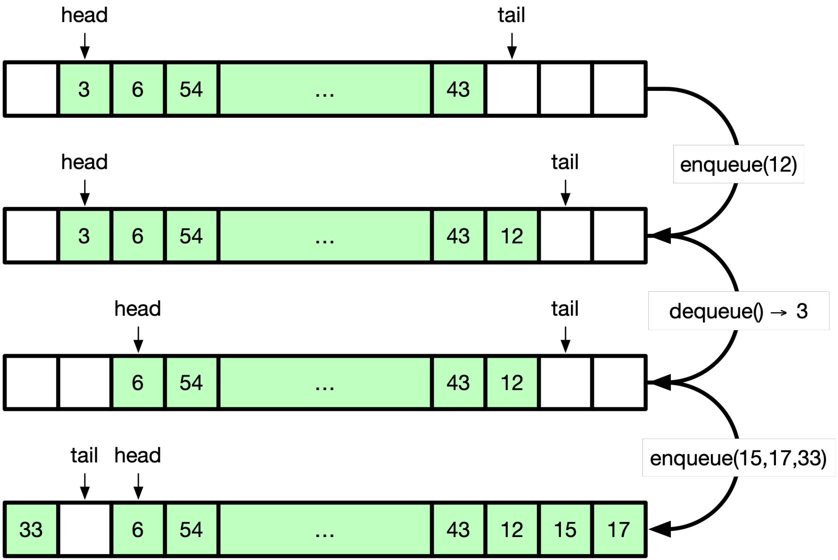
Strutture dati elementari: Coda (Queue)

- Una Coda è una struttura dati che supporta due operazioni principali
  - ENQUEUE: aggiunge un elemento in fondo alla coda
  - DEQUEUE: rimuove l'elemento in testa alla coda
- Gli elementi sono rimossi nello stesso ordine in cui sono inseriti
  - Non è possibile accedere ad elementi nel mezzo della Coda
- Intuitivamente, una coda rappresenta una fila di elementi
  - Ad esempio, una fila di persone in attesa di un servizio
  - Modalità FIFO (First In First Out)
- Applicazioni della code
  - Scheduling dei processi nei sistemi operativi
  - Visita di tipo Bread-first-search su grafi
  - ...

Implementazione della struttura dati Coda

- Una Coda è una Lista che supporta un numero limitato di operazioni
- Implementazione con Liste concatenate circolari (esercizio)
  - ENQUEUE: inserisce l'elemento in coda alla lista
  - DEQUEUE: rimuove la testa della lista
  - Pro: dimensione illimitata
  - Con: overhead di memoria (valore+2 puntatori)
- Implementazione con Liste concatenate semplici (esercizio)
  - Usiamo la versione con puntatore a testa e coda
  - ENQUEUE: inserisce in coda usando il puntatore alla coda
  - Pro e Con come per Liste concatenate circolari
- Implementazione con Array circolari
  - Pro: nessun overhead di memoria (memorizza solo il valore)
  - Con: dimensione limitata
- In tutti e tre i casi ENQUEUE and DEQUEUE costano $O(1)$

Implementazione con array circolare



Implementazione con array circolare

```
function ENQUEUE(QUEUE Q, INT x)
    if Q.size == Q.length then
        error "overflow"
    Q, buf[Q.tail] = x
    Q.tail = (Q.tail%Q.length) + 1
    Q.size = Q.size + 1
```

```
function DEQUEUE(QUEUE Q) → INT
    if Q.size == 0 then                ▷ Empty Q
        error "underflow"
    x = Q.buf[Q.head]
    Q.head = (Q.head%Q.length) + 1
    Q.size = Q.size − 1
    return x
```

- Entrambe costano (ottimo,medio,pessimo) $O(1)$
- *Q.size* è il numero di elementi in *Q*
- *tail* punta alla prima cella libera
- % = operazione modulo
- Il modulo permette una visita circolare dell'array
- Come implementare una struttura dati Coda con array dinamico circolare?

Java (asdlab.libreria.StruttureElem.Coda)

Interfaccia Coda

```
public interface Coda {
        //Verifica se la coda è vuota
        public boolean isEmpty();

        //Aggiunge l'elemento in fondo alla coda
        public void enqueue(Object e);

        //Restituisce il primo elemento della coda
        public Object first();

        //Cancella il primo elemento nella coda
        public Object dequeue();
}
```

Implementazione con liste concatenate: CodaCollegata.java

Java: coda con array circolare 1/2

Implementazione con array circolare (non in asdlab)

```java
public class CodaArrayCircolare implements Coda {
    private Object[] buffer; // Array di oggetti
    private int head;        // Dequeuing index
    private int tail;        // Enqueuing index
    private int size;        // Numero di elementi nella coda

    public CodaArrayCircolare(int max) {
      buffer = new Object[max];
      head = tail = size = 0;
    }

    @Override
    public boolean isEmpty() { return (size==0); }

    @Override
    public Object first() throws EccezioneStrutturaVuota {
      if (size == 0) throw new EccezioneStrutturaVuota("Coda vuota");
      else return buffer[head];
    }
    ...
```

Java: coda con array circolare 2/2

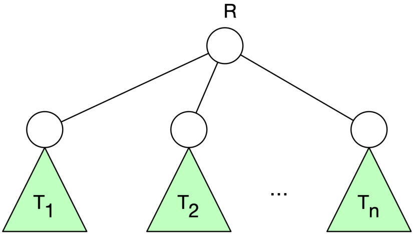
Implementazione con array circolare (non in asdlab)

```java
    @Override
    public void enqueue(Object o) {
      if (size == buffer.length)
           throw new EccezioneArrayPieno("Coda piena");
        buffer[tail] = o;
        tail =(tail+1) % buffer.length;
        size++;
    }

    @Override
    public Object dequeue() {
      if (size == 0) throw new EccezioneStrutturaVuota("Coda vuota");
        Object res = buffer[head];
        head =(head+1) % buffer.length;
        size--;
        return res;
    }
}
```

Strutture dati elementari: Alberi

- Un Albero è una struttura dati non-lineare ad albero gerarchico
- Definizione di struttura dati Albero:
  - Un insieme di nodi (or vertici)
  - Un insieme di archi che connettono nodi
  - Esiste un solo percorso per andare da un nodo all'altro
- Un Albero è ordinato se i figli di ogni nodo sono ordinati
  - Possiamo identificare il primo figlio, il secondo figlio, ...
- Un albero è radicato se uno dei suoi nodi è identificato come radice

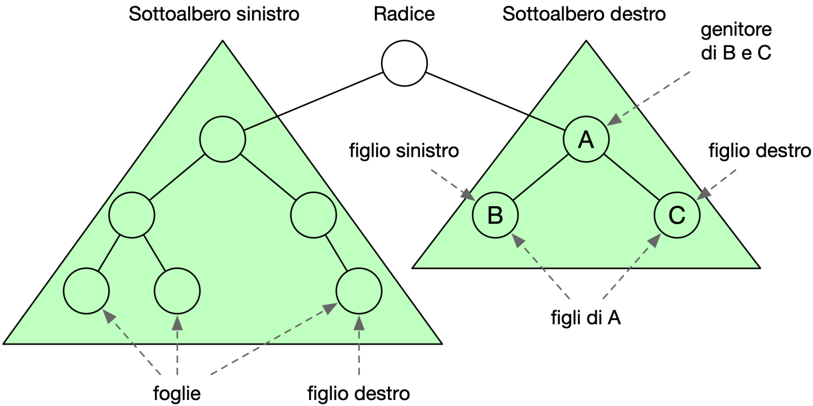


Definizione ricorsiva di Albero radicato:

- Insieme vuoto di nodi oppure
- Una radice R e zero o più alberi disgiunti (sotto-alberi) le cui radici sono connesse ad R

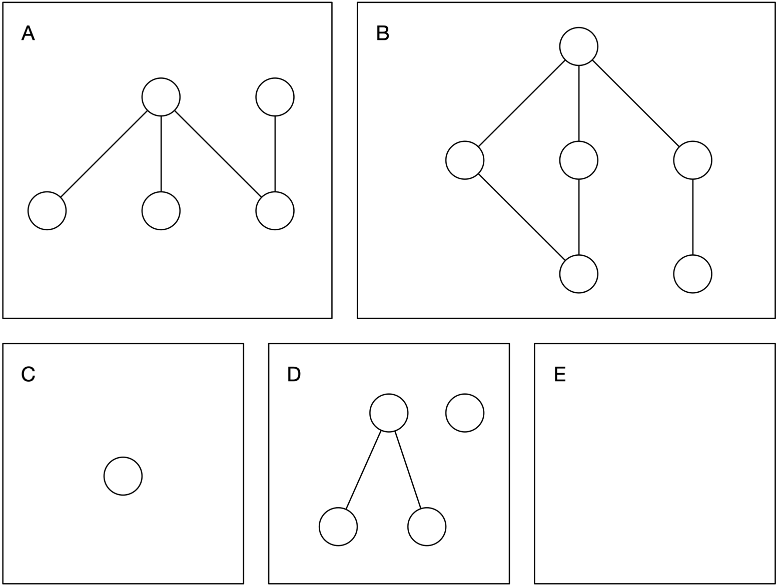
Alberi Binari

- Un Albero Binario è un Albero in cui ogni nodo ha al massimo due figli



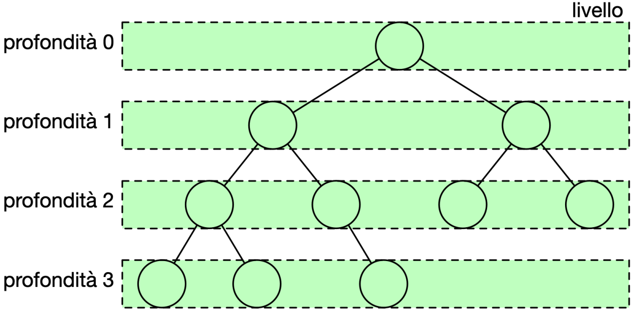
- Un Albero Binario è un Albero ordinato
  - Ogni nodo può avere un figlio sinistro e/o un figlio destro
  - Un nodo può avere figlio destro ma non sinistro (e al contrario)

Quali sono Alberi (Binari)?



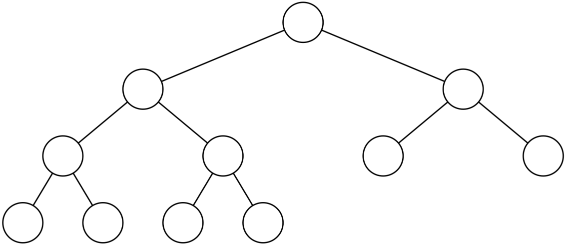
Alcune definizioni

- La profondità di un nodo u è la lunghezza del percorso (unico) che va dalla radice al nodo u (numero di archi)
- Un livello è l'insieme di tutti i nodi alla stessa profondità
- L'altezza di un Albero è la sua massima profondità
- Il grado di un nodo è il numero dei suoi figli

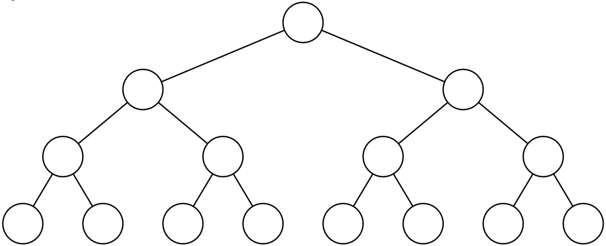


Alberi binari speciali

- Un albero binario è completo se ogni nodo intermedio ha due figli



- Un albero binario è perfetto se è completo e se tutte le foglie sono alla stessa profondità



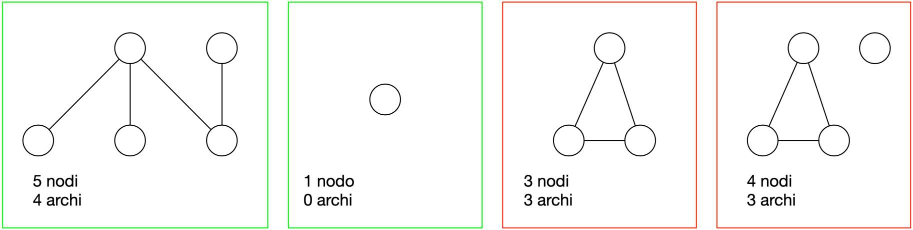
Proprietà fondamentale di un Albero

Teorema

Ogni Albero non-vuoto con n nodi ha esattamente $n - 1$ archi
(Dimostrazione per induzione)

- Possiamo usare tale proprietà per dimostrare che una struttura dati non è un Albero

- Il Teorema non funziona nella direzione opposta

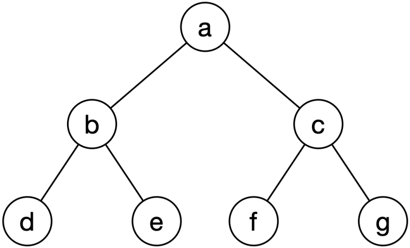


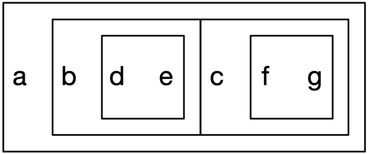
Algoritmi di visita su Alberi

- Algoritmo di visita (o anche *algoritmo di ricerca*) su Albero
  - Algoritmo per *visitare* tutti i nodi di una struttura dati Albero
- Visita in profondità o Depth-First Search (DFS)
  - La ricerca va in profondià il più possibile prima di visitare il nodo successivo nello stesso livello
  - Esistono tre varianti: pre-ordine, post-ordine, in-ordine
- Visita in ampiezza o Breadth-First Search (BFS)
  - La ricerca viene eseguita livello per livello

VISITA IN PROFONDITÀ: PRE-ORDINE (PRE-ORDER)

```
function PREORDER(NODE T)
    if T ≠ NIL then
        VISIT(T)
        PREORDER(T.left)
        PREORDER(T.right)
```

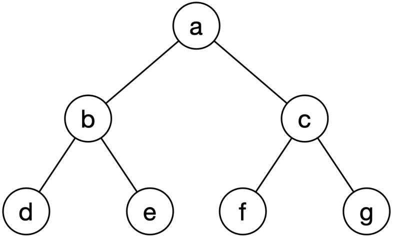


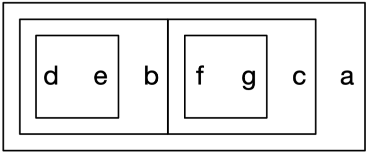


- NODE indica una struttura nodo di un albero
- Assumiamo che VISIT abbia costo $O(1)$
- Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Visita in profondità: post-ordine (post-order)

```
function POSTORDER(NODE T)
    if T ≠ NIL then
        POSTORDER(T.left)
        POSTORDER(T.right)
        VISIT(T)
```

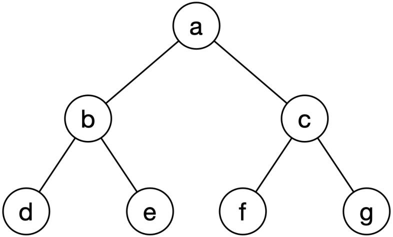


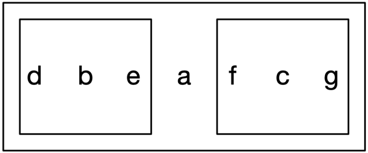


Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Visita in profondità: in-ordine (in-order)

```
function INORDER(NODE T)
    if T ≠ NIL then
        INORDER(T.left)
        VISIT(T)
        INORDER(T.right)
```

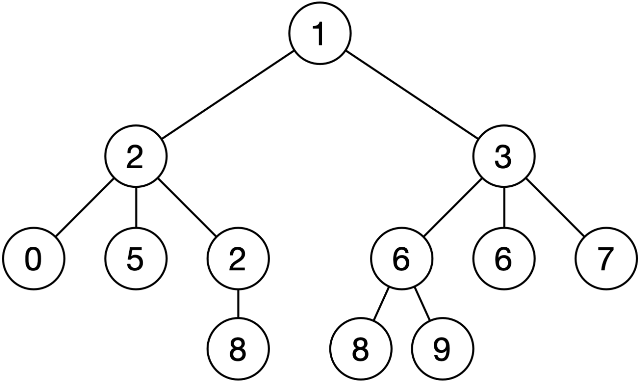




Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

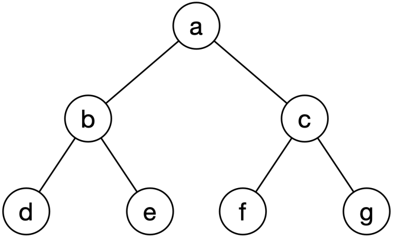
Visita in profonditàsu Alberi non-binari

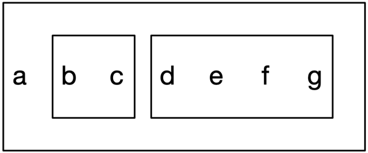
- Gli algoritmi di visita possono essere generalizzati ad Alberi non Binari
- Solo la visita in-ordine richiede specifiche aggiuntive
  - Es: visitiamo i primi *k* figli, poi nodo corrente, poi i figli rimanenti
- Qual è l'ordine di visita (pre,post,in) nel seguente Albero?



Visita in ampiezza (BFS)

```
function BFS(Tree T)
    Let Q be a new Queue
    if T.root ≠ NIL then
        enqueue(Q, T.root)
    while Q.size ≠ 0 do
        x = dequeue(Q)
        visit(x)
        if x.left ≠ NIL then
            enqueue(Q, x.left)
        if x.right ≠ NIL then
            enqueue(Q, x.right)
```



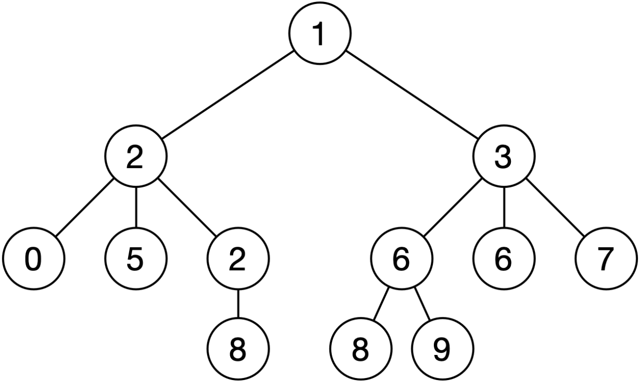


- N.B. Usiamo una Coda per imporre un ordine di visita per livello
- Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Visita in ampiezza su Alberi non-binari

- Anche la visita in ampiezza è generalizzabile ad Alberi non-binari

- Qual è l'ordine di visita in ampiezza dei nodi nel seguente Albero?



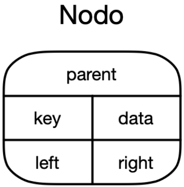
Esempio: algoritmo su Albero Binario

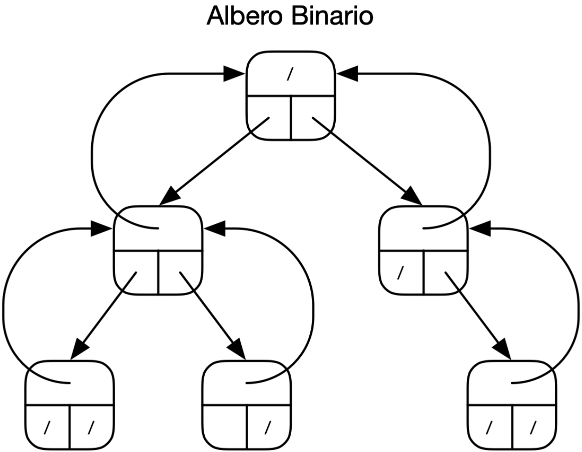
Scrivere un algoritmo per contare i nodi di un Albero Binario

```
function COUNTNODES(NODE T) → INT
    if T == NIL then
        return 0
    else
        return 1+COUNTNODES(T.left)+COUNTNODES(T.right)
```

- Quale tipo di visita è implementato in COUNTNODES?
  - post-ordine (per valutare T bisogna prima valutare i suoi figli)
- Qual è il tempo di calcolo di COUNTNODES?
  - Costo (ottimo,pessimo,medio): $\Theta(n)$
- Come modificare l'Albero Binario in modo che COUNTNODES sia $O(1)$?
  - Aggiungiamo un campo *T.totnodes* ad ogni nodo
- Qual è l'impatto di tale modifica sul costo di altre operazioni, ad esempio *aggiungi una nuova foglia*, su un Albero Binario?

Implementazione di un Albero Binario

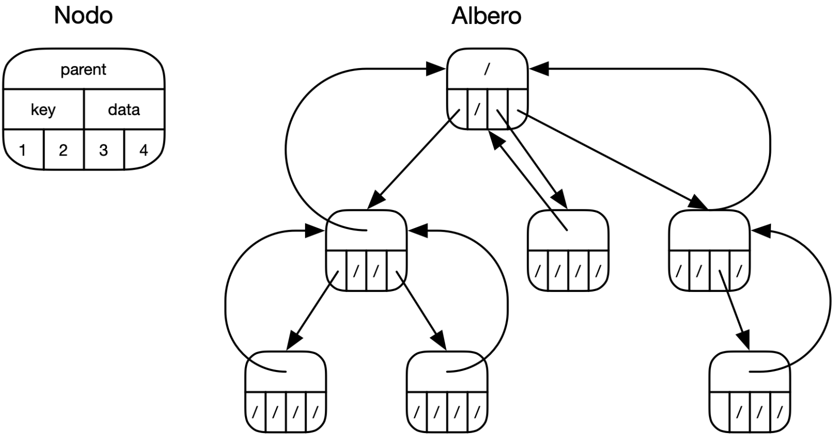




Implementazione con puntatori

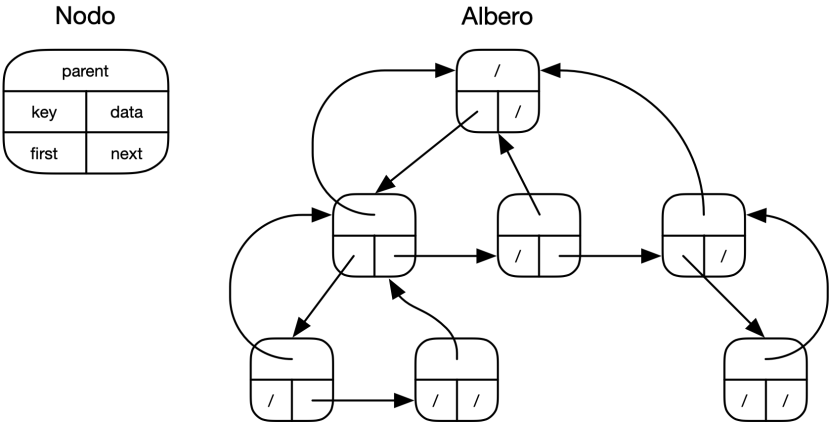
- left: puntatore al figlio sinistro
- right: puntatore al figlio destro

Implementazione di un Albero non-binario 1



- Ogni nodo contiene un array di puntatori a k figli
- Il numero massimo di k figli è fisso
- Rischio di sprecare spazio se molti nodi hanno meno di k figli

Implementazione di un Albero non-binario 2



- Ogni nodo ha un puntatore al primo (first) figlio
- Ogni nodo ha un puntatore al fratello successivo (next)
- La lista di figli è gestita con una Lista concatenata sempice

Visite in profondità su Albero non binario

Visite per la rappresentazione primo-figlio, fratello-successivo

```
// Solo ricorsiva
function PREORDER(NODE T)
    if T ≠ NIL then
        VISIT(T)
        PREORDER(T.first)
        PREORDER(T.next)
```

```
// Solo ricorsiva
function POSTORDER(NODE T)
    if T ≠ NIL then
        POSTORDER(T.first)
        POSTORDER(T.next)
        VISIT(T)
```

```
// Mista ricorsiva/iterativa
function PREORDER(NODE T)
    if T ≠ NIL then
        VISIT(T)
        tmp = T.first
        while tmp ≠ NIL do
            PREORDER(tmp)
            tmp = tmp.next
```

```
// Mista ricorsiva/iterativa
function POSTORDER(NODE T)
    if T ≠ NIL then
        tmp = T.first
        while tmp ≠ NIL do
            POSTORDER(tmp)
            tmp = tmp.next
        VISIT(T)
```

Costo (ottimo, pessimo, medio) in tutti i casi: $\Theta(n)$ (n = numero di nodi)

Visita in ampiezza su Albero non binario

Visita BFS per la rappresentazione primo-figlio, fratello-successivo

```
function BFS(Tree T)
    Let Q be a new Queue
    if T.root ≠ NIL then
        enqueue(Q, T.root)
    while Q.size ≠ 0 do
        x = dequeue(Q)
        visit(x)
        tmp = x.first
        while tmp ≠ NIL do
            enqueue(Q, tmp)
            tmp = tmp.next
```

Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Java (asdlab.libreria.Alberi.Albero)

Interfaccia Albero

```
public interface Albero {
  public int numNodi();

  public int grado(Nodo v);

  public Object info(Nodo v);

  public Nodo radice();

  public Nodo padre(Nodo v);

  public List figli(Nodo v);

  public List visitaDFS();

  public List visitaBFS();
  ...
}
```

Java (asdlab.libreria.Alberi.Nodo)

Classe astratta nodo

```
public abstract class Nodo implements Rif {

  // Il contenuto informativo associato a ciascun nodo
  public Object info;

  // Costruttore per l'istanziazione di nuovi nodi
  public Nodo(Object info) {this.info = info;}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public abstract Object contenitore();
}

// asdlab.libreria.StruttureElem.Rif
public interface Rif {
}
```

Rif è una interfaccia vuota (*marker interface*) utilizzata solo per *classificare*

Java (asdlab.libreria.Alberi.NodoBinPF)

Classe nodo per albero binario

```java
public class NodoBinPF extends Nodo {
  public NodoBinPF padre; // Padre del nodo corrente.
  public NodoBinPF sin;   // Figlio sinistro del nodo corrente.
  public NodoBinPF des;   // Figlio destro del nodo corrente.
  public AlberoBin albero; // Albero cui il nodo appartiene.

  // Costruttore per l'istanziazione di nuovi nodi.
  public NodoBinPF(Object info) {super(info);}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public AlberoBin contenitore(){
    NodoBinPF n = this;
    while (n.padre != null) n = n.padre;
    return n.albero;
  }
}
```

Java (asdlab.libreria.Alberi.NodoPFFS)

Classe nodo per albero generico Primo figlio-Fratello Successivo

```java
public class NodoBinPF extends Nodo {
  public NodoPFFS padre; // Padre del nodo corrente.
  public NodoPFFS primo; // Primo figlio del nodo corrente.
  public NodoPFFS succ; // Fratello successivo del nodo corrente.
  public Albero albero; // Albero cui il nodo appartiene.

  // Costruttore per l'istanziazione di nuovi nodi.
  public NodoPFFS(Object info) {super(info);}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public AlberoBin contenitore(){
    NodoPFFS n = this;
    while (n.padre != null) n = n.padre;
    return n.albero;
  }
}
```