Array e liste

IASD a.a. 2015-16

Sequenze lineari

- n elementi $a_0, a_1, \ldots, a_{n-1}$ dove $a_j = (j+1)$-esimo elemento per $0 \leq ij \leq n-1$.
- In una sequenza è importante l'ordine degli elementi
  - Consideriamo una sequenza di transazioni bancarie.
  - Due tipi di transazioni: deposito e prelievo
  - Non è possibile prelevare una somma maggiore del saldo
  - La sequenza *deposita 5, preleva 2, deposita 10, preleva 11* non genera errori quando partiamo da un saldo pari a 0
  - Se amo le ultime due transazioni, la sequenza genera un errore in quanto tentiamo di prelevare 11 quando il saldo è uguale a 3

Sequenze lineari: modalità di accesso

- Due tipi:
  - **array**: sono sequenze ad accesso diretto in cui, dato j, si accede direttamente ad a_j
    - il costo di ciascun accesso è costante (non dipende da j)
  - **liste**: sono sequenze ad accesso sequenziale in cui, dato j, si accede ad $a_0, a_1, \ldots, a_j$ attraversando la sequenza a partire dall'inizio o dalla fine
    - costo $O(j)$ per accedere all'elemento di indice j partendo da a_0 e costo $O(n-j)$ partendo da a_{n-1}. In generale costo $O(p)$ per raggiungere a_{j+p} a partire da a_j.

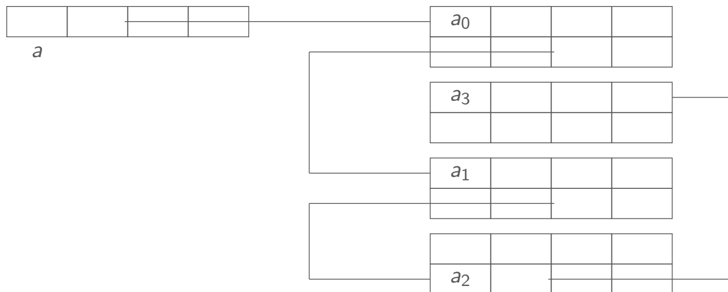
Sequenze: allocazione in memoria

- Array: memorizzato in locazioni di memoria consecutive
  - a: indirizzo di $a[0]$
  - $a[j]$: $a + j \times$ numero byte occupati da una singola cella
- Ad esempio se ciascuna cella dell'array occupa 4 byte allora $a[j]$: $a + j \times 4$

- Accesso all'elemento a_j in $O(1)$ tempo

Sequenze: allocazione in memoria

- Lista: gli elementi della lista sono memorizzati in locazioni di memoria non sono necessariamente contigue a causa del fatto che la memoria è gestita dinamicamente man mano che gli elementi vengono inseriti o cancellati dalla lista.
- Ciascun elemento della lista deve memorizzare l'indirizzo dell'elemento successivo se esiste
  - a: indirizzo del primo elemento a_0
  - L'elemento a_{j-1} contiene indirizzo elemento a_j se esiste
  - Per accedere ad a_j a partire da $a[0]$, occorre scandire i primi j elementi. Tempo O(j).



Array dinamici

- Alcuni linguaggi (C++, Java) prevedono **array** che possono essere **ridimensionati**
- Operazioni necessarie per ridimensionare un'array
  - Creare un nuovo array b
  - Copiare gli elementi di a in b
  - Deallocare a dalla memoria
  - Ridenominare b come a
- Costo $O(n)$ ($n =$ numero totale elementi)
- Metodo 1: L'array a viene ridimensionato aggiungendo o eliminando una posizione in fondo all'array
- **Vantaggio:** ottimale in termini di memoria allocata
- **Svantaggio:** oneroso in termini di computazionali.
- È possibile pagare tale costo ogni per tutti i ridimensionamenti?

Array dinamici

- Supponiamo di partire da un array di dimensione d e di aggiungere una posizione nuova ogni volta che l'array si riempie.
- Quanti spostamenti si fanno per n=d+f inserimenti?
  - I primi d inserimenti non richiedono di allocare un nuovo array: nessun spostamento.
  - I successivi richiedono un numero di spostamenti

$$\begin{aligned}\sum_{i=0}^{f-1}(d+i) &= \sum_{i=0}^{f-1} d + \sum_{i=0}^{f-1} i = fd + \frac{f(f-1)}{2} \\ &= (n-d)d + \frac{(n-d)(n-d-1)}{2}\end{aligned}$$

  - Quadratico!

Array dinamici: tecnica del raddoppio

- **Idea**: quando allochiamo un array più grande, lo facciamo in modo tale che, prima che venga allocato nuovamente un nuovo array, debbano essere fatti "molti" inserimenti".
- In questo modo il costo per spostare gli elementi nel nuovo array può essere "spalmato" sulle operazioni di inserimento effettuate tra un ridimensionamento dell'array e il successivo.
- Ciascun ridimensionamento richiede d spostamenti dove d è la dimensione dell'array in quel momento. Se il nuovo array ha dimensione $2d$ allora dovranno essere effettuati d nuovi inserimenti prima che venga nuovamente ridimensionato. Possiamo quindi "spalmare" il costo degli spostamenti sui d inserimenti.
- Il costo degli spostamenti mediato su tutta la sequenza di inserimenti è quindi $O(1)$

Array dinamici: algoritmo per raddoppiare la dimensione dell'array

```
1  VerificaRaddoppio( ):
2    IF (n == d) {
3      b = NuovoArray( 2 × d );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7    }
```

Array dinamici: tecnica del dimezzamento

- Quando nell'effettuare una cancellazione, ci accorgiamo che l'array è diventato troppo grande rispetto al numero degli elementi allora trasferiamo gli elementi in un array più piccolo.
- **Idea**: La taglia del nuovo array deve essere scelta in modo tale che prima che avvenga un nuovo ridimensionamento debbano avvenire molte cancellazioni.
- In questo modo il costo per trasferire gli elementi nel nuovo array può essere " spalmato" sulla cancellazioni effettuate tra un ridimensionamento e l'altro
- Ciascun ridimensionamento richiede m spostamenti dove m è il numero di elementi presenti nell'array.
- Se effettuiamo il ridimensionamento quando $m = d/4$, dove d è la dimensione dell'array in quel momento, e se il nuovo array ha dimensione $d' = d/2 = 2m$ allora il nuovo ridimensionamento avviene quando il numero di elementi diventa $d'/4 = m/2$
- Devono essere effettuate $m/2$ cancellazioni prima che l'array venga nuovamente ridimensionato. Possiamo quindi "spalmare" il costo degli $m/2$ spostamenti su queste m cancellazioni.
- Il costo degli spostamenti mediato su tutta la sequenza di icancellazioni è quindi $O(1)$

Array dinamici: algoritmo per il dimezzamento dell'array

```
1  VerificaDimezzamento( ):
2    IF ((d > 1) &&  (n == d/4)) {
3      b = NuovoArray( d/2 );
4      FOR (i = 0; i < n; i = i+1)
5        b[i] = a[i];
6      a = b;
7    }
```

Array dinamici: approccio “spalmato”

- $n = d$: raddoppia
- $n = d/4$: dimezza

- Dopo un raddoppio o dimezzamento:
  - occorrono almeno n inserimenti per un ulteriore raddoppio
  - occorrono almeno $n/2$ cancellazioni per un ulteriore dimezzamento
- Costo $O(n)$ di ridimensionamento è spalmato su almeno $n/2$ operazioni: costo $O(1)$ “ammortizzato” in più per operazione

Array dinamici: approccio “spalmato”

- Che succede se si dimezza la taglia dell’array quando $n = d/2$ anziché quando $n = d/4$?
- Consideriamo la seguente sequenza di $n = 2^k$ operazioni a partire da un array di dimensione 1:
- Effettuo $n/2 = 2^{k-1}$ inserimenti:
  - primo inserimento $\rightarrow$ non raddoppio ($d = 1$);
  - secondo inserimento $\rightarrow$ raddoppio ($d = 2$);
  - terzo inserimento $\rightarrow$ raddoppio ($d = 4$);
  - quarto inserimento $\rightarrow$ non raddoppio;
  - quinto inserimento $\rightarrow$ raddoppio ($d = 8$);
  - sesto, settimo e ottavo inserimento $\rightarrow$ non raddoppio;
  - nono inserimento $\rightarrow$ raddoppio ($d = 16$);
  - ...
  - inserimento $2^{k-2} + 1$-esimo raddoppio ($d = 2^{k-1}$)
  - ultimi $2^{k-2} - 1$ inserimenti $\rightarrow$ non raddoppio ($d = n/2 = 2^{k-1}$)

Array dinamici: approccio “spalmato”

- Ora effettuo una sequenza di 2^{k-2} inserimenti alternati a 2^{k-2} cancellazioni
  - primo inserimento (numero elem.$= 2^{k-1}+1$)$\rightarrow$ raddoppio ($d = 2^k$);
  - prima cancellazione (numero elem.$= 2^{k-1} = d/2$) $\rightarrow$ dimezzo ($d = 2^{k-1}$);
  - secondo inserimento (numero elem.$= 2^{k-1}+1$)$\rightarrow$ raddoppio ($d = 2^k$);
  - seconda cancellazione (numero elem.$= 2^{k-1} = d/2$) $\rightarrow$ dimezzo ($d = 2^{k-1}$);
  - ...
- quest'ultima sottosequenza di $2^{k-1} = n/2$ operazioni richiede di trasferire ogni volta $2^{k-1} = n/2$ elementi per un totale di

$$2^{k-1} \cdot 2^{k-1} = 2^{2(k-1)} = n^2/4 = \Theta(n^2)$$

  spostamenti

Ricerca sequenziale

- Array:

```
1   RicercaSequenziale( a, k ):
2     trovato = FALSE;
3     indice = -1;
4     FOR (i = 0; (i<n) && (!trovato); i = i+1) {
5       IF (a[i] == k) {
6         trovato = TRUE;
7         indice = i;
8       }
9     }
10    RETURN indice;
```

Applicazione ordinamento: scheduling della CPU

- Task P_0, P_1, P_2, P_3 da eseguire in sequenza sulla CPU del calcolatore
- Durata in millisecondi di ciascun task: $t_0 = 21$, $t_1 = 3$, $t_2 = 1$, $t_3 = 2$
- **Scheduling**: sequenza di esecuzione dei task

First Come First Served (FCFS): i task vengono eseguiti in ordine di arrivo

Tempo medio d'attesa: $(0 + 21 + 24 + 25)/4 = 17,5$ ms

Applicazione dell'ordinamento: scheduling della CPU

- È possibile migliorare il tempo medio d'attesa?
- **Idea**: i task che richiedono meno tempo vengono eseguiti prima

Shortest Job First (SJF): i task vengono eseguiti in ordine di durata.

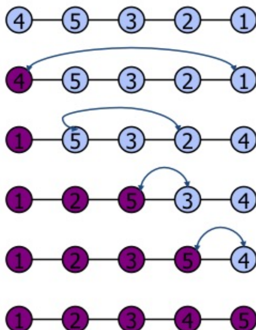
Tempo medio d'attesa: $(0 + 1 + 3 + 6)/4 = 2,5$ ms

Ordinamento

- *Dato un array di n elementi e una loro relazione d'ordine $\leq$, vogliamo disporli in modo che risultino ordinati in modo crescente*
- Vediamo due semplici algoritmi:
  - SelectionSort
  - InsertionSort
- In seguito vedremo algoritmi più efficienti

Selection-sort sul posto

- **Selection-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Al passo i-esimo **selezioniamo** l'elemento minimo tra quelli non ancora ordinati e lo scambiamo con l'i-esimo elemento della sequenza



SelectionSort

- Passo i: **seleziona** l'elemento di rango $(i+1)$ ossia il minimo tra i rimanenti $n-i$ elementi

```
1  SelectionSort( a ):
2    FOR (i = 0; i < n-1; i = i+1) {
3      minimo = a[i];
4      indiceMinimo = i;
5      FOR (j = i+1; j < n; j = j+1) {
6        IF (a[j] < minimo) {
7          minimo = a[j];
8          indiceMinimo = j;
9        }
10     }
11     a[indiceMinimo] = a[i];
12     a[i] = minimo;
13   }
```

Analisi del SelectionSort

- Chiamiamo t_i il costo dell'esecuzione del ciclo di for esterno al passo i
- Il costo t_i è dominato dal costo del ciclo for interno
- Il corpo del ciclo for interno richiede $n - i - 1$ iterazioni, ciascuna di costo costante
- Quindi $t_i = O(n - i)$ per il for esterno
- In totale SelectionSort richiede un numero di operazioni proporzionale a

$$\sum_{i=0}^{n-2}(n-i) = \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1 = \Theta(n^2).$$

InsertionSort

- Passo *i*: **inserisci** l'elemento in posizione *i* al posto giusto tra i primi *i* elementi (già ordinati)

```
1  InsertionSort( a ):
2    FOR (i = 0; i < n; i = i+1) {
3      prossimo = a[i];
4      j = i;
5      WHILE ((j > 0) && (a[j-1] > prossimo)) {
6        a[j] = a[j-1];
7        j = j-1;
8      }
9      a[j] = prossimo;
10   }
```

Analisi dell'InsertionSort

- Usiamo le regole viste: al passo i del `FOR` esterno, il costo t_i è dominato dal costo del ciclo `WHILE` interno
- Il ciclo `WHILE` interno richiede al massimo $i + 1$ iterazioni, ciascuna di costo costante
- Quindi $t_i = O(i + 1)$ per il `FOR` esterno
- In totale InsertionSort richiede $O(n^2)$ tempo perché questo è proporzionale a

$$\sum_{i=0}^{n-1}(i + 1) = \frac{n(n + 1)}{2}$$

- Osservazione: può richiedere $O(n)$ operazioni (quando?)

Esercizio

1.1 Dato un array di n elementi con ripetizioni, sia k il numero di elementi distinti in esso contenuti. Progettare ed analizzare un algoritmo che restituisca un array di k elementi contenente gli elementi distinti dell'array originale.

Soluzione

- Prima ordiniamo gli elementi
- Una volta ordinati gli elementi li scandiamo a partire da a[0]
  - Ogni volta che si passa da un elemento più piccolo ad uno più grande, l'elemento più piccolo viene inserito nell'array output.

```
1  SelectDistinctElts( a,k ):
2    InsertionSort(a);
3    crea un nuovo array b di lunghezza k
4    p=0; i=0;j=0;
5    WHILE (i< n) {
6      j = j+1;
7      WHILE ((j < n) && (a[j] ==a[i])) {
8        j=j+1;
9      }
10     b[p]=a[i];
11     p=p+1;
12     i = j;
13   }
14   RETURN b;
```

Soluzione

- Analisi dell'algoritmo.
- Esecuzione di InsertionSort richiede $O(n^2)$
- Corpo del while esterno eseguito al massimo n volte.
  - Se si esclude il costo per eseguire il while interno, il costo di ciascuna iterazione del while esterno, è $O(1)$
- Indichiamo con t_i il numero di iterazioni del corpo del while interno durante l'iterazione i del while esterno
- singola esecuzione del while interno $O(1) \rightarrow$ costo totale del while interno $O(\sum_{i=1}^{n} t_i)$
- in ciascuna iterazione del while interno j viene incrementato di 1 e non viene mai decrementato in nessun punto dell'algoritmo per cui il numero totale di iterazioni del while interno non può eccedere n

$$\sum_{i=1}^{n} t_i \leq n$$

- Se si esclude il costo dell'ordinamento, il costo dell'algoritmo è O(n). In totale il costo è $O(n^2)$

Soluzione

- Possiamo usare un unico ciclo di while

```
1  SelectDistinctElts( a ,k):
2    InsertionSort(a);
3    crea un nuovo array b di lunghezza k
4    p=0;
5    i=0;
6    WHILE (i< n) {
7      IF ((i+1< n) && (a[i+1] >a[i])) {
8        b[p]=a[i];
9        p=p+1;
10     }
11     IF (i= n-1) {
12       b[p]=a[i];
13     }
14     i=i+1;
15   }
16   RETURN b;
```

Esercizio

1.3 Fornire un algoritmo lineare che, dato un array di n numeri, determini la sottosequenza più lunga di elementi contigui dell'array non decrescente. Ad esempio se l'array è 1, 4, 3, 2, 5, 3, 7, 8, 9, 6, 10, allora la sottosequenza è 3 ,7,8,9.

Soluzione

- Scandisco gli elementi da sinistra a destra
- Tengo traccia della lunghezza della sottosequenza non decrescente più lunga incontrata (massimo)
- Uso un contatore per contare gli elementi di ciascuna sottosequenza non decrescente
- Fino a che l'elemento scandito è minore o uguale di quello successivo incremento il contatore
- Quando si ha che l'elemento scandito è maggiore di quello successivo controllo se devo aggiornare il massimo
- Nella soluzione proposta la funzione restituisce un array di lunghezza 2 contente gli indici in cui la sottosequenza desiderata inizia e finisce (l'array deve essere aggiornato ogni volta che si aggiorna il massimo)

Soluzione

```
1  LNDS( a ):
2    i=0; maxLength=0;
3    WHILE (i< n) {
4      count=1;
5      j=i;
6      WHILE ((j+1< n) && (a[j] <=a[j+1])) {
7        count=count+1;
8        j=j+1;
9      }
10     IF (maxLength<count) {
11       maxLength = count;
12       toReturn[0]=i;
13       toReturn[1]=j;
14     }
15     i=j+1
16   }
17   RETURN toReturn;
```

Soluzione

- Analisi dell'algoritmo.
- Corpo del while esterno eseguito al massimo n volte.
  - Se si esclude il costo per eseguire il while interno, il costo di ciascuna iterazione del while esterno, è $O(1)$
- Indichiamo con t_i il numero di iterazioni del corpo del while interno durante l'iterazione i del while esterno
- singola esecuzione del while interno $O(1) \rightarrow$ costo totale del while interno $O(\sum_{i=1}^{n} t_i)$
- in ciascuna iterazione del while interno j viene incrementato di 1 e non viene mai decrementato in nessun punto dell'algoritmo per cui il numero totale di iterazioni del while interno non può eccedere n

$$\sum_{i=1}^{n} t_i \leq n$$

Liste

- sequenza lineare di elementi ad accesso sequenziale
  - elemento `x`
    - `x.succ` è il successore di `x` (`null` se `x` è l'ultimo elemento della lista)
    - `x.dato` è il contenuto informativo di `x`
  - codice per accedere all'elemento in posizione i

```
p = a;
j = 0;
WHILE ((p != null) && (j < i)) {
  p = p.succ;
  j = j+1;
}
```

- tempo $O(i)$:

Ricerca sequenziale

- Liste:

```
1 RicercaSequenziale( a, k ):
2   p = a;
3   WHILE ((p != null) && (p.dato != k))
4     p = p.succ;
5   RETURN p;
```

Inserimento in testa

- Codice per l'inserimento in testa alla lista

```
x.succ = a;
a = x;
```

- Prima dell'inserimento

- Dopo l'inserimento

Inserimento in una posizione interna

- Inserimento dopo una posizione *p*

```
x.succ=p.succ;
p.succ=x;
```

Cancellazione

- Cancellazione del primo elemento

```
t=a.succ
a.succ = null;
a=t;
```

- Cancellazione dell'ultimo elemento
  - $x =$ elemento da cancellare
  - $p =$ predecessore di x

```
p.succ=null;
```

- Cancellazione dell'unico elemento della lista

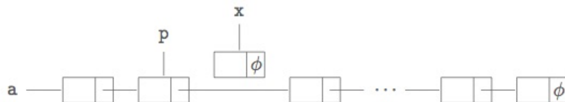
```
a=null;
```

Cancellazione di un elemento interno

```
p.succ = x.succ;
x.succ = null;
```

- Prima della cancellazione

- Dopo la cancellazione



Esercizio

- Scrivere lo pseudocodice di un algoritmo che prende in input due liste ordinate e restituisce una nuova lista ordinata contenente gli elementi delle due liste input. Si assuma che ciascun elemento di una lista contiene il riferimento all'elemento successivo.

Soluzione

- Cominciamo con lo scrivere una funzione che ci sarà utile per aggiungere elementi alla lista output.

•Funzione che inserisce un elemento x alla fine della lista a
- La lista potrebbe essere vuota

```
InserisciAllaFine(a,x,p):     //p e` l'ultimo elemento; p=null se la
lista a e` vuota
   x.succ=null;
   If(p!=null) //lista a non vuota
      p.succ=x;
  Else a=x;   //lista a vuota
  p=x;
  Return p;
```

Soluzione

•Funzione che fonde due liste ordinate a1 e a2 in un'unica lista ordinata

```
Merge(a1,a2){
1.  output=null;
2.  predecessor=null;
3.  daInserire =null;
4.  c1=a1;        //cursore per scandire a1
5.  c2=a2         //cursore per scandire a2
6.  While(c1!=null && c2!=null){
7.  daInserire= NuovoNodo();
8.  If(c1.dato≤c2.dato){
9.     daInserire.dato=c1.dato;
10.    c1=c1.succ;  }      //avanzo in a1
11. Else{
12.   daInserire.dato=c2.dato;
13.    c2=c2.succ;  }      //avanzo in a2
14. predecessor=InserisciAllaFine(output,daInserire,predecessor);
15. }    //fine While
```


continua

Soluzione

```
16.  While(c1!=null){
17.    daInserire= NuovoNodo();
18.    daInserire.dato=c1.dato;
19.    predecessor=InserisciAllaFine(output,daInserire,predecessor);
20.  c1=c1.succ;
21.   }
22.  While(c2!=null){
23.    daInserire= NuovoNodo();
24.    daInserire.dato=c2.dato;
25.    predecessor=InserisciAllaFine(output,daInserire,predecessor);
26.    c2=c2.succ;
27.   }
28.  Return output;
```

Tempo O(n), dove n \`e\` la somma del numero di elementi di a1 e del numero di elementi di a2. Perché?

Esercizio svolto nel libro

- Scrivere lo pseudocodice di un algoritmo ricorsivo che inverte una lista. Si assuma che ciascun elemento di una lista contiene il riferimento all'elemento successivo.

Soluzione

```
InvertiLista(p,x):
 IF(x.succ!=null){
   b=Inverti(x,x.succ);
}
ELSE{
  b=x;                    //caso base in cui x fa riferimento all'ultimo nodo della lista
}
x.succ=p;
RETURN b;
```

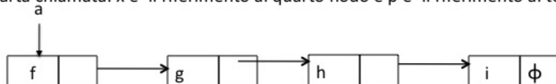
Per invertire la lista a, invochiamo la funzione con x=a e p=null (p rappresenta il predecessore di x)

Prima chiamata: x e` il riferimento al primo nodo e p=null

Seconda chiamata: x e` il riferimento al secondo nodo e p e` il riferimento al primo nodo

Terza chiamata: x e` il riferimento al terzo nodo e p e` il riferimento al secondo nodo

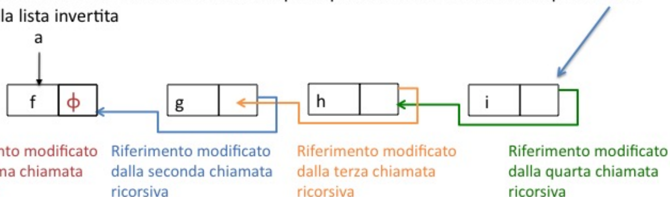
Quarta chiamata: x e` il riferimento al quarto nodo e p e` il riferimento al terzo nodo



Soluzione

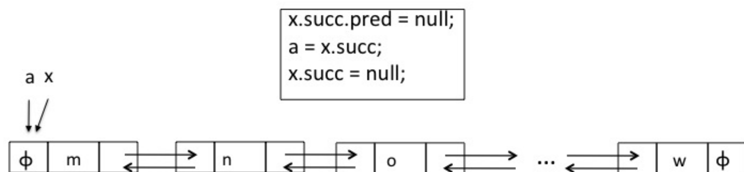
```
InvertiLista(p,x):
 IF(x.succ!=null){
   b=Inverti(x,x.succ);
}
ELSE{
  b=x;                  //caso base in cui x fa riferimento all'ultimo nodo della lista
}
x.succ=p;
RETURN b;
```

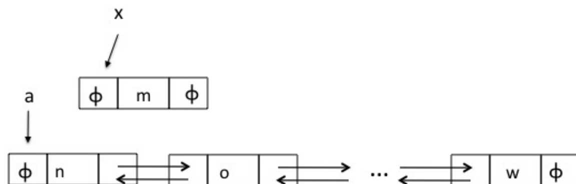
Ciascuna chimata ricorsiva resituisce a quella piu` esterna il riferimento al primo nodo della lista invertita



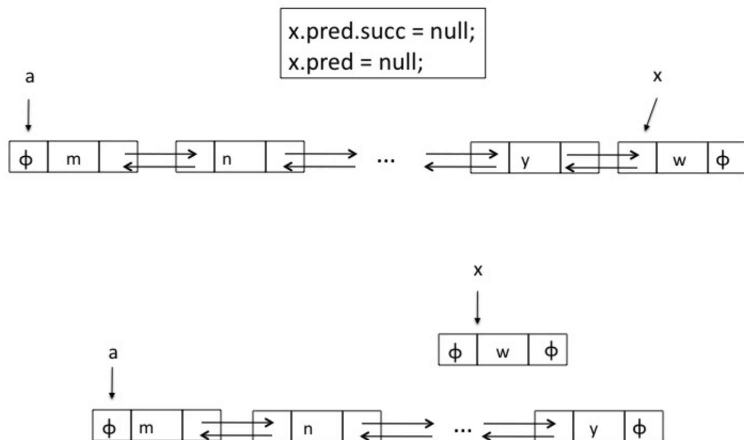
Liste doppie

Cancellazione di x: x fa riferimento al primo elemento della lista





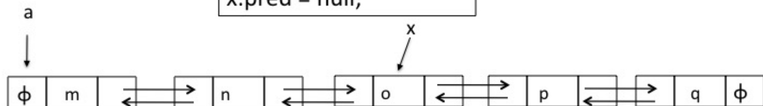
Liste doppie

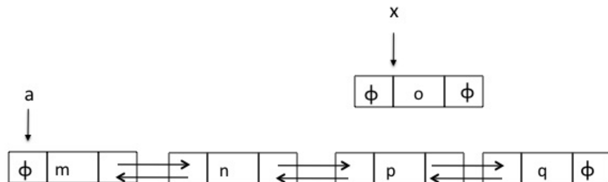


Liste doppie

Cancellazione di x: x fa riferimento ad un elemento interno

```
x.succ.pred = x.pred;
x.pred.succ = x.succ;
x.succ = null;
x.pred = null;
```

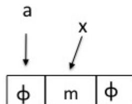




Liste doppie

Cancellazione di x: x fa riferimento all’unico nodo della lista

```
a= null;
```

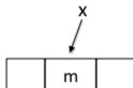


a=ϕ

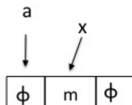
Liste doppie

Inserimento di x in una lista vuota

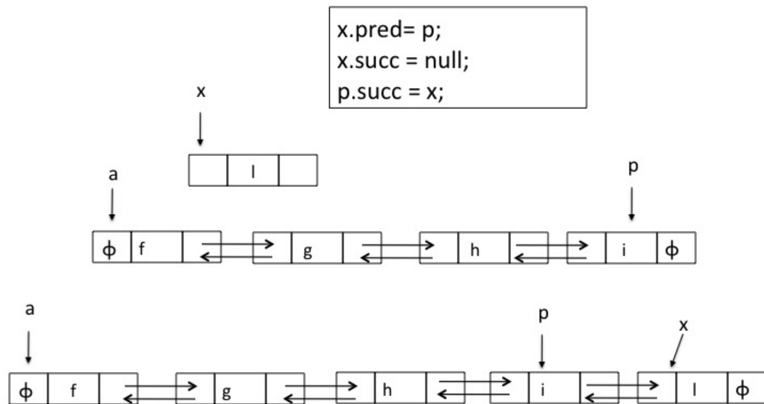
```
x.pred = null;
x.succ = null;
a = x;
```



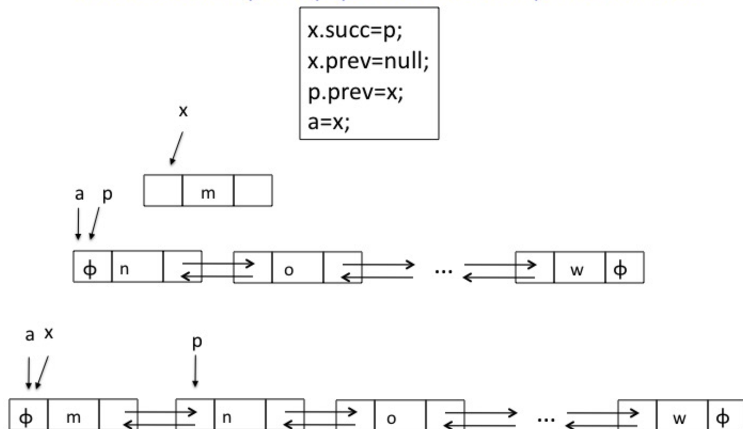
a=ϕ



Liste doppie



Liste doppie



Liste doppie