Corso sul linguaggio C++

Modulo 4

3 – Le liste

Prerequisiti

- Strutture dinamiche
- Allocazione dinamica
- Uso di puntatori

Introduzione

In questa Unità si applicano i concetti di allocazione dinamica, di puntatore e di lista con le sue operazioni e si esamina nella seconda parte l'importanza degli algoritmi ricorsivi sulle liste.

Argomenti

- Array e strutture dinamiche
- Lista concatenata
- Caratteristiche di una lista
- Implementazione della lista
- Operazioni su liste
- Creazione lista
- Creazione cella
- Inserimento
  - in testa
  - in coda
  - intermedio
- Eliminazione
  - in testa
  - in coda
  - della lista
- Operazioni varie
  - Ricerca di un valore
  - Test lista vuota
  - Stampa della lista
  - Lunghezza della lista
  - Calcolo del valore medio
  - Calcolo del valore massimo
- Liste e ricorsività
- Altri tipi di liste

Array e strutture dinamiche

L'array può richiedere:

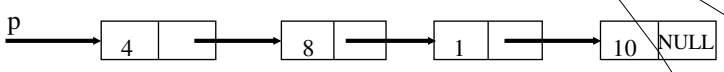
- **sovradimensionamento**: array molto più grande di quanto effettivamente necessiti, per disporre di un sufficiente "margine di sicurezza"
- **compattazione**: necessario spostare continuamente gli elementi dell'array per gestire gli spazi vuoti creatisi ad ogni nuovo evento

Con l'allocazione dinamica

- la memoria impegnata è ogni volta solo quella strettamente necessaria
- le operazioni di gestione degli ***eventi*** (introduzione o rimozione di un oggetto nella lista) sono più efficienti.

Lista concatenata

Un'applicazione dell'allocazione dinamica è la gestione di **liste**.
Una **lista semplice concatenata** ha la struttura seguente:



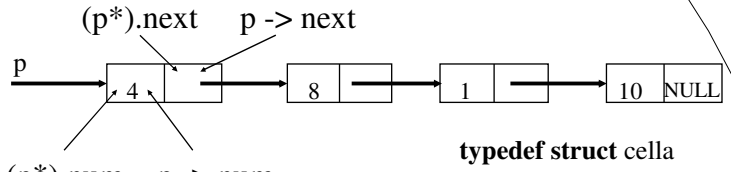
Caratteristiche di una lista

Nella gestione di una lista:

- la struttura di controllo **while** è la più indicata;
- è cura del programmatore far sì che il **puntatore d'accesso** sia conservato, pena la perdita di dati e l'esaurimento della memoria heap;
- la scansione della singole celle segue un **accesso sequenziale**;
- è importante che l'ultima cella abbia il puntatore **NULL** che indica il termine della lista;
- occorre tenere presente che la lunghezza della lista **varia durante l'esecuzione** del programma.

Implementazione della lista

L'accesso ai campi delle celle tramite il loro indirizzo avviene mediante **l'operatore di indirezione (*)**. Nella lista di interi rappresentata sotto, e definita come indicato nella **struct**, possiamo usare due sintassi equivalenti per accedere ai campi.




```
typedef struct cella
{
    int num;
    cella *next;
};
cella *p;
```

Operazioni su liste

- **Creazione** della lista vuota
- **Inserimento** di un nuovo elemento
  - in testa
  - in coda
  - in una posizione specifica
- **Eliminazione** di un elemento
  - in testa
  - in coda
  - Intermedio
  - Globale della lista
- **Operazioni varie**

Per ciascuna di queste operazioni occorre tenere presente che sarà cura del programmatore mantenere i collegamenti tra nodi consecutivi, mediante assegnazioni sui puntatori.

Creazione lista

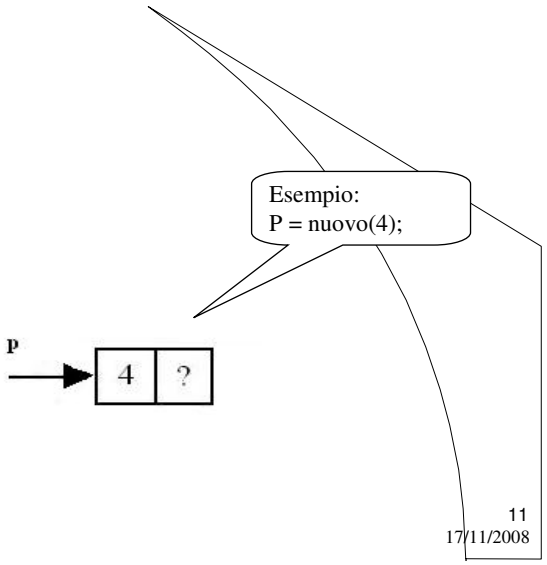
Si inizializza la lista ponendo

p = **NULL**;

*Supponiamo di trattare liste di numeri interi; in casi diversi il campo **num** sarà una **struct** opportuna.*

Creazione cella

```
cella *nuovo (int n)
{       cella *tmp= new cella;
        tmp->num=n;
        return tmp;
}
```



Inserimento

Sono possibili i seguenti tipi di inserimento:

- inserimento in testa
- inserimento in coda
- inserimento in una data posizione

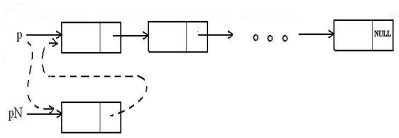
Inserimento
Inserimento in testa

Richiede:
- creazione di una cella *pN* mediante la funzione *nuovo(x)*;
- inserimento di *pN* nella lista puntata da *p*.

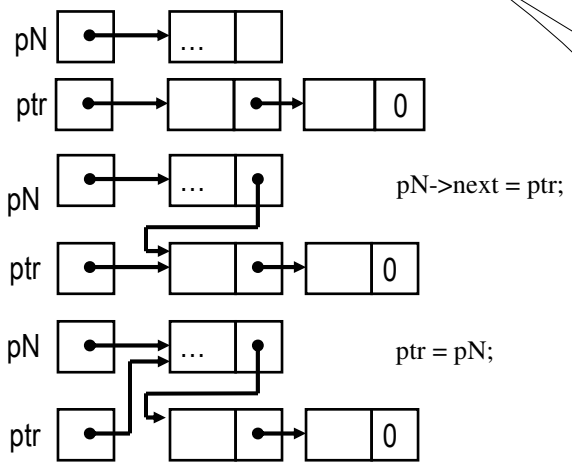
```
void InsTesta (cella* &ptr, cella *pN)
{       pN->next=ptr;
        ptr=pN;
}
```

Nel main si ha:

```
…
cella *pN = nuovo(x);
InsTesta(p, pN);
….
```



Inserimento
Inserimento in testa



Inserimento
Inserimento in coda

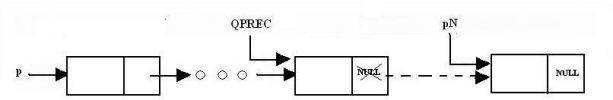
```
void InsCoda (cella* &ptr, cella *pNl)
{  cella *temp;
   if (!ptr)   InsTesta(ptr, pN);
   else
   {  temp=ptr;
      while((*temp).next)
         temp=(*temp).next;
      (*temp).next=pN;
      pN.next = NULL;
   }
}
```

Per inserire una cella *pN* avente valore *x* in coda alla lista *p* occorre:

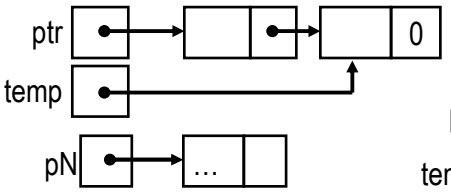
- scansione della lista con un puntatore di lavoro *temp* fino *all'ultima cella*
- inserimento di pN;

Nel main si ha:

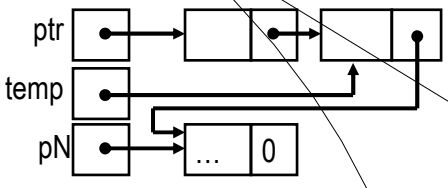
…
cella *pN = nuovo(x);
InsCoda(p, pN);
….

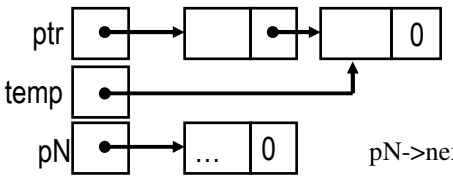


Inserimento
Inserimento in coda



temp->next = pN;





pN->next = NULL;

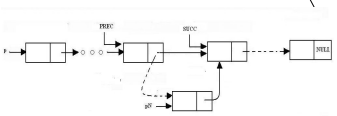
Inserimento

Inserimento intermedio

```
void ins_pos(cella* &lis, int a, int pos)
{   int i=0;
    cella *prec, *succ, *t;
    if (pos>lunghezza(lis))  ins_coda(lis, a);
    else
      if ((lis==NULL)|| (pos<1)) ins_testa(lis, a);
      else  {   prec=lis;
                succ=lis->next;
                while (i<pos-1)
                {    prec=succ;
                     succ=succ->next;
                      i++;
                 }
          nuovo(t, a);
          prec->next=t;
          t->next=succ;
      }
}
```

Per inserire una cella *pN* avente valore *x* in posizione n della lista *p* occorre:

- scansione della lista con due puntatori *prec* e *succ* fino *a che prec giunga in posizione n;*
- inserimento di pN;



Abbiamo supposto esistente la funzione *lunghezza(p)* (che dà il numero di celle presenti) la cui implementazione è lasciata al lettore.

Eliminazione

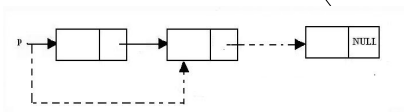
Sono possibili i seguenti tipi di eliminazione:

- eliminazione in testa
- eliminazione in coda
- eliminazione intermedia (lasciata al lettore)
- eliminazione della lista

Eliminazione
Eliminazione in testa

La funzione per eliminare una cella dalla testa della lista *ptr* si può esprimere come segue:

```
void  CancTesta (cella* &ptr)
{
   if (ptr!=NULL)
   {
       cella* t=ptr;
       ptr=ptr->next;
       delete t;
    }
}
```



Nel main si ha:

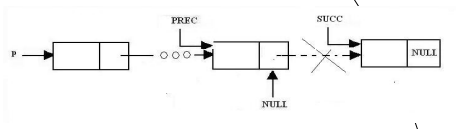
```
…
cella *pN = nuovo(x);
CancTesta(p);
….
```


Eliminazione
Eliminazione in coda

```
void CancCoda (cella* &ptr)
{  if (ptr != NULL)
   {
    cella* prec = ptr;
    cella* succ = ptr->next;
    while (succ->next != NULL)
    {
        prec = succ;;
        succ = succ->next;
     }
    prec->next=NULL;
    delete succ;
   }
}
```

Per eliminare una cella dalla coda della lista *p* occorre:

- scansione della lista con due puntatori *prec* e *succ* fino a che *succ punti all'ultima cella;*
- eliminare *succ*;



Nel main si ha:

```
…
CancCoda(p);
….
```

Eliminazione
Eliminazione della lista

```
void eliminaLista (cella* &ptr)
{
    cella  *temp;
    while (ptr!=NULL)
    {
       temp=ptr;
       ptr=ptr->next;
       delete temp;
    }
}
```

L'eliminazione dell'intera lista deve essere effettuata deallocando cella per cella, mediante un ciclo; porre semplicemente **NULL** nel puntatore di accesso, rischia di rendere inaccessibili le celle allocate e a lungo andare potrebbe causare problemi di esaurimento della memoria *heap*, sebbene i moderni compilatori dispongano di efficienti *garbage collector.*

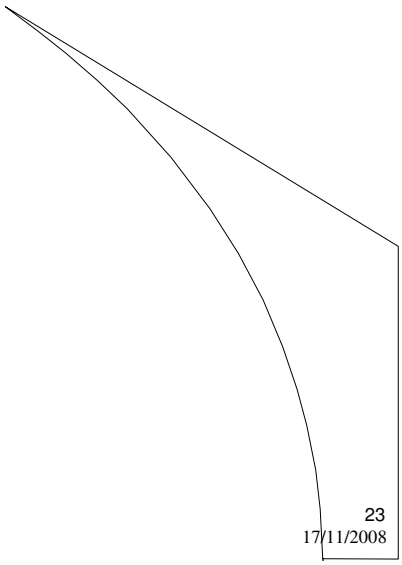
Operazioni varie
Ricerca di un valore

```
int esiste (cella* ptr, int n)
{ cella *p=ptr;
  int i=0;
  while (p && p->num!=n)
  {  p=p->next;
     i++;
  }
 if (p==NULL)
     return 0;
 else
     return i+1;
}
```

Per cercare la posizione di un certo valore *val* nella lista *p* occorre scandire la lista con un puntatore *t* fino a quando non è finita oppure **Info(t)** è uguale a *val*.

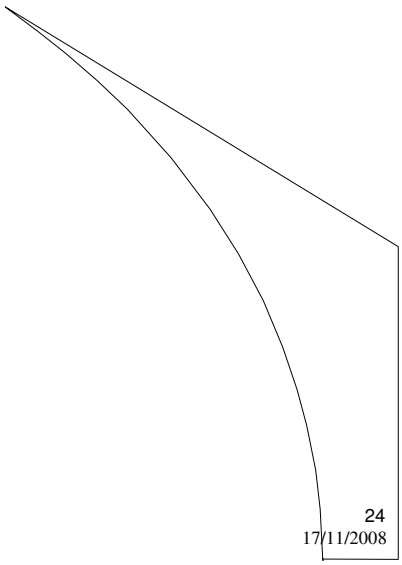
Operazioni varie
Test lista vuota

```
bool vuota(cella* ptr)
{
        return ptr==NULL;
}
```



Operazioni varie
Stampa della lista

```
void show (cella* ptr)
{   cella *p=ptr;
    cout<<"Lista: [";
    while (p!=NULL)
    {
        cout<<p->num;
        p=p->next;
    }
    cout<<"]“<<endl;
}
```



Operazioni varie
Lunghezza della lista

```
int lunghezza(cella *ptr)
{
   int i=0;
   while (ptr!=NULL)
   {
      c++;
      ptr=ptr->next;
   }
   return c;
}
```


Operazioni varie
Calcolo del valore medio

```
float media(cella *ptr)
{  int s=0;
   float m;
   cella *p=ptr;
   if (p==NULL)   m=0;
   else
   {   while (p!=NULL)
       {  s+=p->num;
          p=p->next;
       }
       m=(float) s/lunghezza(ptr);
   }
return m;
}
```

Operazioni varie

Calcolo del valore massimo

```
int max(cella *ptr)
   {   int m=0;
       cella *p=ptr;
       if (p!=NULL)
       {  m=p->num;
          p=p->next;
         while (p!=NULL)
         {  if (p->num>m)
             m=p->num;
             p=p->next;
          }
       }
      return m;
   }
```


Liste e ricorsività

- **Esistenza elemento ricorsiva**

```
int esiste (cella *ptr, int v)
{
  if (!ptr)   return 0;                 /* la lista è vuota, l'elemento…*/
                                        /* …  non è presente */
  else
    if (ptr->info== v)                  /* se v è il primo elemento…*/
       return 1;                        /*…restituisci 1, altrimenti…*/
   else
     return (esiste(ptr->next, v));     /* …cerca nella parte restante */
}
```

Liste e ricorsività

- **Inserimento ordinato ricorsivo**

```
void ins_ord_ric (cella* &lis, int a)
{     cella *t;
      if (lis==NULL || lis->info>a)
          ins_testa(lis, a);
      else
      {
          t=lis->next;          /* punta al resto della lista */
          ins_ord_ric(t, a);    /* inserisce nella sottolista */
          lis->next=t;          /* collega con l'istanza sospesa */
   }
}
```

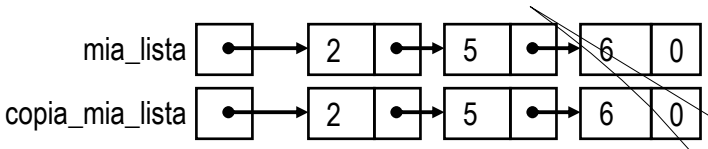

Liste e ricorsività

- **Inserimento in coda ricorsivo**

```
void inscoda(rec* & lis, int a)
{
     if (lis == NULL)          // passo base: inserisce
        instesta (lis, a);
     else
        inscoda( lis->next, a );    //passo induttivo: inserisce in coda al resto
}
```

Copia di una lista

- Obiettivo: data una **lista**, costruirne una **copia**



rec* copia(rec* lis);

```
…
rec* mia_lista=NULL;
… (riempimento lista mia_lista) …
rec* copia_mia_lista;
copia_mia_lista = copia( mia_lista );
…
```


Esempio: Copia di una lista

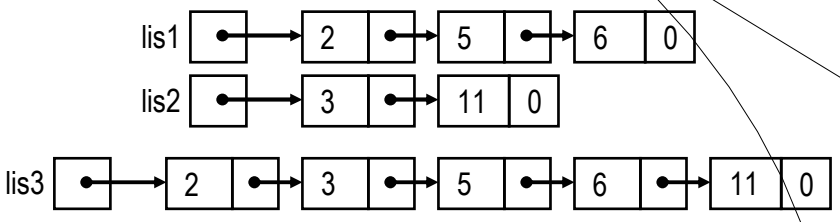
```
rec* copia(rec* lis)
{
   if (lis == NULL)          //passo base: se la lista è vuota la copia è vuota
      return lis;
   else {
         rec* tempp = new rec;    //altirimenti copio un elemento
         temp->info = lis->info;
         tempp->next = copia( lis->next );  //e mi faccio dare la copia di resto
         return tempp;
    }
}
```

È ancora una lista

Copia di una lista versione ricorsiva

Fusione di due liste

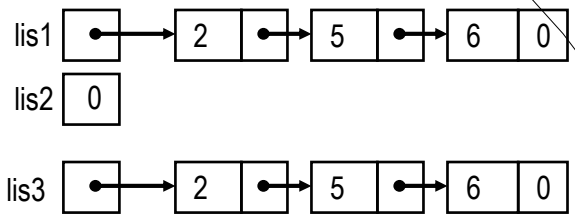
- Obiettivo: data due **liste ordinate**, **fonderle** in un'unica lista ordinata.



rec* merge(rec* & lis1, rec* & lis2);

Fusione di due liste: passo base

- Se una delle due liste è **vuota**, allora il risultato della fusione è **l'altra**.

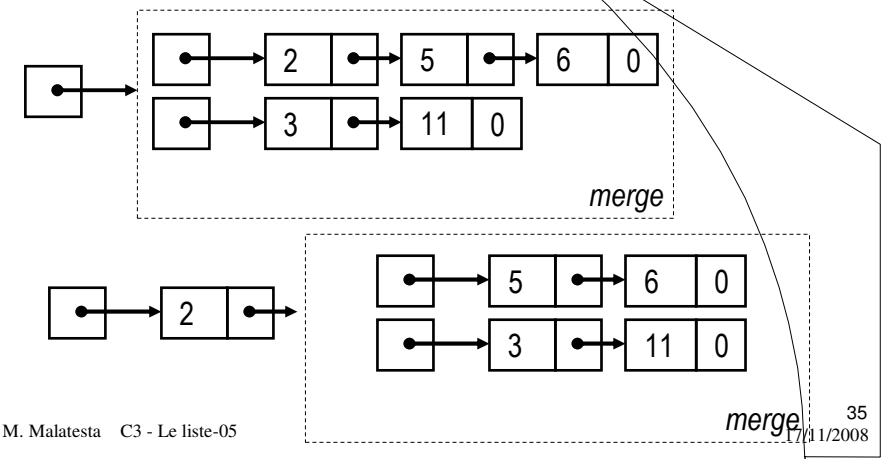


if (lis2==NULL) lis3=lis1;

if (lis1==NULL) lis3=lis2;

Fusione di due liste: passo induttivo

- Il risultato dalla fusione di due liste è uguale alla lista ottenuta ***inserendo*** l'elemento **minore** tra le due teste in **testa** al risultato della ***fusione dei resti***

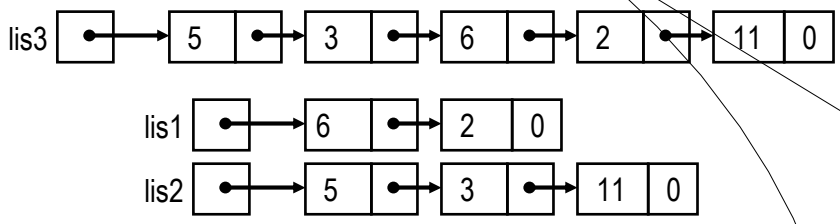


Fusione di due liste ordinate

```
rec* merge(rec* & lis1, rec* & lis2) {
  rec* lis3;
  if ( lis1 == NULL ) { lis3 = lis2; lis2 = NULL; return lis3 ; }  //passo base
  if ( lis2 == NULL) { lis3 = lis1; lis1 = NULL; return lis3 ; }   //passo base
  if ( lis1->info < lis2->info ) {       // scelgo l'elemento minore
     lis3=lis1; lis1=lis1->next;       // lo stacco
  }
  else {
     lis3=lis2; lis2=lis2->next;
  }
  lis3->next = merge( lis1, lis2 );  // passo ricorsivo
  return lis3 ;
}
```

Splitting di una lista

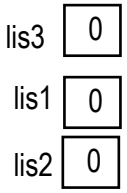
- Obiettivo: data una lista, **dividerla** in due liste, una contente i numeri ***pari*** e una i ***dispari***.



void split(rec* & lis3, rec* & lis1, rec* & lis2);

Splitting di una lista: passo base

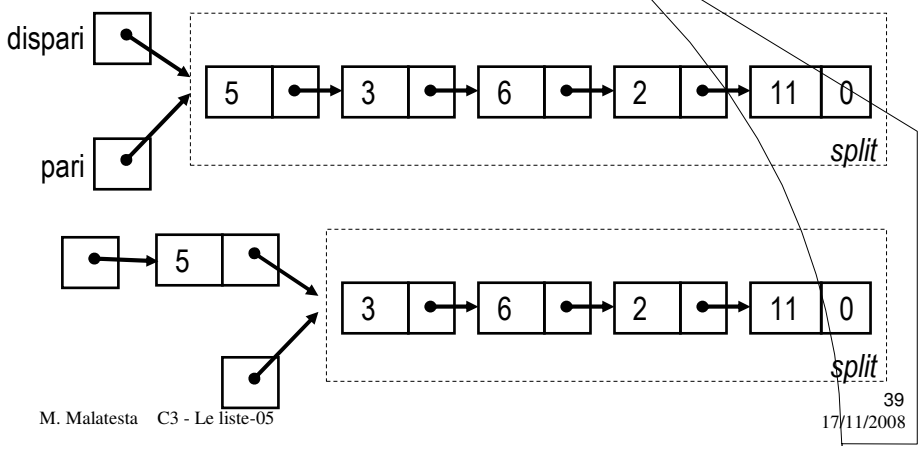
- Se la lista è **vuota**, allora il risultato dello splitting sono **due liste vuote**.



if (lis3==NULL) { lis1=NULL; lis2=NULL; }

Splitting di una lista: passo induttivo

- I risultati dello **splitting** di una lista sono uguali alle liste ottenute assegnando la **testa** della lista ai risultati dello ***splitting del resto***



Splitting di una lista

```
void split( rec* & lis3, rec* & lis1, rec* & lis2 ) {
      if ( lis3 != NULL ) {
        rec* tempp = lis3;
        lis3 = lis3->next;
        if ( (tempp->info) % 2 ) { // dispari ? (resto div 2 diverso da 0)
           lis1=tempp;
           split( lis3, lis1->next, lis2 );       // ricorsione
        } else {                          // pari
           lis2=tempp;
           split( lis3, lis1, lis2->next );       // ricorsione
        }
     } else {
        lis1=NULL; lis2=NULL;     // passo base
      }
}
```

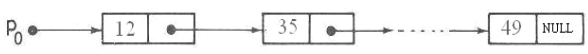
Altri tipi di liste

Altri tipi di liste che possiamo avere sono:

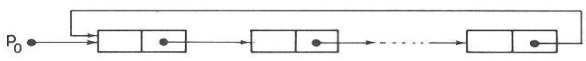
- liste ordinate
- liste circolari
- liste bidirezionali
- liste bidirezionali circolari
- liste multiple

Altri tipi di liste

La **lista ordinata** ha la caratteristica di avere i valori dei dati disposti in ordine, ad esempio crescente

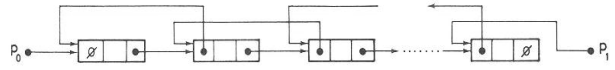


La **lista ordinata** ha la caratteristica di avere i valori dei dati disposti in ordine, ad esempio crescente

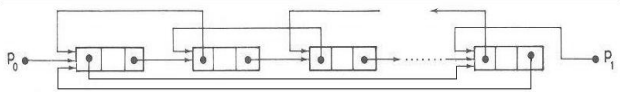


Altri tipi di liste

La **lista bidirezionale** ha la caratteristica di avere due puntatori di accesso: uno all'inizio p0 ed uno alla fine p1.

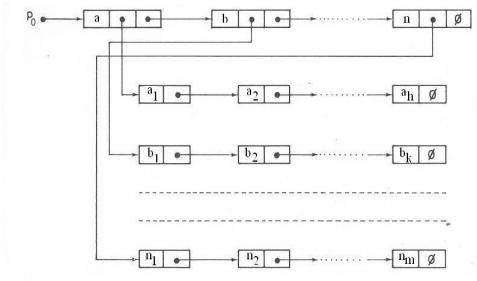


La **lista bidirezionale circolare** ha le caratteristiche di quella circolare e della bidirezionale; può essere visitata in entrambi i sensi e non ha il puntatore **NULL** alle estremità



Altri tipi di liste

La **lista multipla** ha la caratteristica di avere in ogni nodo, oltre al puntatore al successivo, un puntatore ad una **sottolista**. Consiste in sostanza, in una lista a più livelli.



Cosa ho imparato

- Come utilizzare la memoria dinamica per creare liste
- Cosa è una lista semplice concatenata
- Quali sono altri tipi di liste
- Quali sono le operazioni principali su una lista

Cosa ho imparato a fare

- Allocare e deallocare memoria dinamicamente
- Implementare operazioni su liste semplici
- Riconoscere diversi tipi di liste

Terminologia

- Lista semplice concatenata
- Puntatore d'accesso
- Liste ordinate
- Liste circolari
- Liste bidirezionali
- Liste bidirezionali circolari
- Liste multiple

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- A.Garavaglia, F.Petracchi,S.Forte-INFORMATICA-ed. Masson