Sommario

- Collezioni:
  - Array e liste.
  - Nodi: dati e riferimenti.
- Liste:
  - LinkedList: specifica e implementazione.
  - Prestazioni.

Collezioni

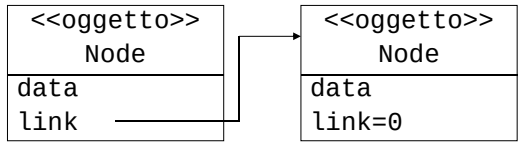
- Una *collezione* (contenitore) è un oggetto che raggruppa più dati in una singola struttura. Vi sono due tecniche *fondamentali* per rappresentare collezioni di elementi: quella basata su *strutture indicizzate* (*array*, composti da elementi consecutivi) e quella basata su *strutture collegate* (*list*, composte da nodi: ogni nodo contiene dati e collegamenti ad altri nodi).
- Le collezioni sono usate per memorizzare, leggere, manipolare dati strutturati e per trasmettere tali dati da un metodo ad un altro.

Collezioni

- Fino ad ora si sono usate le strutture dati array (semplici e performanti). La loro caratteristica è di avere celle memorizzate sequenzialmente che sono accessibili attraverso indici con costo *O(1)*.
- Tuttavia si può pensare di usare strutture dati in cui l'*ordinamento* dei dati è solo *a livello logico* (senza usare un sottostante ordinamento fisico).
- Questo si può realizzare associando ad ogni elemento anche un *puntatore* all'elemento che lo segue nella sequenza.

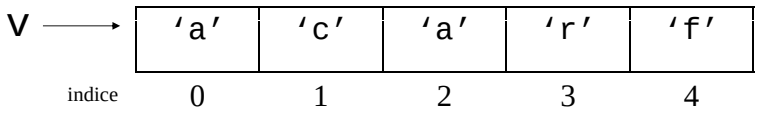
Collezioni

- Per ora vediamo di realizzare questa idea di *dati più collegamenti*: un oggetto (nodo) che contenga, oltre all'informazione sul *dato* memorizzato, anche un *puntatore* al nodo che contiene l'elemento che nella sequenza lo segue.
- Esempio di due nodi:

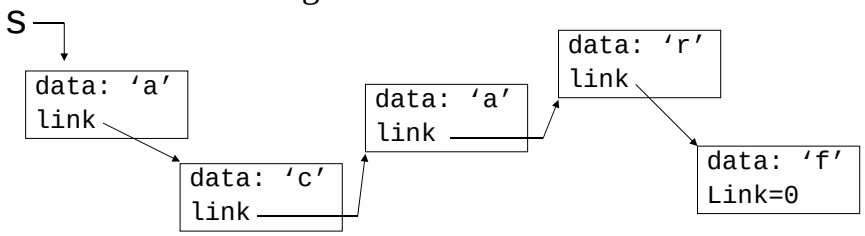


Collezioni

Struttura indicizzata (array)



Struttura collegata



Nodi

- L'idea di *dati più collegamenti* è implementata con una classe `Node`.
- Lo stato di tale classe è descritto da un campo che contiene il dato (per esempio di tipo `float` o `Persona`) e un campo puntatore (`Node *`) al nodo collegato.
- Il comportamento è descritto dai metodi per leggere e scrivere lo stato, oltre al costruttore e distruttore.
- Nell'implementazione che segue il tipo del dato memorizzato nel nodo è `char`.

Nodi: *Node.h*

```
#ifndef NODE_H
#define NODE_H

class Node{
   char data;
   Node *link;

public:
   Node(char d, Node *l=0);
   ~Node();
   char GetData()const;
   Node * GetLink()const;
   void SetData(char d);
   void SetLink(Node *l);
};
#endif
```

La dichiarazione del campo `link` di un nodo può sembrare non corretta. Tuttavia è corretta, ciò che dichiara è un puntatore ad un oggetto di tipo `Node` e non un oggetto stesso.

Nodi: *Node.cpp*

```
#include "Node.h"
#include <iostream>
using namespace std;
Node::Node(char d, Node *l){
   data=d;
   link=l;
   cout<<"Node("<<data<<")"<<endl;
}
Node::~Node(){
   cout<<"~Node("<<data<<")"<<endl;
}
char Node::GetData()const{
   return data;
}
Node * Node::GetLink()const{
   return link;
}
void Node::SetData(char d){
   data=d;
}
void Node::SetLink(Node *l){
   link=l;
}
```

Vista la dichiarazione del campo `link` di un nodo, nel costruttore non si deve allocare un oggetto `Node`, ma solo assegnare un puntatore passato al costruttore.

Nodi: *test.cpp*

```
void test1(){
  Node *n=new Node('A');
  n->SetLink(new Node('B'));

  cout<<n->GetData()<<" ";
  cout<<n->GetLink()->GetData()<<endl;

  delete n->GetLink();
  delete n;
}
```

Si collega l'oggetto nodo `n` ad un altro oggetto nodo attraverso il puntatore a tale nodo .

Attraverso `n` si accede al contenuto del nodo collegato senza il nome simbolico.

Si deve prima distruggere l'oggetto nodo collegato e poi il nodo che contiene il collegamento

Una possibile uscita

```
Node(A)
Node(B)
A B
~Node(B)
~Node(A)
```


Liste: introduzione

- Gli array presentano due limitazioni:
  - Modificare la dimensione dell'array richiede la creazione di un nuovo array e la copia di tutti i dati dall'array originale a quello nuovo.
  - Inserire un elemento nell'array richiede lo scorrimento di dati nell'array.
- Queste limitazioni possono essere superate da *strutture concatenate*: una collezione di nodi che contengono dati e puntatori ad altri nodi. In particolare si fa riferimento alle *liste concatenate* (*linked list*).

Liste: introduzione

- Uno svantaggio delle liste concatenate consiste nell'onerosità dell'accesso ai suoi elementi: l'unico modo per raggiungere un elemento della lista è quello di seguire le connessioni della lista dall'inizio in modo *sequenziale*. Mentre negli array è possibile accedere a qualsiasi elemento attraverso un indice.
- I metodi di manipolazione delle liste sono diversi da quelli visti per gli array.
- In generale, una lista rende più semplice la riorganizzazione degli elementi.

Liste: specifica

- Una possibile interfaccia pubblica (*comportamento*) è la seguente:

```
class LinkedList{
  ...
public:
  LinkedList();
  ~LinkedList();
  bool Add(char x);
  bool Remove(char x);
  bool Contains(char x)const;
  bool IsEmpty()const;
  void Clear();
  void Print() const;
};
```

Inserisce l'oggetto `x` in fondo alla lista.

Cancella la prima occorrenza dell'oggetto `x`.

Verifica se la lista contiene l'oggetto `x`.

Verifica se la lista è logicamente vuota.

Svuota la lista.

Stampa a monitor il contenuto della lista.

Liste: comportamento

Sia `L` un oggetto di tipo `LinkedList`: ecco alcune operazioni

Invocazione metodo	Stato della lista	risultato
`L.IsEmpty();`	[]	true
`L.Add('a');`	[a]	true
`L.Add('b');`	[a b]	true
`L.Add('c');`	[a b c]	true
`L.Contains('b');`	[a b c]	true
`L.Remove('b');`	[a c]	true
`L.Contains('b');`	[a c]	false
`L.IsEmpty();`	[a c]	false

Liste: esempio

- Si scriva una funzione che legge caratteri da tastiera e li memorizza in una lista. L'inserimento è terminato dal carattere `'.'`. Si stampi il contenuto della lista prima e dopo la rimozione di un elemento.

- Utilizzando la lista, non e necessario conoscere il numero di caratteri da memorizzare: viene *allocata* la *giusta* quantità di *elementi*.
- Quando si rimuove un elemento, non e necessario spostare una parte degli elementi rimasti per riempire il vuoto lasciato dalla rimozione (come si farebbe con gli array).

Liste: esempio

```
void test2(){
  LinkedList L;
  char c;
  cin>>c;
  while(c!='.'){
      L.Add(c);
      cin>>c;
  }
  cout<<"---------"<<endl;
  L.Print();
  L.Remove('B');
  L.Print();
  cout<<"---------"<<endl;
}
```

Si utilizza un oggetto di tipo lista conoscendo la sua interfaccia pubblica. Non è necessario sapere come è implementato: la lista è un ADT.

Una possibile uscita

```
ABC.
Node(A)
Node(B)
Node(C)
---------
A B C
~Node(B)
A C
---------
~Node(A)
~Node(C)
```


Liste: implementazione

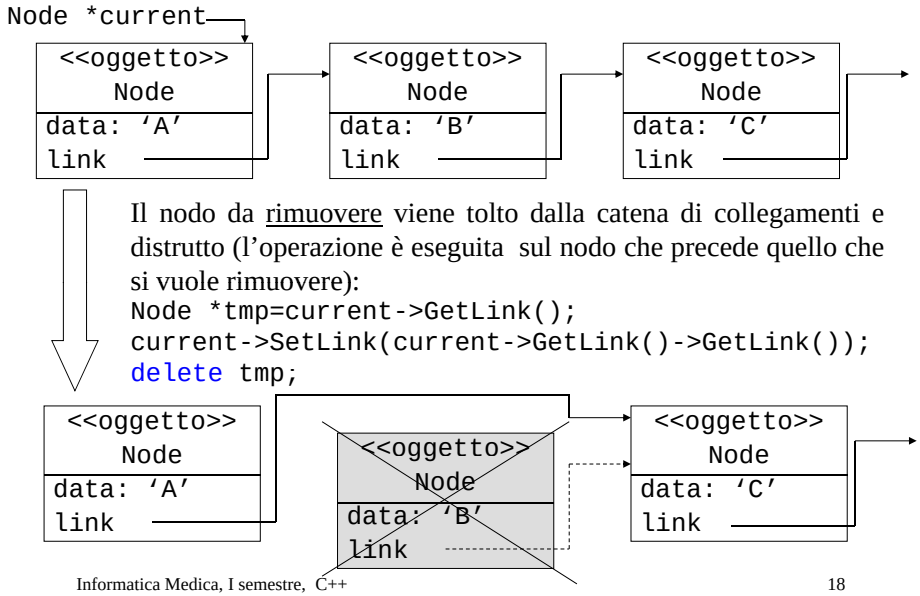
- Una possibile realizzazione della lista consiste nel considerare una sequenza di nodi che contengono un dato e un puntatore al nodo successivo nella sequenza. Tale sequenza è realizzata nella classe `LinkedList` che implementa l'interfaccia mostrata nei lucidi precedenti.
- Un oggetto di tipo `LinkedList` è manipolato utilizzando i metodi resi disponibili dall'interfaccia pubblica (comportamento), non si accede direttamente alla sequenza di nodi. È un *tipo di dato astratto*, ADT.

Nodi: operazioni principali

- Le due principali operazioni da realizzare sulle liste concatenate (sui *nodi* della lista nella fase di implementazione) sono: la *rimozione* di un elemento della lista e l'*inserimento* di un elemento nella lista. Quindi diminuire o aumentare la lunghezza di una lista.

- La semplicità di tali operazioni è la ragione d'essere delle liste concatenate: si modifica solo i puntatori, inoltre solo quelli relativi ai nodi da manipolare.

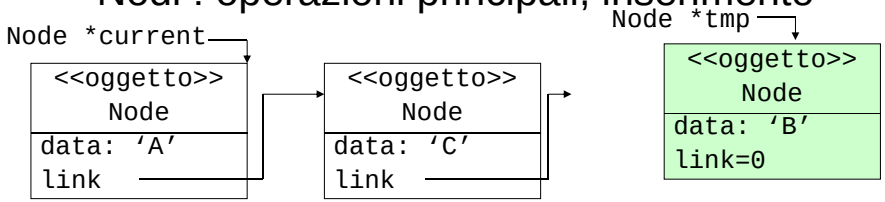
Nodi : operazioni principali, rimozione



Il nodo da rimuovere viene tolto dalla catena di collegamenti e distrutto (l'operazione è eseguita sul nodo che precede quello che si vuole rimuovere):

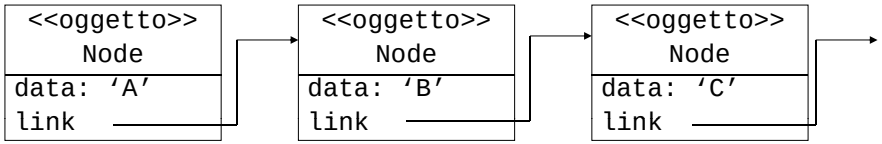
```
Node *tmp=current->GetLink();
current->SetLink(current->GetLink()->GetLink());
delete tmp;
```


Nodi : operazioni principali, inserimento



Il nodo da inserire viene messo nella catena di collegamenti (l'operazione è fatta sul nodo che si vuole inserire e sul nodo che precede la posizione di inserimento):

```
tmp->SetLink(current->GetLink());
current->SetLink(tmp);
```



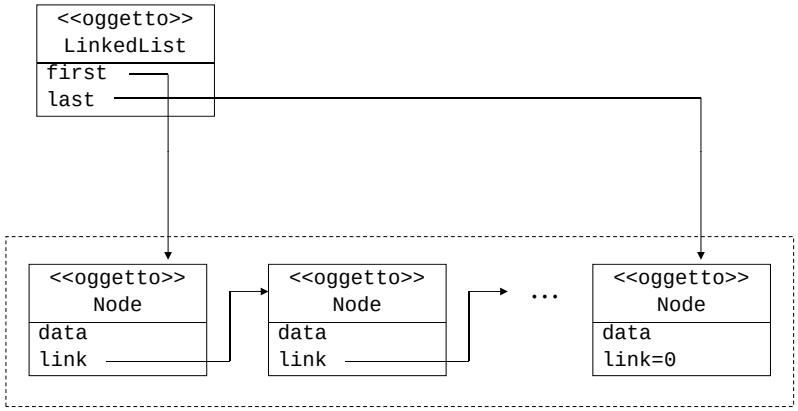
Liste: implementazione

- Vediamo una possibile implementazione della lista.

- Si usa un campo `first` che mantiene il puntatore al primo nodo della lista e un campo `last` che mantiene un puntatore all'ultimo nodo.
- L'ultimo nodo è caratterizzato dal campo `link` con valore `0`.
- Il costruttore crea una lista vuota: cioè i due campi puntatori hanno valore `0`.
- Il distruttore deve deallocare i nodi ancora presenti nella lista.

Liste: implementazione

- Una possibile rappresentazione grafica:



Liste: *LinkedList.h*

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include "Node.h"
class LinkedList{
   Node * first;
   Node * last;
public:
   LinkedList();
   ~LinkedList();
   bool Add(char x);
   bool Remove(char x);
   bool Contains(char x)const;
   bool IsEmpty()const;
   void Clear();
   void Print() const;
};
#endif
```


Liste: *LinkedList.cpp*

```
#include "Node.h"
#include "LinkedList.h"
#include <iostream>
using namespace std;

LinkedList::LinkedList(){
  first=last=0;
}

LinkedList::~LinkedList(){
  Clear();
}
```

La lista viene creata vuota: non esiste nessun nodo collegato.

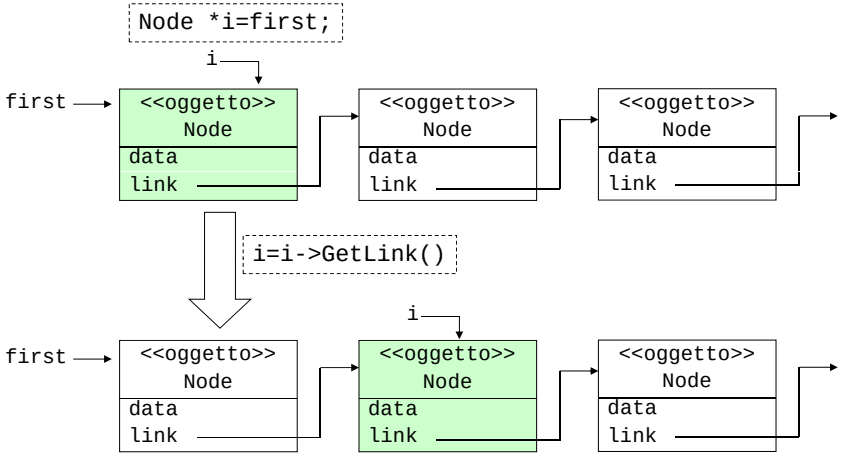
Il distruttore deve deallocare i nodi ancora presenti nella lista, quindi deve eseguire un `Clear()`. In tal modo si fattorizza il codice, cioè non si duplica esplicitamente il codice ma si richiama il metodo opportuno.

Liste: *LinkedList.cpp*

```
bool LinkedList::Contains(char x)const
{
  for(Node *i=first; i!=0; i=i->GetLink())
     if (i->GetData()==x)
          return true;
  return false;
}
```

Per verificare la presenza di un dato oggetto `x` nella lista, si devono esaminare i nodi della lista. Si può fare con un ciclo `for`: si definisce il contatore `i` come un puntatore di tipo `Node*` a cui si assegna il puntatore al primo nodo. Si esce dal ciclo quando si finiscono i nodi, cioè il `link` vale `0`. L'incremento si ottiene con l'assegnamento `i=i->GetLink()`, che fa puntare il contatore `i` al nodo collegato.

Liste: ricerca



Liste: *LinkedList.cpp*

```
bool LinkedList::IsEmpty()const{
   return first==0;
}
```

La lista è vuota se il campo che punta al primo nodo vale `0`

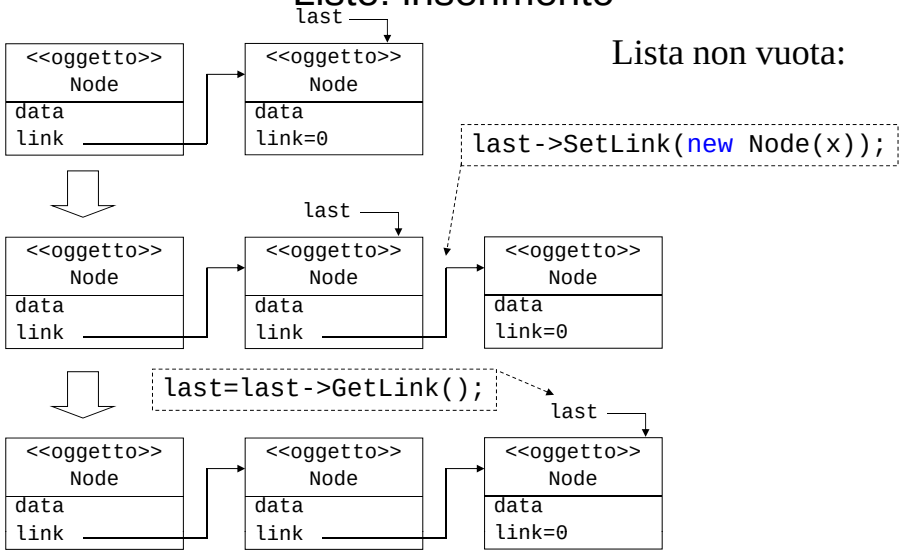
```
bool LinkedList::Add(char x)
{
   if (IsEmpty())
       first=last=new Node(x);
   else{
       last->SetLink(new Node(x));
       last=last->GetLink();
   }
   return true;
}
```

Lista vuota: modifico entrambi i campi

Lista non vuota: modifico solo in campo che punta all'ultimo nodo

Liste: inserimento

Lista non vuota:



Liste: prestazioni

- Aggiungere un nodo alla lista ha un costo computazionale costante *O(1)*: è indipendente dal numero di nodi nella lista. Non si devono copiare gli elementi in una nuova struttura, come nel caso di array pieni.
- Verificare se la lista contiene un dato oggetto implica l'uso di un ciclo `for`. Si consideri il caso medio: se ogni nodo della lista ha la stessa probabilità di contenere il dato, allora si ha una iterazione per verificare il primo nodo, due iterazioni per verificare il secondo nodo, e in una sequenza lunga `n` si ha in media

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

Liste: *LinkedList.cpp*

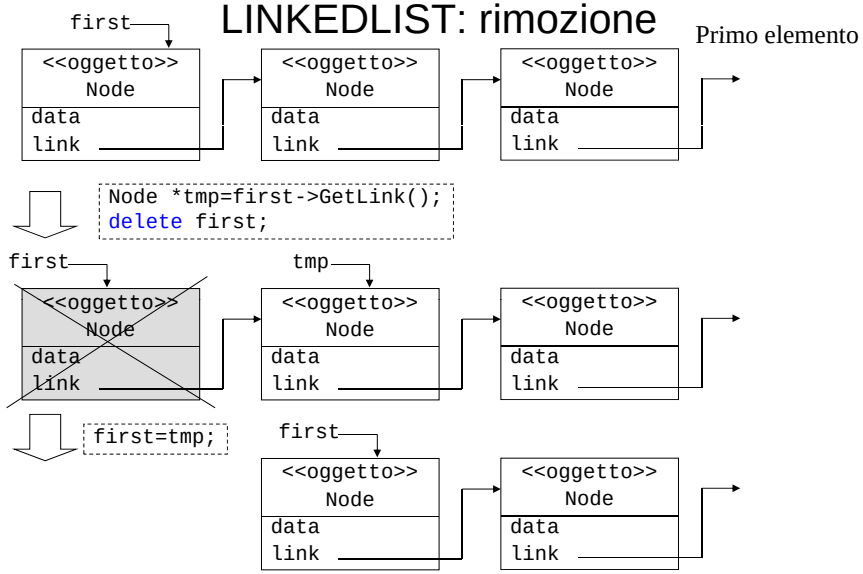
```
bool LinkedList::Remove(char x)
{
   if (IsEmpty()) return false;
   if (first->GetData()==x)
   {
        if (first==last) last=0;
        Node *tmp=first->GetLink();
        delete first;
        first=tmp;
        return true;
   }
   for (Node *i=first; i->GetLink()!=0; i=i->GetLink())
   {
        if (i->GetLink()->GetData()==x)
        {
               if (i->GetLink()==last)
                      last=i;
               Node *tmp=i->GetLink();
               i->SetLink(i->GetLink()->GetLink());
               delete tmp;
               return true;
        }
   }
   return false;
}
```

L'operazione di rimozione è eseguita sul nodo che precede quello che si vuole rimuovere

Controllo il primo elemento

Se non è il primo elemento

LINKEDLIST: rimozione



Liste: prestazioni

- Considerazioni analoghe a quelle sviluppate per calcolare la complessità computazionale media del metodo per la ricerca di un dato valgono anche per il metodo di rimozione di un dato.
- Quindi il metodo `Remove()` ha in media un costo lineare, cioè *O(n)*.

Liste: *LinkedList.cpp*

```
void LinkedList::Clear()
{
   Node *i=first;
   while(i!=0)
   {
        Node *tmp=i->GetLink();
        delete i;
        i=tmp;
   }
   first=0;
   last=0;
}


void LinkedList::Print()const
{
   for(Node *i=first; i!=0; i=i->GetLink())
        cout<<i->GetData()<<" ";
   cout<<endl;
}
```

Per svuotare la lista si devono deallocare tutti i nodi presenti e aggiornare lo stato.

Liste: considerazioni

- Se è necessario un *accesso immediato* a ciascun elemento, allora l'array è la scelta migliore.
- Se si accede frequentemente solo ad alcuni elementi (i primi, gli ultimi) e se la *modifica* della struttura è il cuore di un algoritmo, allora la lista concatenata è la scelta migliore.
- Un vantaggio delle liste è che usano la *quantità minima di memoria* necessaria e sono facilmente ridimensionabili. Al contrario un array può essere modificato in dimensione solo di quantità fisse (per esempio dimezzato o raddoppiato).

Liste doppiamente concatenate

- Esistono altri tipi di lista, vediamone alcuni.

- La lista descritta è di tipo semplicemente concatenata: i nodi hanno informazione solo sul loro successore, per cui non si ha accesso al loro predecessore.
- Si può definire una *lista doppiamente concatenata* in modo che ogni nodo abbia due campi riferimento: uno al successore e uno al predecessore.
- In tal modo è possibile percorrere la lista nei due sensi.

Liste con salti

- Per trovare un elemento in una lista è necessaria uno *scansione sequenziale*. Anche se la lista è *ordinata* è sempre necessaria una scansione sequenziale.
- Si evita questo problema utilizzando una *lista con salti*: liste che consentano di saltare alcuni nodi per evitare manipolazioni sequenziali.
- La struttura interna è più complessa: nodi con un numero di campi riferimento diverso.