UNIVERSITÀ DEGLI STUDI DI PARMA

Corso di Laurea in

"Ingegneria dei Sistemi Informativi"

Programmazione di Applicazioni Software

Strutture dati dinamiche lineari

Andrea Prati

Strutture dati dinamiche lineari

- una struttura dati si definisce **dinamica** se permette di rappresentare insiemi dinamici la cui **cardinalità varia** durante l'esecuzione del programma
- una struttura dati si definisce **lineare** se ogni elemento contiene solo il riferimento all'elemento **successivo** e l'**accesso** agli elementi avviene seguendo specifiche modalità partendo sempre dal **primo elemento**
- strutture dinamiche lineari
  - **lista (list)**
  - **pila (stack)**
  - **coda (queue)**

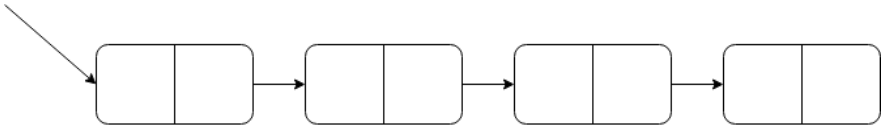
STRUTTURA DATI DINAMICA LINEARE: LISTA

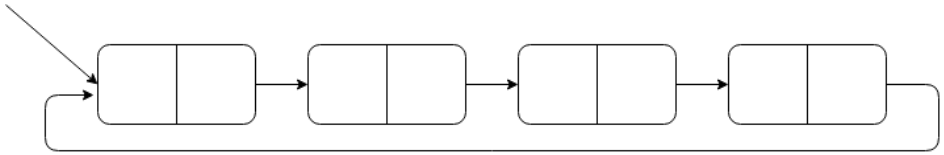
Lista - definizione

- si dice lista una **tripla L = (E, t, S)** dove
  - **E** è un **insieme di elementi**
  - **t** $\in$ **E** è detto **testa**
  - **S** è una **relazione binaria** su E **(S** $\subseteq$ **E** $\times$ **E)**
- la relazione **S** soddisfa le seguenti proprietà
  - $\forall$ **e** $\in$ **E, (e,t)** $\notin$ **S**
  - $\forall$ **e** $\in$ **E,** se **e** $\neq$ **t** allora esiste **uno e un solo e'** $\in$ **E** tale che **(e', e)** $\in$ **S**
  - $\forall$ **e** $\in$ **E** esiste **al più** un **e'** $\in$ **E** tale che **(e, e')** $\in$ **S**
  - $\forall$ **e** $\in$ **E,** se **e** $\neq$ **t** allora **e** è **raggiungibile** da **t,** cioè esistono $\mathbf{e'_1, \ldots, e'_k} \in \mathbf{E}$ con $k \geq 2$ tali che $\mathbf{e'_1 = t}$, $\mathbf{(e'_i, e'_{i+1})} \in \mathbf{S}$ per ogni $1 \leq i \leq k-1$, ed $\mathbf{e'_k = e}$

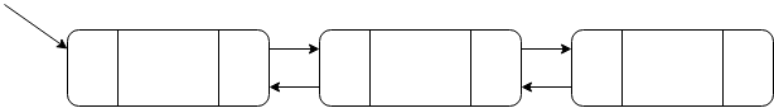
Lista - rappresentazione

- una lista viene rappresentata come una **struttura dati dinamica lineare**, in cui **ogni elemento** contiene solo il **riferimento all'elemento successivo** (lista singolarmente collegata)





- se ogni elemento contiene **anche il riferimento all'elemento precedente** (lista doppiamente collegata) la struttura è dinamica ma non lineare



Lista ordinata

- una lista **L = (E, t, S)** è detta **ordinata**
  - se le **chiavi** contenute nei suoi elementi sono disposte in modo da soddisfare una **relazione d'ordine totale**
  - $\forall$ $\mathbf{e_1}$, $\mathbf{e_2} \in \mathbf{E}$, se $\mathbf{(e_1, e_2)} \in \mathbf{S}$ allora la chiave di $\mathbf{e_1}$ **precede** quella di $\mathbf{e_2}$ nella relazione d'ordine totale

Lista - caratteristiche

- il **link** dell'elemento successivo contenuto nell'**ultimo** elemento di una lista è **indefinito**, così come l'indirizzo dell'elemento precedente contenuto nel primo elemento di una lista doppiamente collegata
- fa eccezione il caso dell'implementazione **circolare** di una lista, nella quale l'ultimo elemento è collegato al primo elemento
- gli **elementi** di una lista **non** sono necessariamente **memorizzati in modo consecutivo**, quindi l'**accesso** ad un qualsiasi elemento avviene scorrendo tutti gli elementi che lo precedono (*struttura sequenziale*)
- l'accesso indiretto necessita di un **riferimento** al primo elemento della lista, detto **testa**, il quale è indefinito se e solo se la lista è vuota

Liste - algoritmi

- **visita**:
  - data una lista, attraversare **tutti** i suoi **elementi** esattamente **una volta**
- **ricerca**:
  - dati una **lista** e un **valore**, stabilire se il valore è **contenuto** in un elemento della lista, riportando in caso affermativo l'indirizzo di tale elemento
- **inserimento**:
  - dati una **lista** e un **valore**, **inserire** (se possibile) nella posizione appropriata della lista un **nuovo elemento** in cui memorizzare il valore
- **rimozione**:
  - dati una **lista** e un **valore**, **rimuovere** (se esiste) l'**elemento** appropriato della lista **che contiene il valore**

Nodo: elemento della lista

```
class Nodo {
private:
   string info;
   Nodo* next;
public:
   Nodo (string s) :
         info(s), next(nullptr) {
};
   string getInfo() const {return
info;};
   Nodo* getNext() const { return
next; }

   friend class Lista;
};
```

- esempio in cui l'**informazione** associata a un nodo (`info`) è una stringa
- il **link** al nodo successivo (`*next`) è un puntatore a un nodo
- `Lista` è definita come classe ***friend*** per favorire l'accesso ai membri privati di `Nodo`

Lista - definizione

```
class Lista {
private:
   Nodo* testa;
public:
   Lista();
   ~Lista();
   void insTesta(const string
&value);
   void insCoda(const string
&value);
   bool elimTesta(string &value);
   bool elimCoda(string &value);
   bool vuota() const;

friend std::ostream & operator<<
(std::ostream & os, const Lista &
lst);
};
```

- **inserimento** di elementi
  - in testa - **`insTesta(&string)`**
  - in coda - **`insCoda(&string)`**
- **eliminazione** di elementi
  - in testa - **`elimTesta(&string)`**
  - in coda – **`elimCoda(&string)`**
- controllo se la lista è **vuota**
  - **`vuota()`**
- **operatore <<**
  - inserimento in stream
  - funzione ***friend***

Lista - implementazione

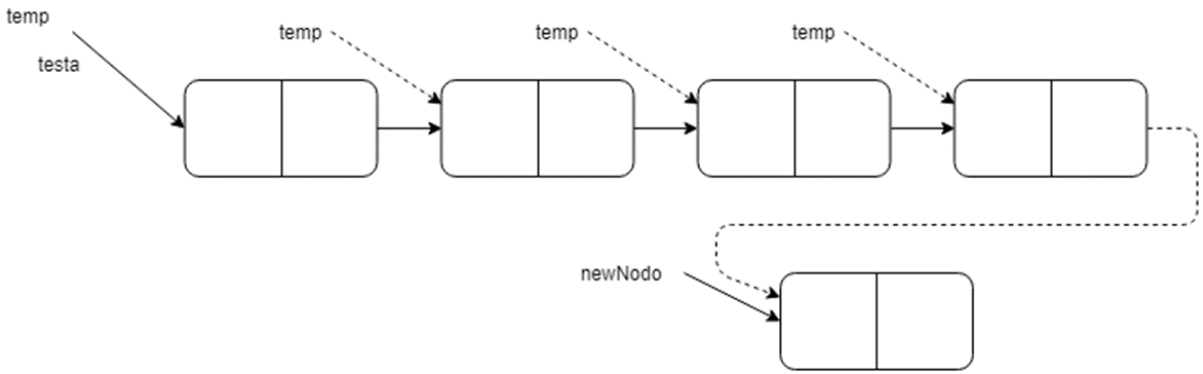
```
Lista::Lista() { testa = nullptr; }

Lista::~Lista() {
   while (testa) {
      Nodo* temp = testa;
      testa = testa->next;
      delete temp;
   }
}

bool Lista::vuota() const {
   return testa == nullptr;
}
```

- **`testa`** è il **link al primo elemento** della lista
  - **`testa`** è il **link iniziale** di tutte le operazioni
- il distruttore **dealloca** la memoria di tutti i nodi della lista

Lista – esempio inserimento

```
void Lista::insCoda(const string
&val) {
   Nodo* newNodo = new Nodo(val);
   if (vuota()) {
      testa = newNodo;
   }
   else {
      Nodo* temp = testa;
      while(temp->next)
          temp = temp->next;
      temp->next = newNodo;
   }
}
```

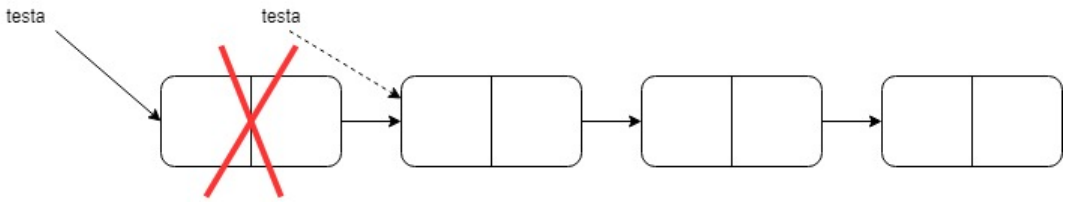
- **inserimento** di un nodo con valore ricevuto come parametro in **coda** alla lista
  - **`insCoda(&string)`**
  - **`temp`** punterà all'ultimo nodo della lista
- analogo discorso per inserimento in testa
  - non è necessario scorrere tutti gli elementi



Lista – esempio eliminazione

```
bool Lista::elimTesta(string &val)
{
   if (vuota()) return false;
   val = testa->info;
   Nodo* temp = testa;
   testa = testa->next;
   delete temp;   // deallocazione
   return true;
}
```

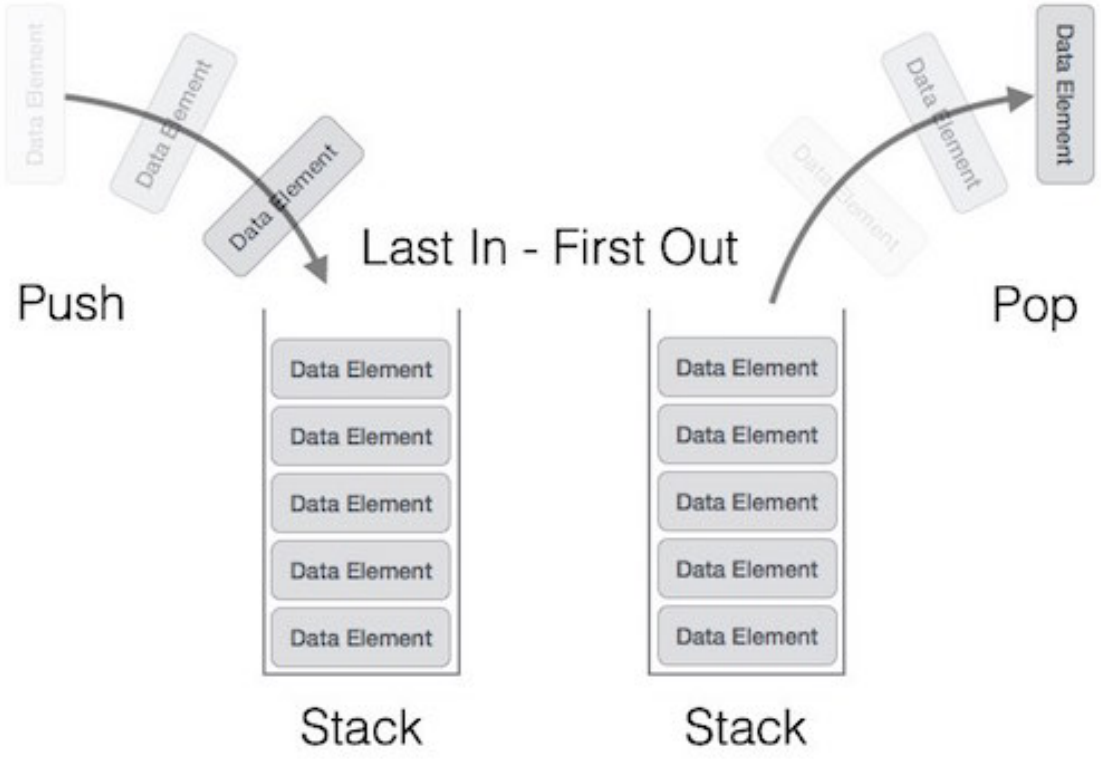
- **eliminazione** di un valore in **testa** alla lista
  - **`elimTesta(&string)`**
  - **`false`** se la lista è vuota
  - restituzione del valore nel parametro
- analogo discorso per eliminazione in coda
  - è necessario scorrere tutti gli elementi



STRUTTURA DATI DINAMICA LINEARE: PILA (STACK)

Pila - stack

- una **pila** è una lista gestita in base al principio **LIFO (last in, first out)**
- gli **inserimenti** (*push*) e le **rimozioni** (*pop*) avvengono nella stessa estremità della lista



Stack

```
class Stack {
private:
   Nodo* top;
public:
   Stack();
   ~Stack();
   void push(const std::string
& value);
   bool pop(std::string &
value);
   bool empty() const;

   friend std::ostream &
operator<< (std::ostream & os,
const Stack & st);
};
```

```
Stack::Stack() : top(nullptr) {
}

Stack::~Stack() {
   while (top) {
      Nodo* temp = top;
      top = top->next;
      delete temp;
   }
}

bool Stack::empty() const {
   return top == nullptr;
}
```

Stack – push e pop

```
void Stack::push(const string & value) {
  Nodo * newNodo = new Nodo(value);
  newNodo->next = top;
  top = newNodo;
}
```

- ***push*** equivale all'**inserimento in testa** alla lista

```
bool Stack::pop(string & value)
{
 if (empty()) return false;
 value = top->info;
 Nodo * temp = top;
 top = top->next;
 delete temp;  // deallocazione
 return true;
}
```

- ***pop*** equivale alla **eliminazione in testa** alla lista

Stack – implementazione con array dinamico

```
class Stack {
private:
  string * dati;
  int top;
  int capacita;
  int incremento;
public:
   Stack(int capacita = 10,
         int incremento = 10);
   ~Stack();
   void push(const string & val);
   bool pop(string & val);
   bool empty() const;

friend std::ostream & operator<<
    (std::ostream & os, const Stack
& s);
};
```

- **`dati`** è un *array dinamico* (in questo caso di stringhe) che contiene i dati inseriti nello stack
- **`top`** è l'indice dell'ultimo elementi inserito nello stack (-1 se lo stack è vuoto)
- **`capacita`** è la capacità dell'array (aumenta se necessario)
- **`push, pop, empty`** sono le funzioni per la gestione dello stack
- push può provocare la creazione di un nuovo array

Stack – implementazione con array dinamico

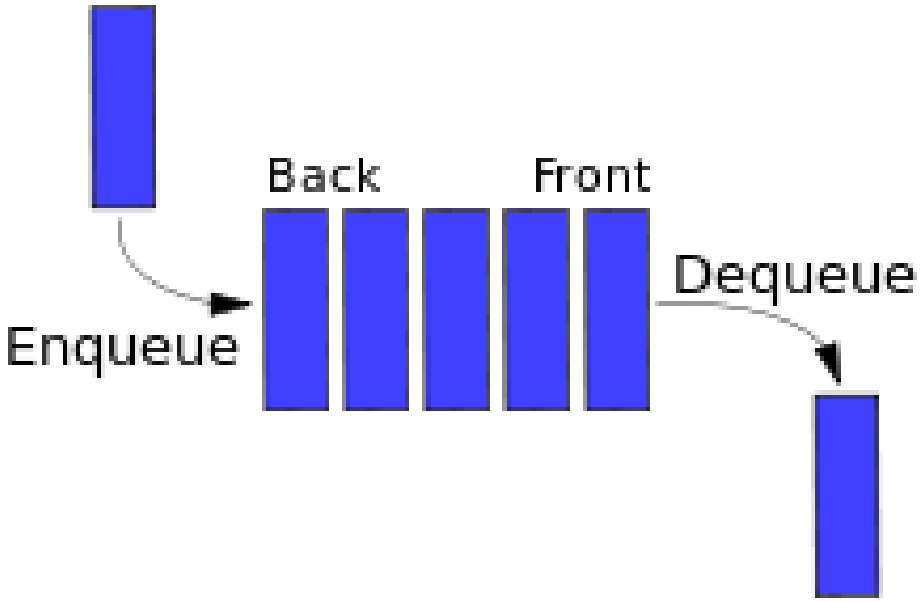
```
Stack::Stack(int cap, int inc) :
   capacita(cap), incremento(inc) {
   dati = new string[capacita];
   top = -1;
}

Stack::~Stack() { delete[] dati; }

bool Stack::empty() const {
   return top < 0;
}

bool Stack::pop(string & val) {
   if (empty()) return false;
   val = dati[top];
   top--;
   return true;
}
```

```
void Stack::push(const string &
val) {
   if (top >= (capacita - 1)) {
      capacita += incremento;
      string * newDati =
               new
string[capacita];
      for (int i = 0; i <= top;
++i)
         newDati[i] = dati[i];
      delete[] dati;
      dati = newDati;
   }
   top++;
   dati[top] = val;
}
```

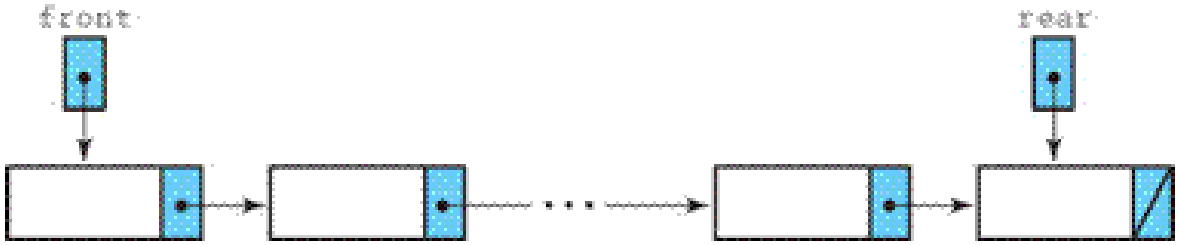
STRUTTURA DATI DINAMICA LINEARE: CODA (QUEUE)

Coda - queue

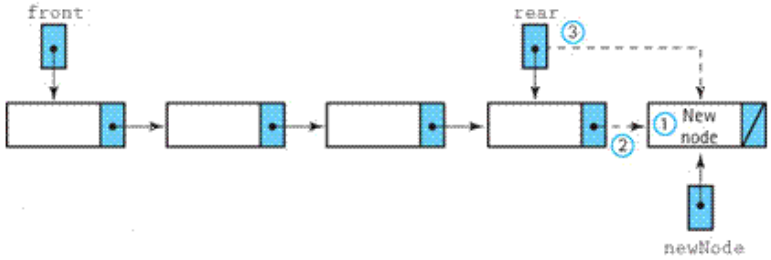
- una **coda** è una lista gestita in base al principio **FIFO (first in, first out)**
- gli **inserimenti** (*enqueue*) e le **rimozioni** (*dequeue*) avvengono nelle estremità opposte della lista



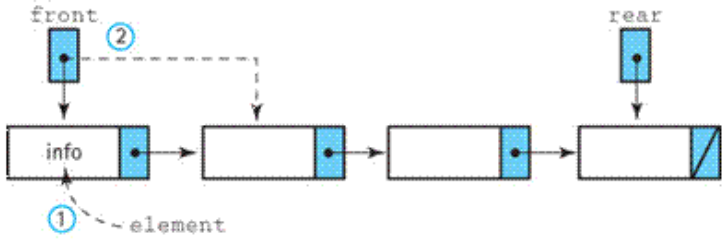
Coda: implementazione con lista



front (head) – rear (tail)



enqueue



dequeue

Queue

```
class Queue {
    public:
        Queue();
        ~Queue();
        void enqueue(const string
&value);
        bool dequeue(string
&value);
        bool empty() const;
    private:
        Nodo* head;
        Nodo* tail;

friend std::ostream & operator<<
(std::ostream & os, const Queue &
q);
};
```

```
void Queue::enqueue(const string
&value) {
    Nodo* t = tail;
    tail = new Nodo(value);;
    if (empty())
        head = tail;
    else
        t->next = tail;
}

bool Queue::dequeue(string &value)
{
    if (empty()) return false;
    Nodo* p = head;
    head = head->getNext();
    if (empty()) tail = nullptr;
    value = p->getInfo();
    delete p;
    return true;
}
```

OVERLOADING DEGLI OPERATORI

Operatori

- gli **operatori** + , - , == , <<, >> sono **funzioni** usate con una sintassi particolare
- C++ consente di **sovraccaricare** gli **operatori** facendo in modo che accettino argomenti di tipo classe
  - è una delle funzionalità tra le più apprezzate del linguaggio
  - rende il programma molto più chiaro rispetto a chiamate a funzione equivalenti
- l'**oggetto più a sinistra** deve essere **membro** della classe
- **non sempre** è possibile (es.: operatori >> e <<)

Un esempio

```
Lista* Lista::operator+(Lista
altraLista) {
 Lista* newList = new Lista();
 Nodo* t = testa;
 while (t!=nullptr) {
  newList->insCoda(t->info);
  t=t->next;
 }
 t=altraLista.testa;
 while (t!=nullptr) {
  newList->insCoda(t->info);
  t=t->next;
 }
 return newList;
}
```

- **concatenazione** fra liste
- viene restituito un puntatore a una **nuova lista** che contiene le **informazioni** presenti nella **lista attuale** seguite da quelle presenti nella lista **altraLista** ricevuta come **parametro**
- utilizzo:
  - `Lista l1,l2;`
  - …
  - …
  - `Lista* l3;`
  - `l3 = l1 + l2;`

Esempio

```
std::ostream & operator<<
(std::ostream & os, const Lista &
lst) {
   os << '{';
   if (!lst.vuota()) {
      Nodo * p = lst.testa;
      while (p) {
         os << p->getInfo();
         if (p->getNext() !=
nullptr)
                os << ',';
         p = p->getNext();
      }
   }
   os << '}';
   return os;
}
```

- gli **operatori <<** e **>>** possono essere sovraccaricati per essere usati per l'**I/O** degli **oggetti** di una classe
- **non** possono essere sovraccaricati come **membri**: l'operatore più a sinistra non è del tipo della classe
- **<<** e **>>** richiedono rispettivamente ***ostream&*** e ***istream&***
- nell'esempio l'**overloading** dell'**operatore <<** viene definito come **funzione friend** di lista