UNIVERSITÀ
DI PARMA

abstract data type
strutture dati dinamiche lineari

Alberto Ferrari

- 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

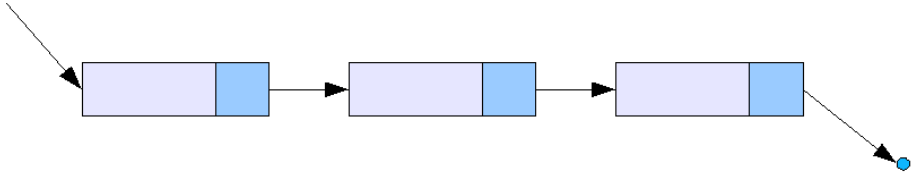
lista

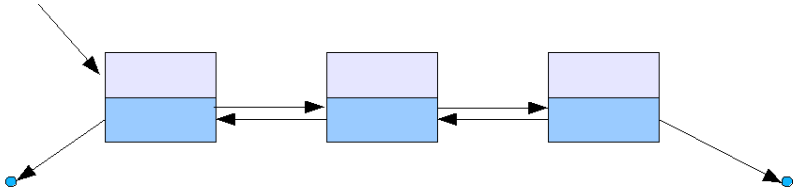
lista - definizione

- si dice lista una ***tripla*** **L = (E, t, *S*)** dove
  - **E** è un insieme di ***elementi***
  - **t ∈ 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 ∈ E, (e,t) ∉ *S***
  - $\forall$ **e ∈ E**, se **e ≠ t** allora esiste ***uno e un solo*** **e' ∈ E** tale che **(e', e) ∈ *S***
  - $\forall$ **e ∈ E** esiste ***al più*** un **e' ∈ E** tale che **(e, e') ∈ *S***
  - $\forall$ **e ∈ E**, se **e ≠ t** allora e è ***raggiungibile*** da **t**, cioè esistono $e'_1, \dots, e'_k \in E$ con $k \geq 2$ tali che $e'_1 = t$, $(e'_i, e'_{i+1}) \in S$ per ogni $1 \leq i \leq k-1$, ed $e'_k = e$

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***
  - $\forall$ $\mathbf{e_1, e_2} \in \mathbf{E}$, 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 {
  public:
    Nodo();
    Nodo(std::string s);
    Nodo(string s, Nodo* n);
    virtual ~Nodo();
    std::string getInfo() { return info; }
    void setInfo(string val) { info=val; }
    Nodo* getNext() { return next; }
    void setNext(Nodo* v) { next = v; }
  private:
    string info;
    Nodo* next;
};

typedef Nodo* link;
```

- 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
- **`link`** definito come alias a **`Nodo*`**

lista - definizione

```
class Lista {
    public:
        Lista();
        Lista(link t);
        virtual ~Lista();
        void insTesta(link);
        link elimTesta();
        void insCoda(link);
        link elimCoda();
        link elimina(int);
        void inserisci(link,int);
        void stampa();
        void elimina();
    private:
        link testa;
};
```

- ***inserimento*** di elementi
  - in testa - **`insTesta(link)`**
  - in coda - **`insCoda(link)`**
  - in posizione specifica - **`inserisci(link,int)`**
- ***eliminazione*** di elementi
  - in testa - **`link elimTesta()`**
  - in coda – **`link elimCoda()`**
  - in posizione specifica – **`link elimina(int)`**
- ***visualizzazione*** di tutti gli elementi
  - **`stampa()`**
- ***eliminazione*** di tutti gli elementi
  - **`elimina()`**

lista - implementazione

```
Lista::Lista() { testa = nullptr; }
…
void Lista::insTesta(link n) {
    n->setNext(testa);
    testa = n;
}
…
void Lista::stampa() {
    link p = testa;
    while(p) {
        std::cout << p->getInfo() << " ";
        p = p->getNext();
    }
    std::cout << std::endl;
}
…
```

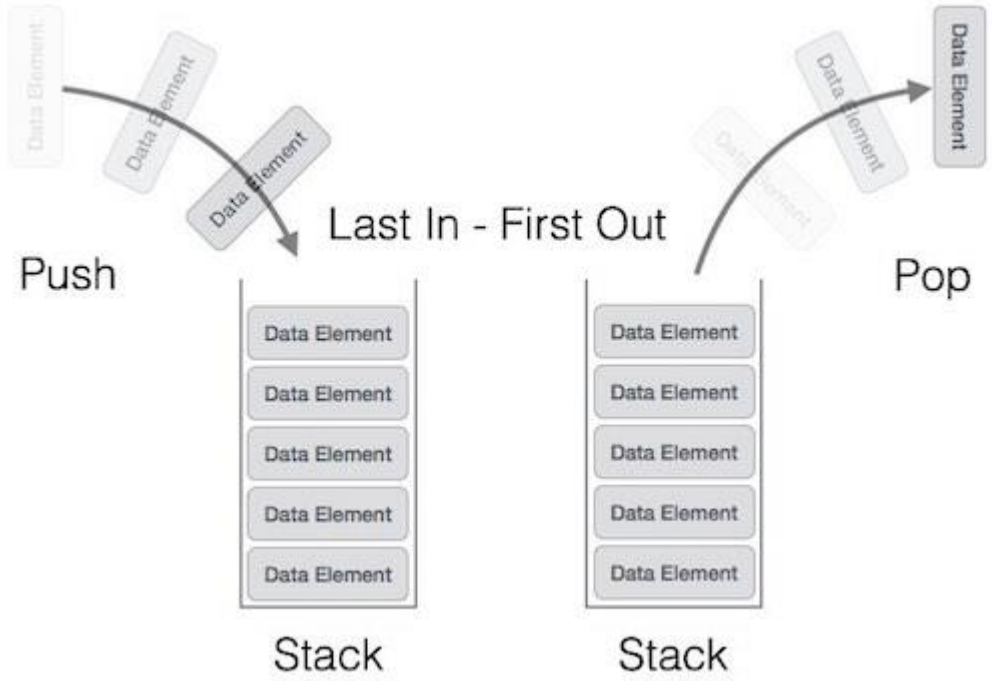
- **`testa`** è il ***link al primo elemento*** della lista
- tutte le operazioni ***accedono*** agli elementi tramite ***link***
- **`testa`** è il ***link iniziale*** di tutte le operazioni

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



```
class Stack
{
    public:
        Stack();
        virtual ~Stack();
        void push(link);
        link pop();
        bool empty();
    private:
        link top;
};
```

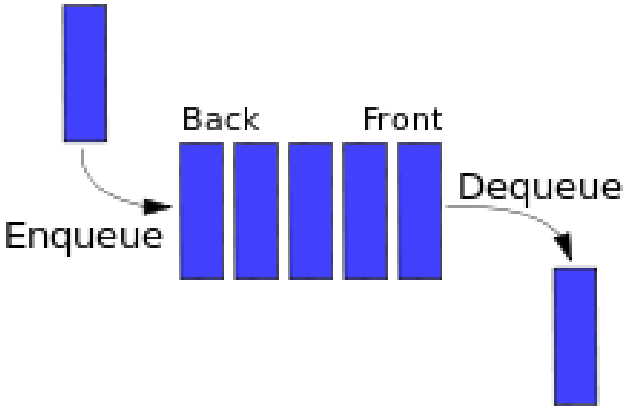
```
Stack::Stack(): top(nullptr) {}
void Stack::push(link n) {
    n->setNext(top);
    top = n;
}
link Stack::pop() {
    if (empty())
        return nullptr;
    link t = top;
    top = top->getNext();
    return t;
}
bool Stack::empty(){
    if (top) return false;
    return true;
}
```

struttura dati dinamica lineare

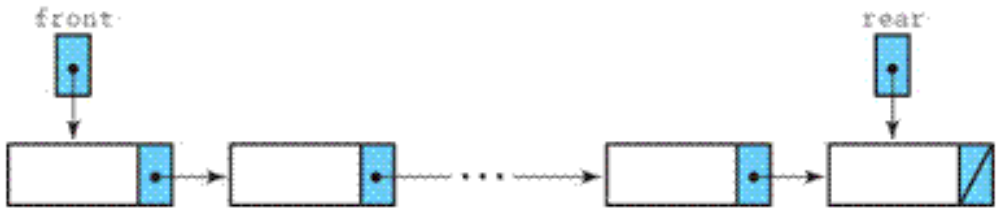
coda (queue)

coda (queue)

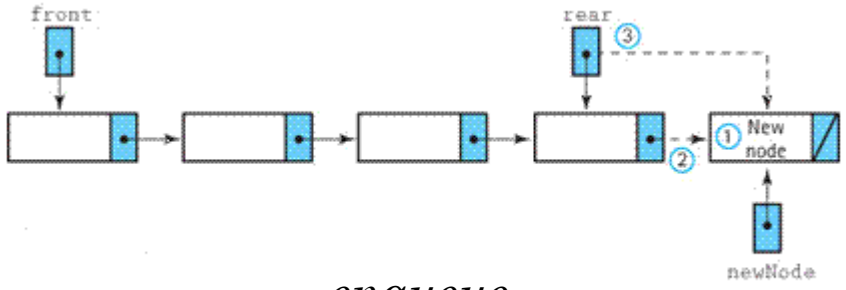
- una ***coda*** è una lista gestita in base al principio ***FIFO*** (***first in, first out***)
- gli ***inserimenti*** (*enqueue*) e le ***rimozioni*** (*dequeue*) avvengono nelle estremità opposte della lista



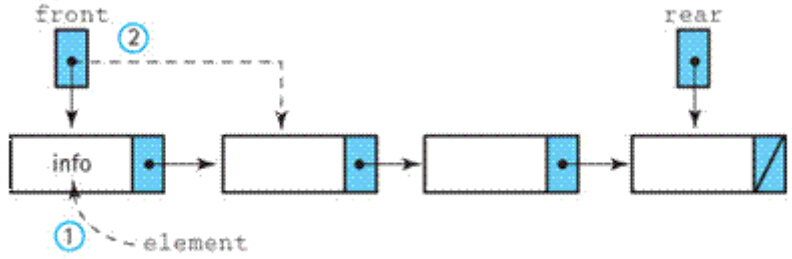
coda: implementazione con lista



front (head) – rear (tail)



enqueue



dequeue

```
class Queue
{
    public:
        Queue();
        virtual ~Queue();
        void enqueu(link);
        link dequeue();
        bool empty();

    private:
        link head;
        link tail;
};
```

```
void Queue::enqueu(link p) {
    link t = tail;
    tail = p;
    if (empty())
        head = tail;
    else
        t->setNext(tail);
}
link Queue::dequeue() {
    if (empty()) return nullptr;
    link p = head;
    head = head->getNext();
    if (empty()) tail = nullptr;
    p->setNext(nullptr);
    return p;
}
```

C++

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 tail) {
 Lista newList;
 link t = testa;
 while (t!=nullptr) {
  newList.insCoda(new Nodo(t->getInfo()));
  t=t->getNext();
 }
 t=tail.testa;
 while (t!=nullptr) {
  newList.insCoda(new Nodo(t->getInfo()));
  t=t->getNext();
 }
 return newList;
}
```

- ***concatenazione*** fra liste
- viene restituita una ***nuova lista*** che contiene le ***informazioni*** presenti nella ***lista attuale*** seguite da quelle presenti nella lista ***tail*** ricevuta come ***parametro***
- utilizzo:

```
Lista l1,l2,l3;
…
…
l3 = l1 + l2;
```

C++

funzioni friend

funzioni friend

- una funzione ***friend*** di una classe ha ***accesso*** ai ***membri privati*** della classe ***pur non essendone membro***
- deve essere ***dichiarata friend*** nella ***definizione*** della classe
- viene ***definita*** e ***chiamata*** come una ***funzione ordinaria***
- l'uso di funzioni friend migliora le prestazioni
  - *non necessitano di accessor*
- una ***funzione*** può essere ***friend di più classi***
- le funzioni friend ***più comuni*** sono gli ***operatori*** sovraccaricati

esempio

```
class Lista {
 public:
  …
  friend std::ostream & operator<<
      ( std::ostream &out, Lista &lis);
 private:
  link testa;
};
std::ostream & operator<<
    ( std::ostream &out, Lista &lis) {
    link t = lis.testa;
    while (t!=nullptr) {
        out << t->getInfo() << " -> ";
        t = t->getNext();
    }
    out << "// " << std::endl;
}
```

- 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