Inf Gen II

INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento

Marco Roveri

roveri@irst.itc.it

Tipi dato astratto

AA 2005/2006

MR

Abstract Data Type

- **Definizione**: un ADT (*Abstract Data Type*) è un tipo di dato (un insieme di valori e una collezione di operazioni su questi valori) accessibili ***solo*** attraverso una interfaccia, nota con il nome di *metodi*.
- Il C++ mette a disposizione dei costrutti particolari per definire gli ADT, le ***classi***, che vedrete nel prossimo corso.
- Noi vedremo come realizzare questi tipi di dati astratti per mezzo di strutture.

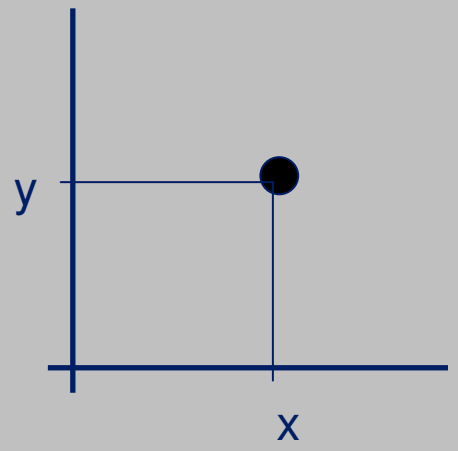
Esempio di dato astratto

- Consideriamo la definizione di un tipo di dato astratto che rappresenta un punto nello spazio cartesiano XxY.
- Le operazioni che vogliamo effettuare su un punto (indipendentemente da come viene implementato) sono:
  - Crea un nuovo punto.
  - Ritorna la coordinata x (y) rispettivamente come double.
  - Assegna la coordinata x (y) rispettivamente.
  - Confronta due punti per vedere se sono uguali o diversi.
  - Stampa le coordinate di un punto.
  - Calcola la distanza tra due punti.
  - Somma due punti.
  - Verifica se tre punti stanno su una retta.

Esempio: tipo di dato astratto punto

Punto.h

```
// Definizione dell'ADT Point
typedef struct Point_ Point;
typedef struct Point _* PointPtr;

// Definizione dei metodi dell'ADT Point
Point Point(void);
Point Point(const double x, const double y);
double Point_GetX(const PointPtr p);
double Point_GetY(const PointPtr p);
void Point_SetX(const PointPtr p);
void Point_SetY(const PointPtr p);
bool Point_Equal(const PointPtr P1, const PointPtr P2);
void Point_Print(const PointPtr P, const char * n);
double Point_GetDistance(const PointPtr P1, const PointPtr P2);
PointPtr Point_Sum(const PointPtr P1, const PointPtr P2);
bool Point_Aligned(const PointPtr P1, const PointPtr P2,
                                      const PointPtr P3);
```



Esempio: tipo di dato astratto punto

mainPoint.cpp

```
#include <iostream>
using namespace std;
#include “Point.h”

int main() {
  double t;
  PointPtr P2 = new Point;
  PointPtr P1 = new Point(5.0, 5.0);
  PointPtr P3;

  Point_Print(P1, “Coordinate del Punto P1”);
  cout << “Inserire coordinate di un Punto P2” << endl << “X = “ ;
  cin >> t;
  Point_SetX(P1, t);
  cout << “Y = “; cin >> t;
  Point_SetY(P1, t);
  cout << “La distanza tra P1 e P2 e’: “ << Point_GetDistance(P1, P2) << endl;
  if (Point_Equal(P1, P2) {
     P3 = Point_Sum(P1, P2);
  }
  else {
     P3 = new Point(1.0, 1.0);
  }
 Point_Print(P3, “Coordinate del Punto P3”);
 if (Point_Aligned(P1, P2, P3) {
   cout << “I tre punti risiedono su una retta” << endl;
 }
 delete P1; delete P2; delete P3;
}
```

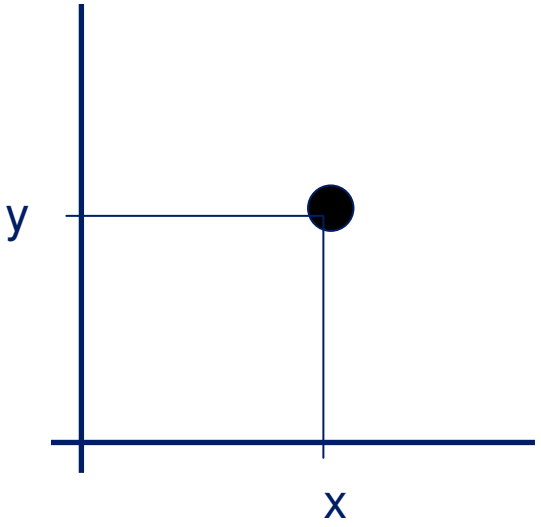
Esempio: tipo di dato astratto punto

- Le definizioni dell'ADT punto memorizzate nel file Punto.h consentono di scrivere un qualunque programma che manipola variabili di tipo Punto indipendentemente da come il tipo è realmente implementato.
  - In questo modo, non sappiamo come si chiamano i campi della struttura che implementa l'ADT.
  - Abbiamo accesso a questi solo attraverso i metodi.
- Per poter linkare e quindi eseguire il programma, sarà necessario anche avere un file Punto.cpp contenente le dichiarazioni dell'ADT punto e dei suoi metodi.
  - Se cambiamo l'implementazione (dichiarazione), a patto che non vengano cambiate le definizioni dei metodi dell'ADT, dobbiamo solo modificare il file delle dichiarazioni, ricompilare solo questo file e ri-linkare.

Esempio: tipo di dato astratto punto

Point.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
#include "Point.h"

// Dichiarazione dell'ADT Point
// con due double per rappresentare
// le coordinate cartesiane
struct Point_ {
   double _x;
   double _y;
   Point_ () {_x = _y = 0.0;}
   Point_ (const double px, const double py) {
      _x = px; _y = py;
   }
};
// Dichiarazione dei metodi
// ....
```




```
PointPtr P = new Point(10.0,10.0);
Point Q = Point(10.0, 10.0);
```

Metodi dell'ADT punto

```
// Ritorna la coordinate X e Y di P
double Point_GetX(const PointPtr p) {
  return p->_x;
}

double Point_GetY(const PointPtr p) {
  return p->_y;
}

// Assegna le coordinate X e Y di P
void Point_SetX(const PointPtr p, const double x) {
  p->_x = x;
}

void Point_SetY(const PointPtr p, const double y) {
  p->_y = y;
}
```

Metodi dell'ADT punto

```
// Predicato per controllare se due Punti sono uguali
bool Point_Equal(const PointPtr P1, const PointPtr P2) {
  return ((Point_GetX(P1) == Point_GetX(P2) &&
           (Point_GetY(P1) == Point_GetY(P2));
}

// Stampa coordinate di un punto P inserendo
// la stringa n prima della stampa delle coordinate
void Point_Print(const PointPtr P, const char * n) {
  cout << n << endl;
  cout << ".X = " << Point_GetX(P) << endl;
  cout << ".Y = " << Point_GetY(P) << endl;
}
```

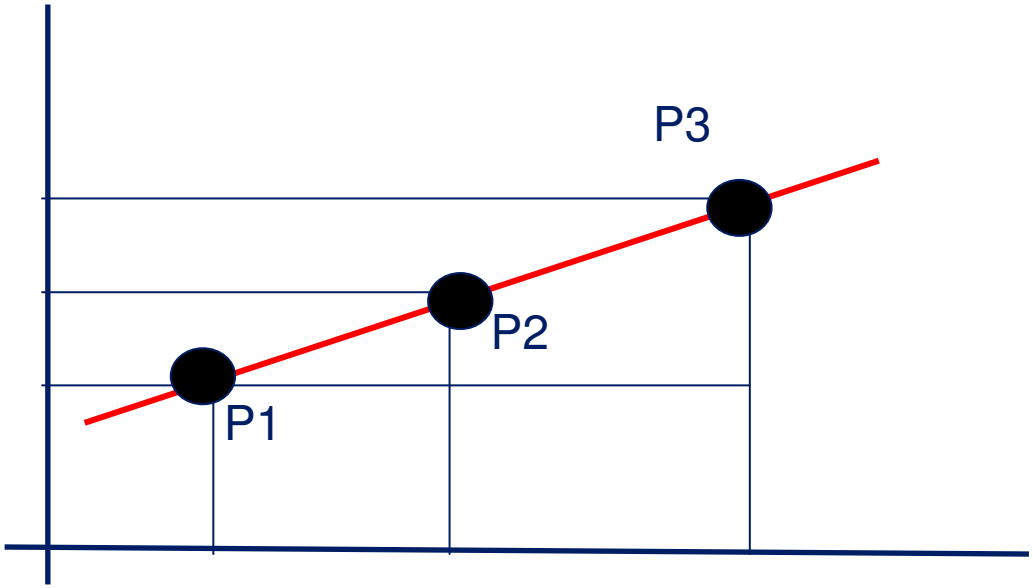
Notare utilizzo dei metodi Point_GetX e Point_GetY per essere ancora più indipendenti dall'implementazione

Metodi dell'ADT punto

```
// calcola la distanza tra due punti
double Points_GetDistance(const PointPtr P1,
                                    const PointPtr P2) {
  double dx = (Point_GetX(P1) - Point_GetX(P2));
  double dy = (Point_GetY(P1) - Point_GetY(P2));

  return sqrt(dx * dx + dy * dy);
}

// Costruisci il punto risultante dalla somma delle
// rispettive coordinate di due punti P1 e P2
PointPtr Point_Sum(const PointPtr P1,
                                const PointPtr P2) {
  PointPtr r;
  r = new Point(Point_GetX(P1) + Point_GetX(P2),
                    Point_GetY(P1) + Point_GetY(P2));
  return r;
}
```

Esercizio

- Scrivere un metodo per l'ADT punto che ritorni *true* se tre punti risiedono sulla stessa retta, *false* altrimenti.

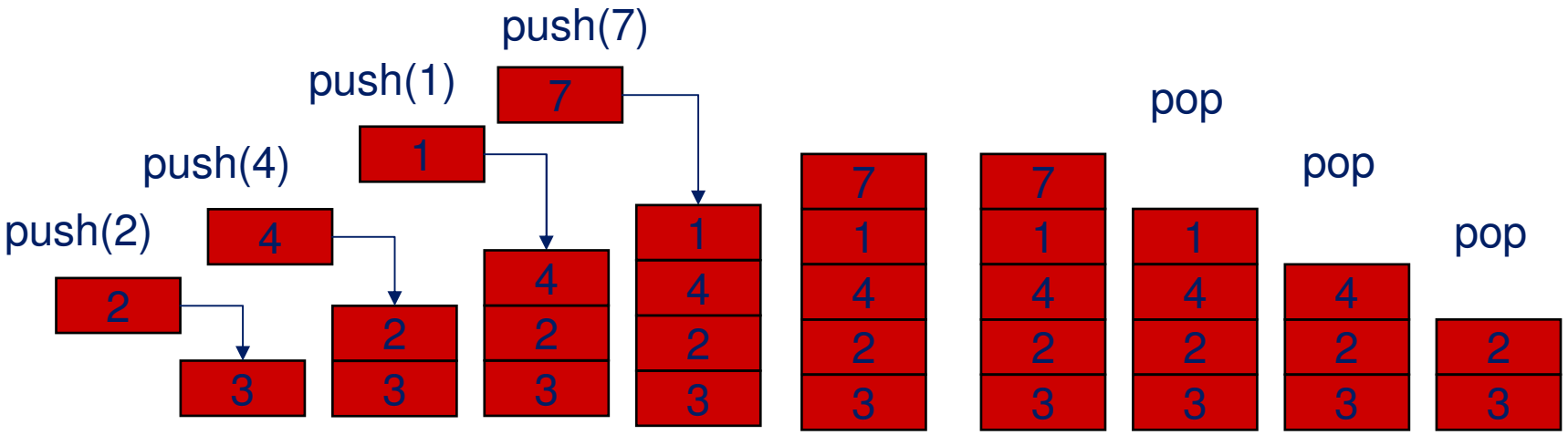


ADT Standard e esempi di uso

- ADT *stack* o *pila*
  - Esempi di utilizzo
  - Analisi di possibili implementazioni
- ADT *coda* FIFO
  - Esempi di utilizzo
  - Analisi di possibili implementazioni

Tipo di Dato Astratto *Stack*

- **Definizione**: uno ***stack*** (o ***pila***) è un ADT che supporta due operazioni base:
  - ***push*** – *inserimento* (*in testa*) di un nuovo elemento;
  - ***pop*** – *cancellazione* (*in testa*) dell'elemento che è stato inserito più di recente e ne ritorna il valore.
  - ***StackIsEmpty*** – *predicato per controllare se lo stack* è vuoto, ovvero non contiene elementi.



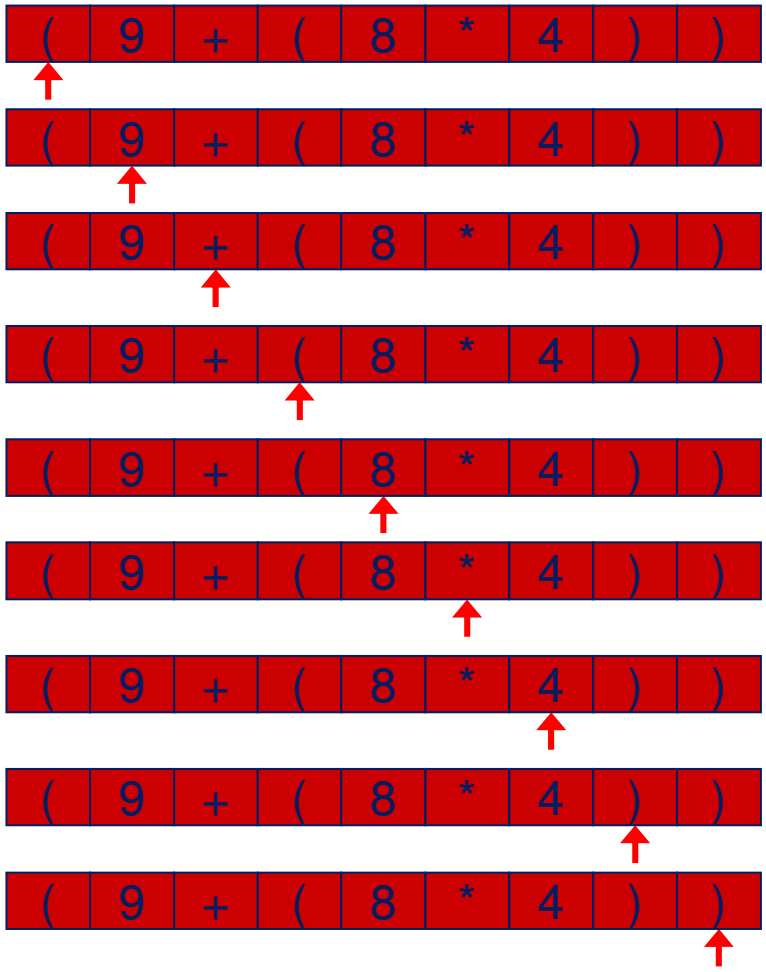
Tipo di Dato Astratto *Stack*

```
// Definizione dell'ADT stack di interi
// (implementazione non specificata)
typedef struct Stack_ Stack;
typedef struct Stack_ * StackPtr;

// Metodi del tipo di dato astratto Stack
// Verifica se lo stack è vuoto o no
bool StackIsEmpty(StackPtr p);

// Inserisce l'elemento intero d nello stack
// incrementando la dimensione dello stack
void Push(StackPtr p, int d);

// Rimuove un elemento dallo stack,
// riducendo la dimensione dello stack e ritorna il valore
int Pop(StackPtr p);
```

Tipo di Dato Astratto *Stack*

- Esempio di uso:
  - Conversione di una espressione infissa, nella corrispondente espressione postfissa.
    - Infissa: (5 * (((9 + 8) * (4 * 6)) +7))
    - Postfissa: 5 9 8 + 4 6 * * 7 + *
  - Procediamo da sinistra a destra nell'espressione:
    - Se troviamo un numero lo scriviamo in output.
    - Se incontriamo una parentesi aperta o uno spazio li ignoriamo;
    - Se incontriamo un operatore lo inseriamo nello stack;
    - Se incontriamo una parentesi chiusa estraiamo l'elemento che sta in cima allo stack e lo stampiamo in output.

Conversione infissa ⇒ postfissa



output	Stack
9	
9	+
9	+
9 8	+
9 8	* +
9 8 4	* +
9 8 4 *	+
9 8 4 * +	

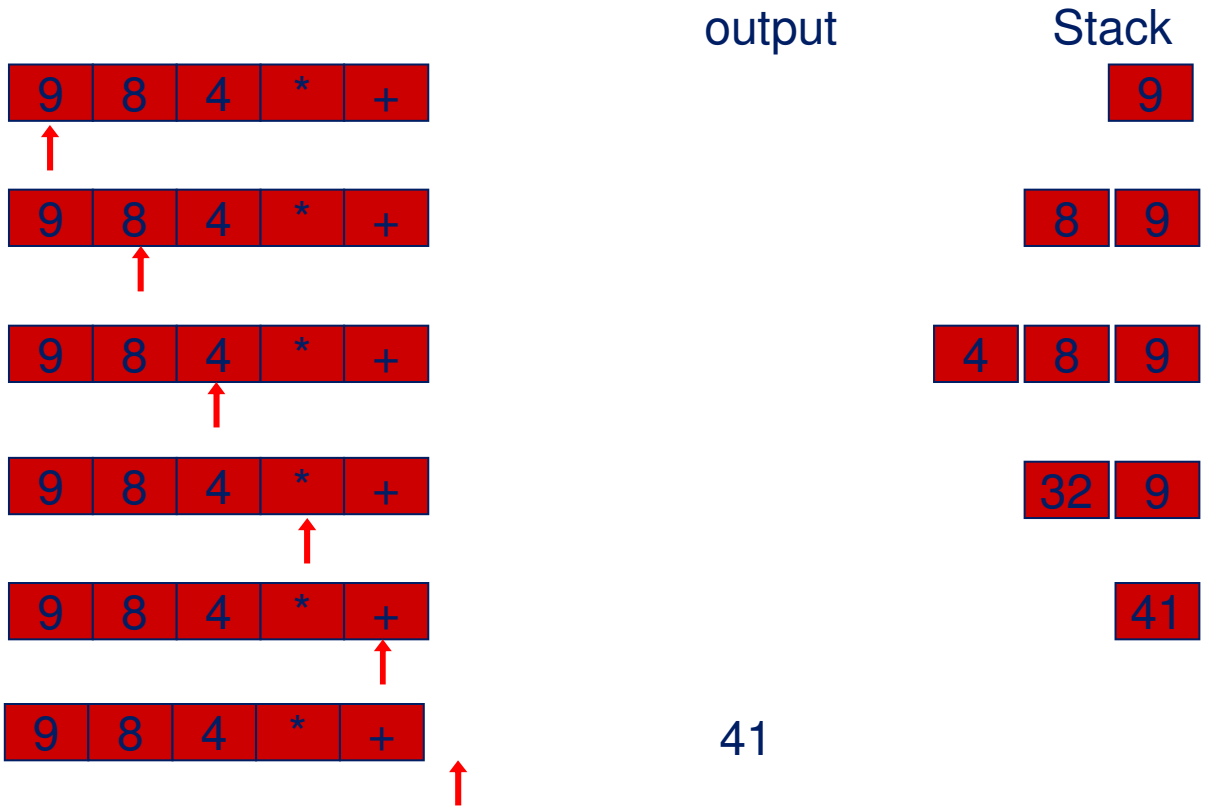
Conversione infissa ⇒ postfissa

```
void  infissa2postfissa(const char * e) {
   StackCharPtr S = new StackChar();
   int i = 0;
   for (i = 0; e[i] != '\0'; i++) {
      if ((e[i] == '(') || (e[i] == ' ')) continue;
      if (e[i] == ')')
          cout << PopChar(S) << " ";
      if ((e[i] == '+') || (e[i] == '*') || (e[i] == '-') || (e[i] == '/'))
          PushChar(S, e[i]);
      if ((e[i] >= '0') && (e[i] <= '9'))
          cout << e[i] << " ";
    }
    cout << endl;
    delete S;
}
```

La conversione considera solo numeri compresi tra 0 e 9.

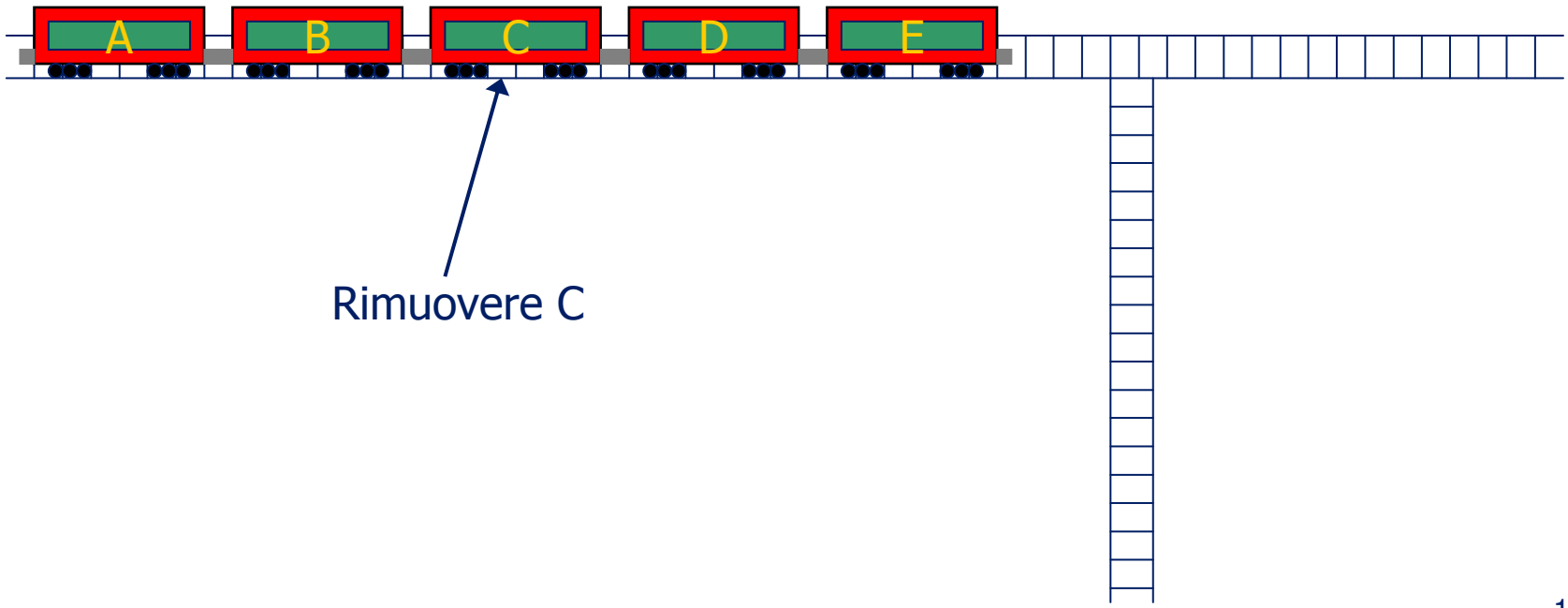
Esercizio

- Modificare la funzione precedente in modo che gestisca numeri maggiori o uguali a 9.
- Scrivere una funzione che prende in input una espressione postfissa e la valuti.



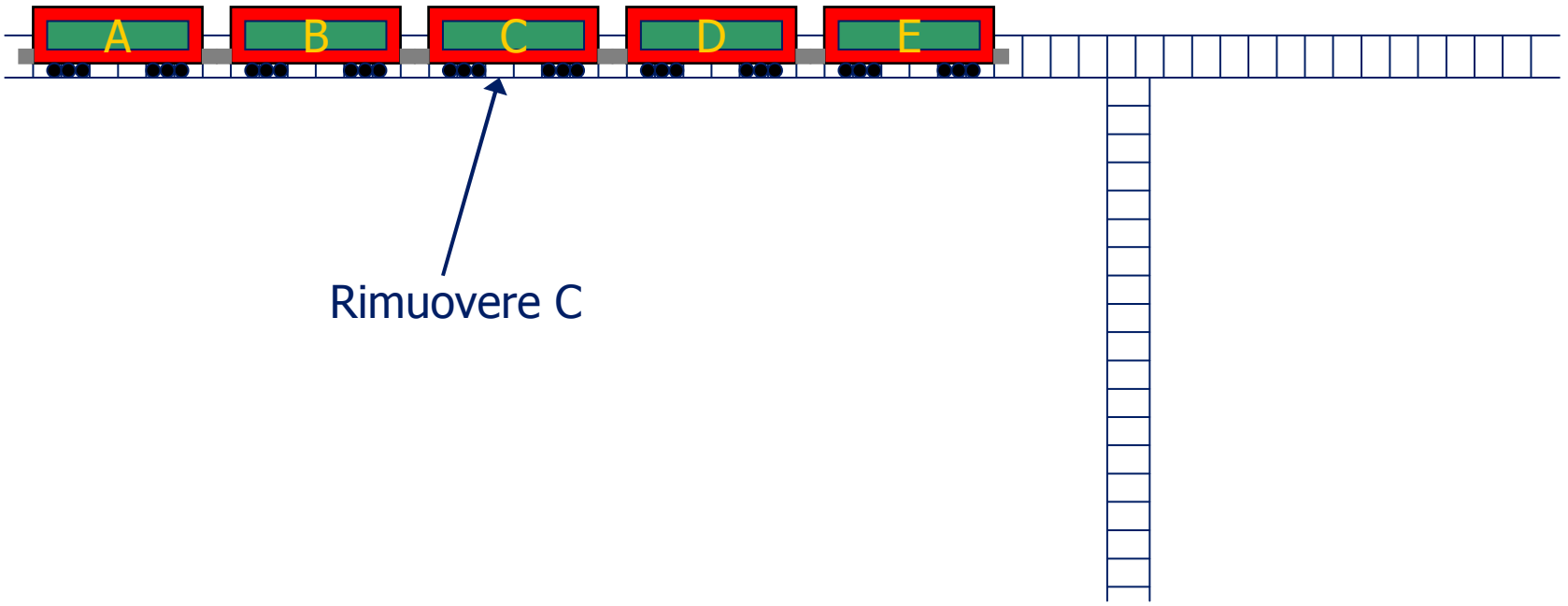
Stack: Esempio d'uso

- Rimozione della prima occorrenza di un nodo target (se esiste) da uno stack s, ritornando true se il nodo è stato rimosso, false altrimenti.



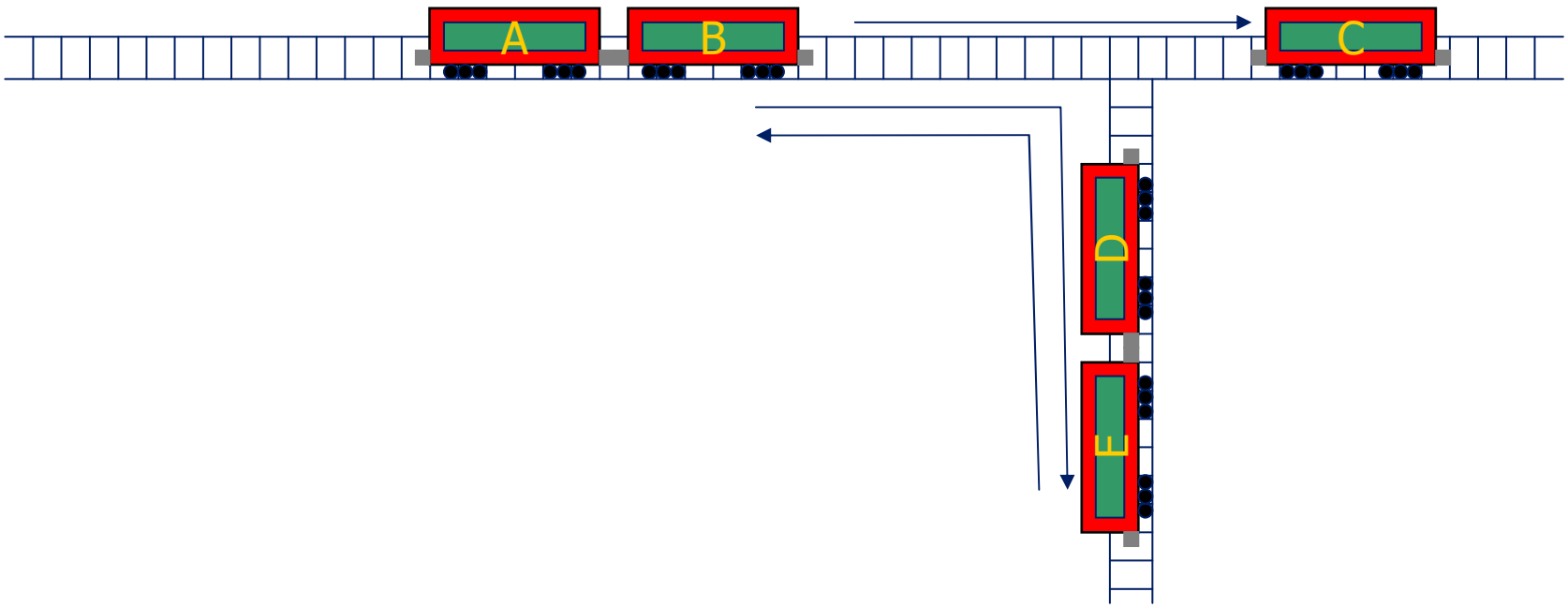
Stack: Esempio d'uso

- Utilizziamo un secondo stack per memorizzare gli elementi rimossi dallo stack fino a trovare l'elemento cercato, per poi estrarli dal secondo stack per rimetterli nello stack originario.



Stack: Esempio d'uso

- Utilizziamo un secondo stack per memorizzare gli elementi rimossi dallo stack fino a trovare l'elemento cercato, per poi estrali dal secondo stack per rimetterli nello stack originario.



Stack: Esempio d'uso

```
bool uncouple( StackPtr s, int c) {
 StackPtr s1 = new Stack();
 bool result = false;

 while ((! StackIsEmpty(s)) && (result == false)) {
   int r = Pop(s);
   if (r == c) {
     result = true;
     break;
   }
   else
     Push(s1, r);
 }
 while(! StackIsEmpty(s1)) {
   int r = Pop(s1);
   Push(s, r);
 }
 delete s1;  return result;
}
```

Implementazione dell'ADT Stack

- Esistono due soluzioni alternative per implementare un ADT di tipo stack.
  - La prima basata su array.
    - Occupazione di memoria fissa.
    - Dimensione massima fissa.
  - La seconda basata su liste concatenate.
    - Occupazione di memoria variabile, dipende da quanti elementi sono inseriti.
    - Dimensione massima può crescere dinamicamente.

Implementazione ADT Stack: Array

```
struct Stack_ {
  int N;
  int dim;
  int * s;
  Stack_ (int X) {
     N = 0;
     dim = X;
     s = new int[X];
  }
};
typedef struct Stack_ Stack;
typedef struct Stack_ * StackPtr;
```

Numero elementi presenti nello stack

Numero massimo di elementi memorizzabili nello stack

Array per memorizzare gli elementi.

Unico costruttore con un solo argomento che corrisponde al numero massimo di elementi memorizzabili nello stack.

Implementazione ADT Stack: Array

```
bool StackIsEmpty(StackPtr p) {
   return (p->N == 0);
}

bool StackIsFull(StackPtr p) {
   return (p->N == dim);
}

void Push(StackPtr p, int x) {
   if (StackIsFull())
       error("Attempting to add an elemento to a full stack");
   p->s[p->N] = x;
    p->N = p->N + 1;
}

int Pop(StackPtr p) {
   if (StackIsEmpty(p))
       error("Attempting to remove an elelement from an empty stack");
  p->N = p->N - 1;
  return p->s[p->N];
 }
```

Nuovo metodo per gestire condizione che la dimensione massima è stata raggiunta.

Implementazione ADT Stack: Liste Concatenate

```
struct _node_ {
  int data;
  _node_ * next;
  _node_(int x, _node_ * n) {
      data = x; next = n;
  }
};

struct Stack_ {
  _node_ * s;
  Stack_ () {
   s = NULL;
  }
};
typedef struct Stack_ Stack;
typedef struct Stack_ * StackPtr;
```

Lista concatenata per gestire gli elementi da memorizzare nello stack.

Implementazione ADT Stack: Liste Concatenate

```
bool StackIsEmpty(StackPtr p) {
  return (p->s == NULL);
}

void Push(StackPtr p, int x) {
  p->s = new _node_(x, p->s);
}

int Pop(StackPtr p) {
  if (StackIsEmpty(p))
     error("Attempting to remove an elelement from an empty stack");
  int r = p->s->data;
  _node_ * t = p->s;
  p->s = p->s->next;
  delete t;
  return r;
}
```

Caratteristiche delle operazioni push e pop

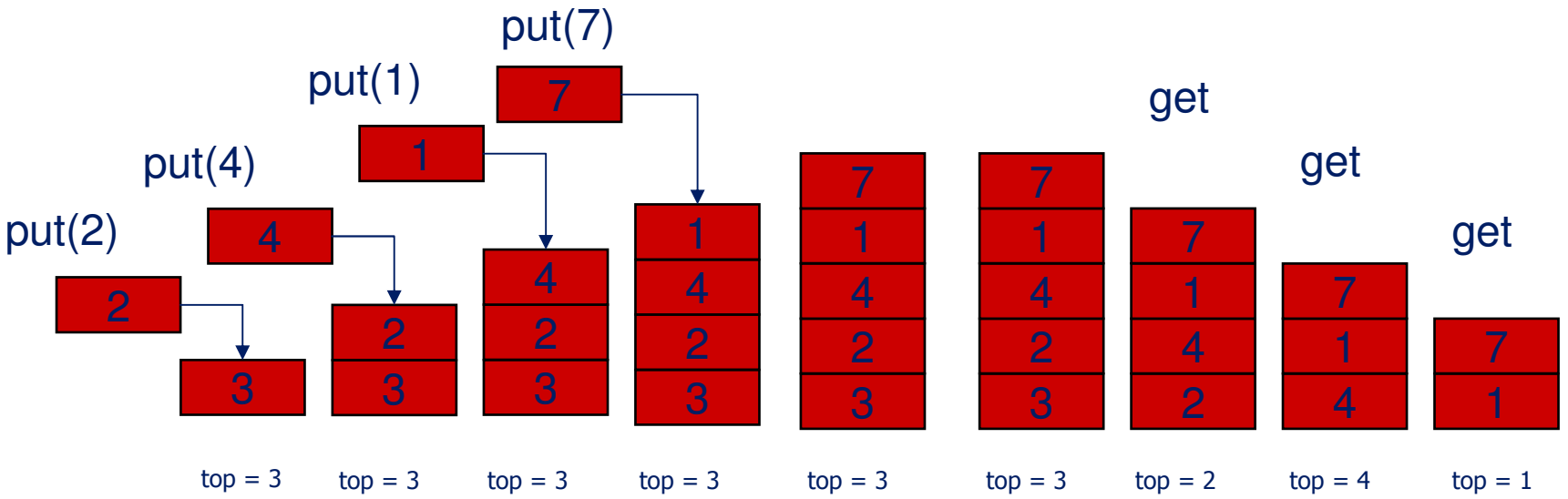
- Le operazioni di ***push*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.
- Le operazioni di ***pop*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.

Tipo di Dato Astratto Coda FIFO

- La *coda FIFO (First-In First-Out)* è un altro ADT fondamentale.
- È simile ad uno stack, ma usa la regola opposta per decidere quale elemento eliminare in una cancellazione.
  - Nelle code viene rimosso l'elemento che è rimasto nella coda più a lungo, a differenza di quello che avviene nello stack in cui viene eliminato l'ultimo elemento inserito.
- Sono largamente utilizzate nella vita comune:
  - Coda per comprare un biglietto per andare al cinema.
  - Nei calcolatori per memorizzare richieste che devono essere servite sulla base dei tempi di arrivo.
  - Nelle reti di calcolatori per memorizzare i pacchetti che arrivano ad un calcolatore da un altro calcolatore.
  - …

Tipo di Dato Astratto Coda *FIFO*

- ***Definizione***: Una ***coda FIFO*** è un ADT che comprende tre operazioni base:
  - ***put*** (*inserisci)*, inserisce un nuovo elemento;
  - ***get*** (*preleva e cancella)* l'elemento che è stato inserito meno recentemente;
  - ***top*** (ispeziona) l'elemento che è stato inserito meno recentemente.



Tipo di Dato Astratto *Coda FIFO*

```
// Definizione dei tipi (implementazione non specificata)
typedef struct Coda_ Coda;
typedef struct Coda_ * CodaPtr;

// Metodi del tipo di dato astratto Coda FIFO
// Verifica se la coda è vuota o no
bool CodaIsEmpty(CodaPtr p);

// Inserisce l'elemento d nella coda
// aumentandone la dimensione
void Put(CodaPtr p, int d);

// Rimuove un elemento dalla coda,
// diminuendone la dimensione
int Get(CodaPtr p);

// ritorna elemento in cima alla coda
// nessuna modifica alla coda
int Top(CodaPtr p);
```

Code: Esempio uso

- Scrivere un programma per effettuare uno scheduling di interviste.
  - Un prompt chiede ad una segretaria di inserire l'ora della prima intervista,
  - e poi continua ad iterare per la richiesta dell'ora della intervista successiva fintanto che la segretaria non inserisce un orario maggiore o uguale alle ore 17.
- Una volta inseriti questi dati, un loop stampa l'ora dell'appuntamento per l'intervista con la durata prevista per l'intervista stessa.
- Quando la coda diventa vuota, la durata dell'ultima intervista corrisponde all'intervallo di tempo compreso tra l'ora di inizio dell'intervista e l'ora di chiusura dell'ufficio (ovvero le 17:00).

Code: Esempio uso

```
#include <iostream>
using namespace std;

#include "CodaTime.h"

void LeggiAgenda(CodaTimePtr);
void StampeAgenda(CodaTimePtr);

int main() {
  CodaTimePtr q = new CodaTime();

  // Lettura dell'agenda
  LeggiAgenda(q);

  cout << "Appuntamento     ";
  cout  << "Tempo disponibile intervista";
  cout  << endl;
  StampaAgenda(q);

  delete q;
}
```

Inclusione dell'header file CodaTime.h, contenente le definizioni del tipo Time e dell'ADT Coda di elementi di tipo Time

Creazione della coda vuota

Riempimento della coda

Svuotamento della coda

Deallocazione della coda

Code: Esempio uso

```
void LeggiAgenda(CodaTimePtr q) {
 int h, m;
 Time inttime;

 cout << "Inserire ora prima intervista: ";  cin >> h;
 cout << "Inserire minuti prima intervista: ";  cin >> m;
 inttime = Time(h,m);
 Put(q, inttime);
 while( TimeGetH(inttime) < 17 ) {
   cout << "Inserire ora prossima intervista: ";
   cin >> h;
   cout << "Inserire minuti prossima intervista: ";
   cin >> m;
   inttime = Time(h,m);
   Put(q, inttime);
 }
```

Lettura dell'orario della prima intervista.

Inserimento dell'orario di inizio nella coda.

Lettura dell'orario della prossima intervista.

Inserimento dell'orario della prossima intervista nella coda.

Code: Esempio uso

```
void StampaAgenda(CodaTimePtr q) {
   while( ! CodaIsEmpty(q) ) {
     Time t = Get(q);

     cout << timeGetH(t) << ":" << timeGetM(t) << "    " ;
     if (CodaIsEmpty(q) )
        cout << difftime(Time(17,0), t) << endl;
     else
        cout << difftime(Top(q), t) << endl;
   }
}
```

Finchè ci sono elementi nella coda

Estraggo l'elemento corrente dalla coda.

Esercizio: Implementare l'ADT Time
e l'ADT CodaTime

Implementazione dell'ADT Coda FIFO

- Similmente al caso dell'ADT Stack, esistono due soluzioni alternative per implementare un ADT di tipo coda FIFO.
  - La prima basata su array.
    - Occupazione di memoria fissa.
    - Dimensione massima fissa.
  - La seconda basata su liste concatenate.
    - Occupazione di memoria variabile, dipende da quanti elementi sono inseriti.
    - Dimensione massima può crescere dinamicamente.

Implementazione ADT Coda FIFO: Array

```
struct Coda_ {
  int N;
  int head;
  int tail;
  int * s;
  Coda_ (int X) {
     N = X + 1;
     head = X + 1;
     tail = 0;
     s = new int[X+1];
  }
 typedef struct Coda_ Coda;
 typedef struct Coda_ * CodaPtr;
```

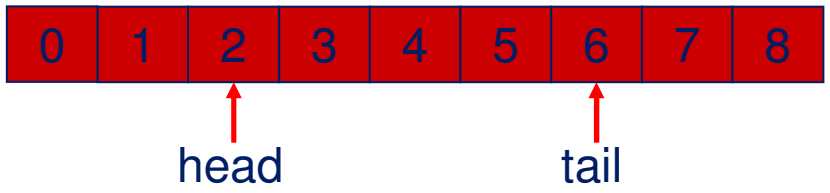
Dimensione massima della coda, gestita come array circolare

Indice che identifica l'elemento in testa alla coda

Indice che identifica l'elemento in cui inserire un nuovo elemento nella coda. Gli elementi sono compresi tra head e tail-1

Array dinamico di dimensione N contente gli elementi della coda

Costruttore ad un argomento atto a specificare la dimensione massima della coda (e quindi dell'array)



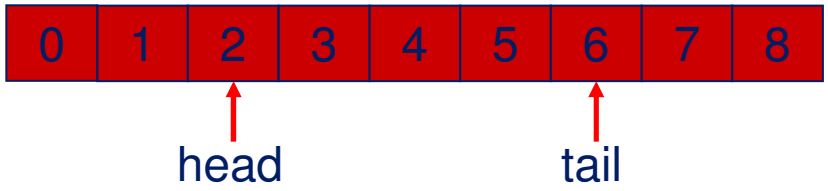
Implementazione ADT Coda FIFO: Array

```
bool CodaIsEmpty(CodaPtr p) {
  if ((p->head % p->N) == p->tail)
     return true;
  return false;
}

void Put(CodaPtr p, int x) {
  if (p->tail == p->head)
    error("Coda piena");
  p->s[p->tail] = x;
  p->tail = (p->tail + 1) % p->N;
}
```

Se tail e head hanno lo stesso valore, allora la coda è piena

Memorizzo in tail il nuovo valore inserito nella coda

Avanzo tail in modo che indicizzi il prossimo posto libero dove inserire i nuovi valori inseriti nella coda.



Implementazione ADT Coda FIFO Array

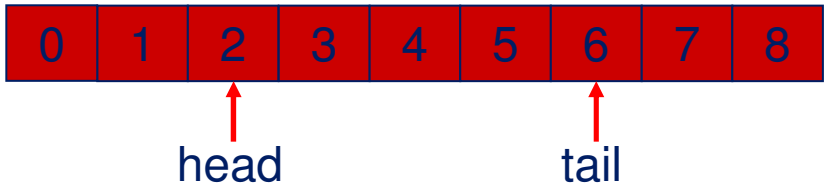
```
int Top(CodaPtr p) {
   int i = p->head % p->N;
   return p->s[i];
 }

 int Get(CodaPtr p) {
   p->head = p->head % p->N;
   int r = p->s[p->head];
   p->head = p->head + 1;
   return r;
 }
```

Normalizzo il valore di head. In Get lo incremento solo, quindi è necessario fare si che assuma valore in 0..N-1 prima di utilizzarlo.

Memorizzo in variabile locale il valore memorizzato nell'elemento dell'array indicizzato da head.

Incremento head in modo che per le prossime Get abbia il valore giusto.

Ritorno il valore



Implementazione ADT Coda FIFO lista concatenata

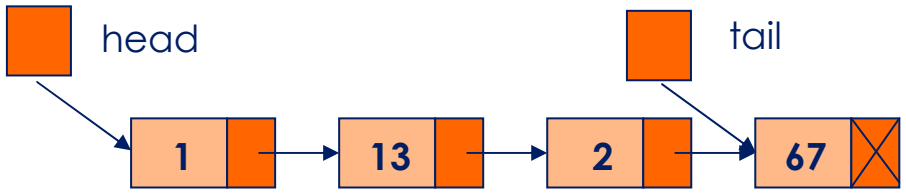
```
struct node {
  int data;
  node * next;
  node(int x, node * n) {
      data = x; next = n;
  }
};

struct Coda_ {
  node * head;
  node * tail;
  Coda_ () {
   head = NULL; tail = NULL;
  }
};
typedef struct Coda_ Coda;
typedef struct Coda_ * CodaPtr;
```

Struttura node per memorizzare gli elementi della lista utilizzata per implementare la coda.

Puntatore al nodo di testa

Puntatore al nodo di coda, per inserimento in coda

Costruttore, che inizializza la lista alla lista vuota: head e tail hanno valore NULL



Implementazione ADT Coda FIFO lista concatenata

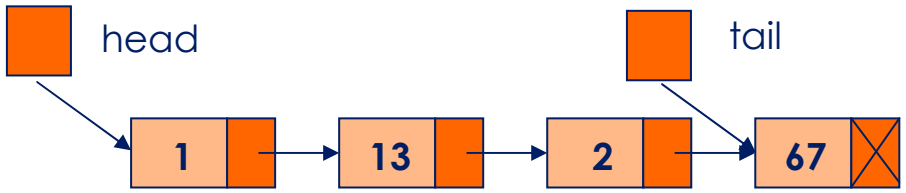
```
bool CodaIsEmpty(CodaPtr p) {
  if (p->head == NULL) return true;
  return false;
}

void Put(CodaPtr p, int x) {
  node * t = p->tail;
  p->tail = new node(x, NULL);
  if (p->head == NULL) p->head = p->tail;
  else t->next = p->tail;
}
```

Inserimento in coda: memorizzo in t il puntatore all'ultimo elemento.

L'ultimo elemento punta ad un nuovo nodo per memorizzare il nuovo valore inserito nella coda.

Aggiorno i campi head e tail per rispecchiare lo stato della coda. Se la testa è NULL, head punto all'unico elemento che corrisponde a tail.

Altrimenti ricostruisco la lista facendo puntare il campo next dell'ultimo elemento precedentemente al nuovo elemento inserito (insriemento in coda)



Implementazione ADT Coda FIFO lista concatenata

```
Time Top(CodaTimePtr q) {
    return q->head->data;
}

int Get(CodaPtr p) {
  int r = p->head->data;
  node * t = p->head->next;
  delete p->head;
  p->head = t;
  return r;
}
```

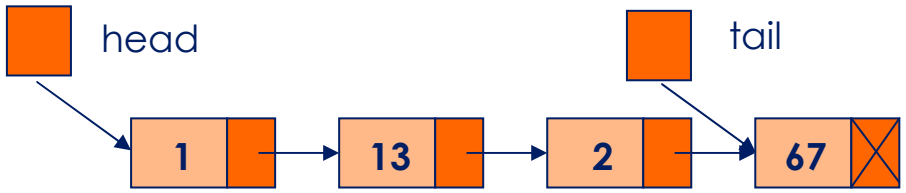
Rimozione in testa. Memorizzo in r il valore memorizzato in testa.

Memorizzo il puntatore al nodo successivo al nodo di testa.

Deallocazione della memoria corrispondente al nodo di testa.

Ricostruzione dei campi della coda in modo che head punti al nodo t, in cui è memorizzato il puntatore al nodo successivo al nodo di testa appena rimosso.

Nell'implementazione proposta non si tiene conto di alcuni casi limite. Per esercizio, modificare l'implementazione per tenerne conto.



Caratteristiche delle operazioni put e get

- Le operazioni di ***put*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.
- Le operazioni di ***get*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.

Esercizi

- Modificare le funzioni di **get** per un ADT coda FIFO in modo che invochi la funzione **error()** quando di richiede di estrarre un elemento dalla coda vuota.
- Modificare l'implementazione della **put** per l'ADT coda FIFO implementata con array, in modo che chiami **error()** se invocata su una coda piena.
- Modificare le implementazioni di coda FIFO in modo da evitare di avere nella coda elementi duplicati.
  - Hint: l'interfaccia non cambia, quello che cambia è l'implementazione (dichiarazione) dei metodi **get**, e **put.**
  - Esempi in cui le code senza duplicati sono utili:
    - Una azienda vuole mantenere una mailing list, inserendo indirizzi provenienti da diverse mailing list, evitando di inserire un cliente già nella lista.
    - Consideriamo il problema dell' instradamento di un messaggio all'interno di una complessa rete di comunicazione. Un messaggio potrebbe percorrere simultaneamente diverse strade per raggiungere la destinazione, ma ogni nodo della rete vuole memorizzare nelle sue strutture interne una sola copia del messaggio.

Prospettive

- Gli ADT sono uno strumento di ingegneria del software di uso diffuso, e molti degli algoritmi che studiamo servono ad implementare ADT fondamentali e di ampia applicabilità.
- GLi ADT ci aiutano a incapsulare gli algoritmi che sviluppiamo, in modo tale da poter riutilizzare lo stesso codice per scopi diversi.
- GLi ADT forniscono un conveniente meccanismo che possiamo sfruttare in fase di sviluppo di algoritmi e di confronto delle prestazioni.
- Gli ADT concretizzano il ragionevole principio secondo il quale siamo obbligati a descrivere con precisione i modi in cui manipoliamo i dati.
- Gli ADT possono essere utilizzati per costruire altri ADT.