INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento
A.A. 2003/2004
II Bimestre

Marco Roveri

roveri@irst.itc.it

Tipi dato astratto

Abstract Data Type

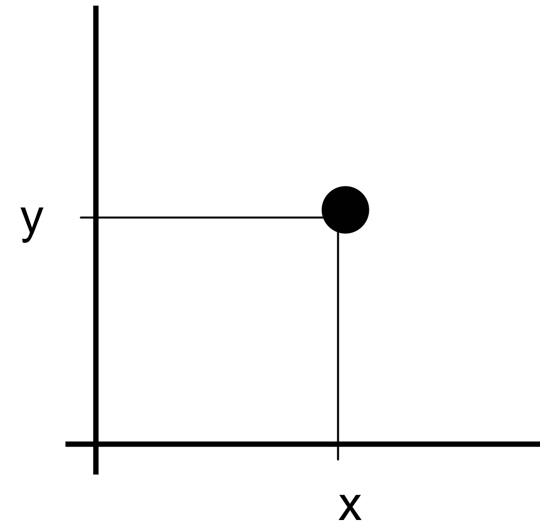
- **Definizione**: un ADT (*Abstract Data Type*) è un tipo di dato (un insieme di valori e una collezione di operazioni su questi valori) accessibili ***solo*** attraverso una interfaccia, nota con il nome di *metodi*.
- Il C++ mette a disposizione dei costrutti particolari per definire gli ADT, le ***classi***, che vedrete nel prossimo corso.
- Noi vederemo 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 sono:
  - Ritorna la coordinata x (y) rispettivamente.
  - 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 punto

```
// Dichiarazione della struttura Point
// con due double per rappresentare
// le coordinate cartesiane
 struct Point_ {
   double x;
   double y;
   Point_ (int px, int py) {
      x = px; y = py;
   }
 };
 typedef struct Point_ Point;
 typedef struct Point _* PointPtr;
```




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

Metodi dell'ADT punto

```
// Ritorna la coordinata X di P
double PointGet_X(PointPtr p) {
  return p->x;
}

// Ritorna la coordinata Y di P
double PointGet_Y(PointPtr p) {
  return p->y;
}
```

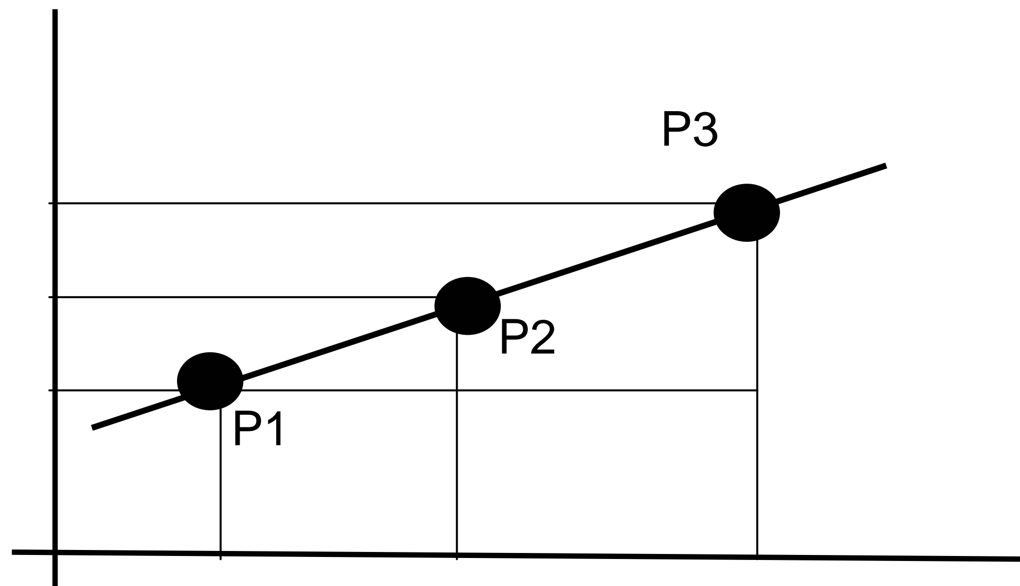
```
// Stampa coordinate di un punto
void PointPrint(const PointPtr P, const char * n) {
  cout << n << ".X = " << PointGet_X(P) << endl;
  cout << n << ".Y = " << PointGet_Y(P) << endl;
}
```

Metodi dell'ADT punto

```
// calcola la distanza tra due punti
double PointsGetDistance(PointPtr P1, PointPtr P2) {
  double dx = (P1->x - P2->x);
  double dy = (P1->y - P2->y);

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

// somma le coordinate di due punti e ritorna un nuovo punto
PointPtr PointsAdd(PointPtr P1, PointPtr P2) {
  PointPtr r;
  r = new Point(P1->x + P2->x, P1->y + P2->y);
  return r;
}
```

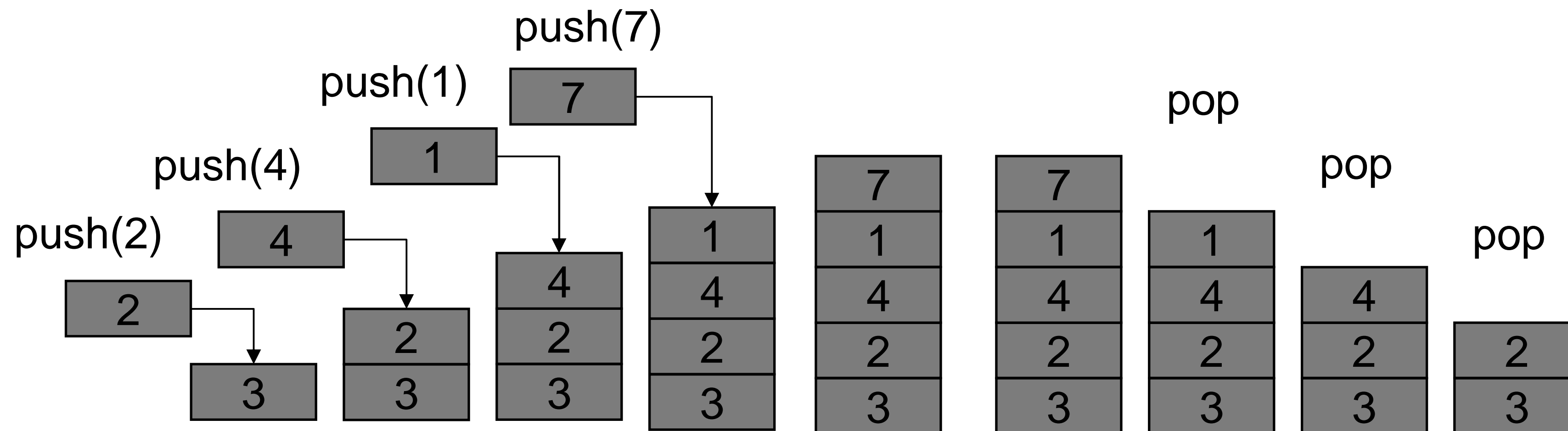
Esercizio

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



Tipo di Dato Astratto *Stack*

- **Definizione**: uno ***stack*** (o ***pila***) è un ADT che supporta due operazioni base: *inserimento* (***push**, inserisci in cima*) di un nuovo elemento, e *cancellazione* (***pop**, preleva dalla cima*) dell'elemento che è stato inserito più di recente.



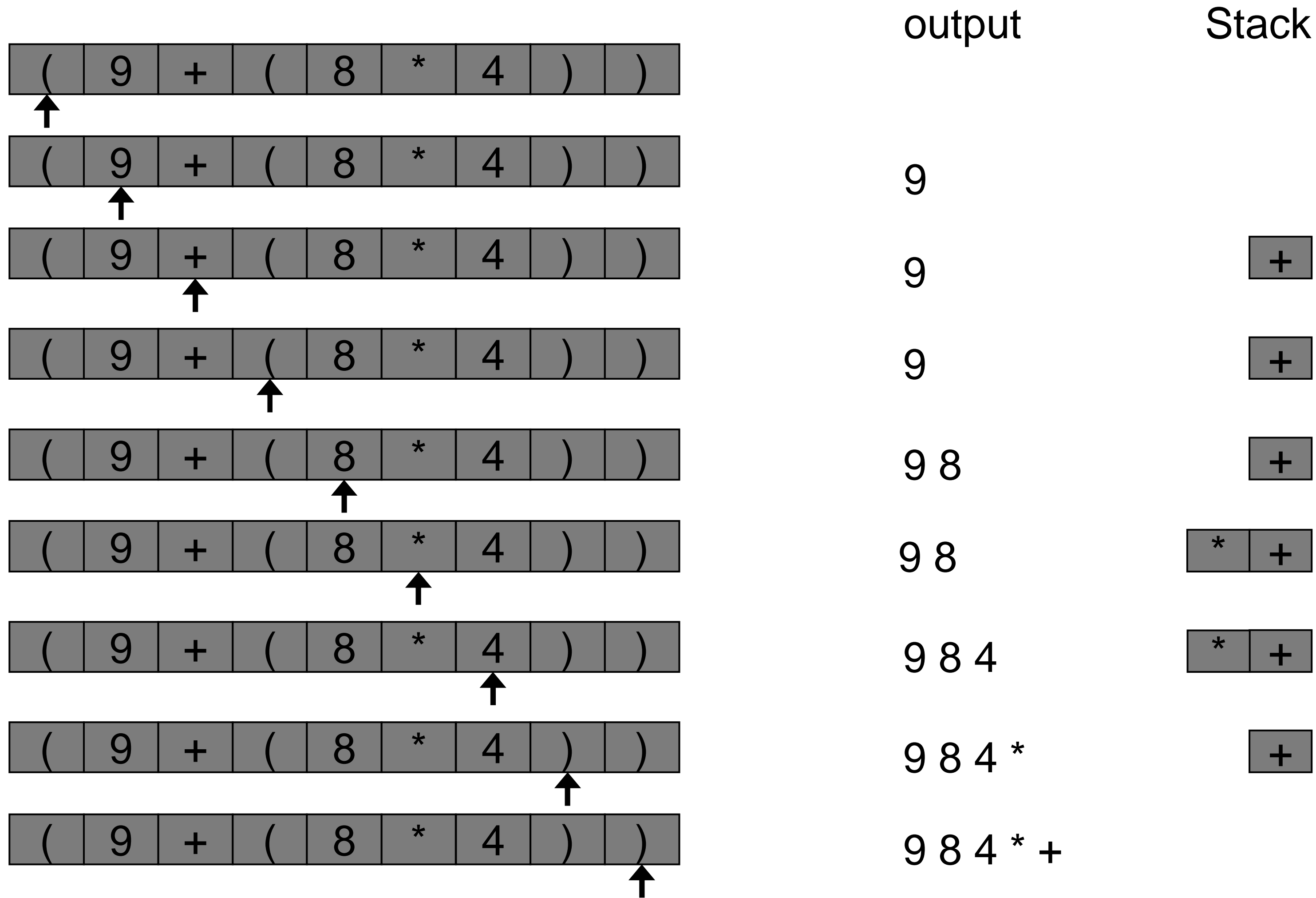
Tipo di Dato Astratto *Stack*

```
// Definizione dei tipi (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 d nello stack
// aumentandone la dimensione
void Push(StackPtr p, int d);

// Rimuove un elemento dallo stack,
// diminuendone la dimensione
int Pop(StackPtr p);
```

Tipo di Dato Astratto *Stack*

- Esempio di uso:
  - Conversione di una espressione infissa, nella corrsipondente 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 la 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

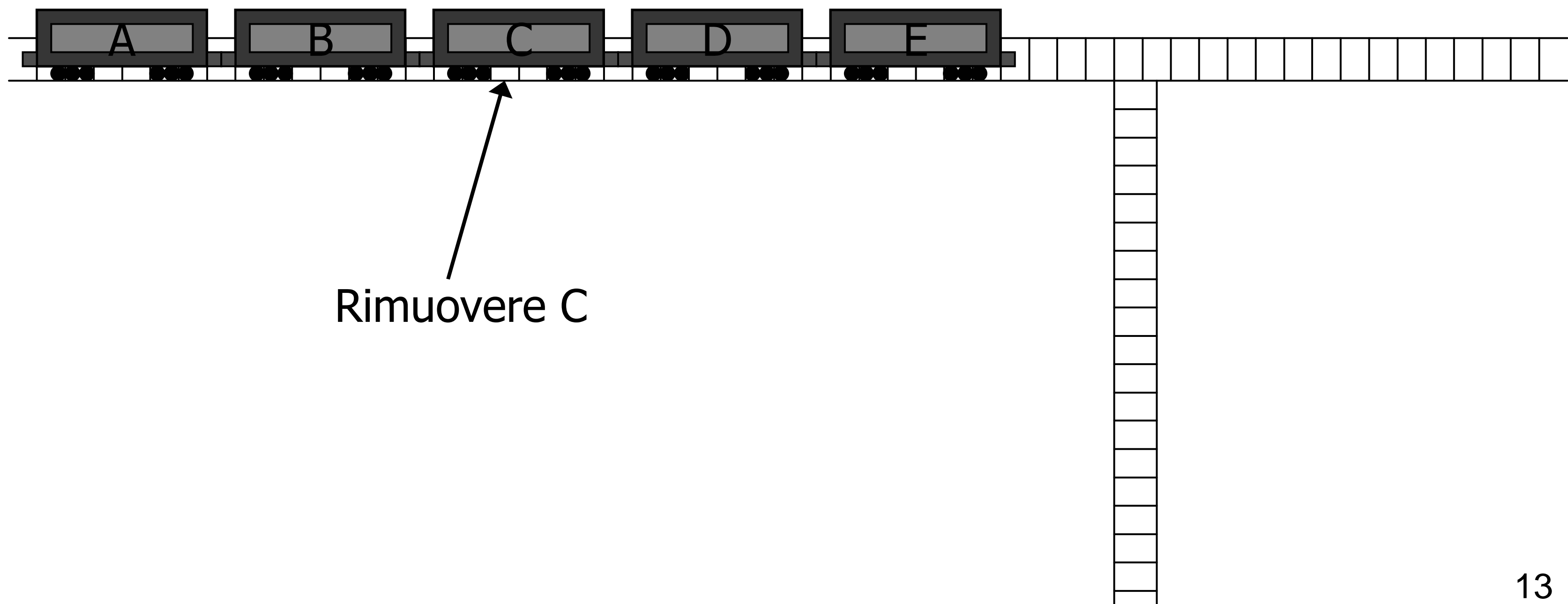


Conversione infissa -> postfissa

```
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] == ‘*’))
         PushChar(S, e[i]);
     if ((e[i] >= ‘0’) && (e[i] <= ‘9’))
         cout << e[i] << “ “;
   }
   cout << endl; // solo numeri compresi tra 0 e 9
   delete S;
}
```

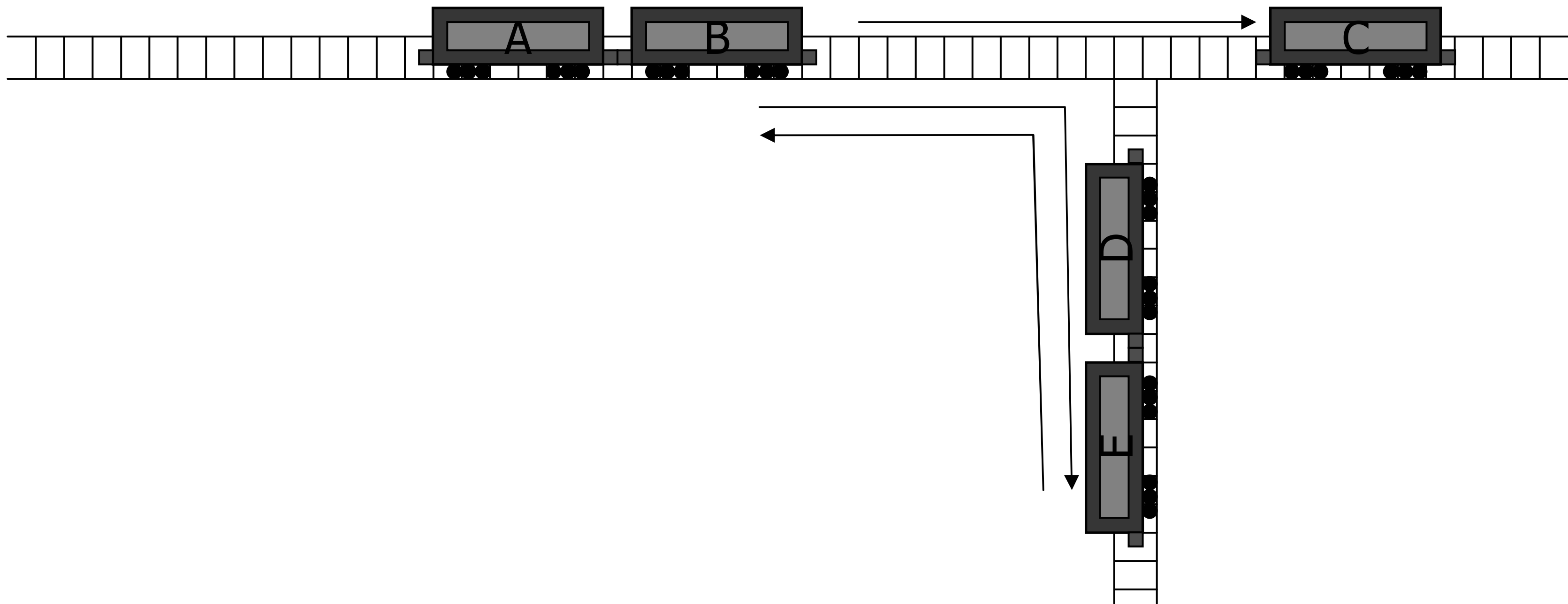
Stack: Esempio d’uso

- 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 estrali dal secondo stack per rimetterli nello stack originario.

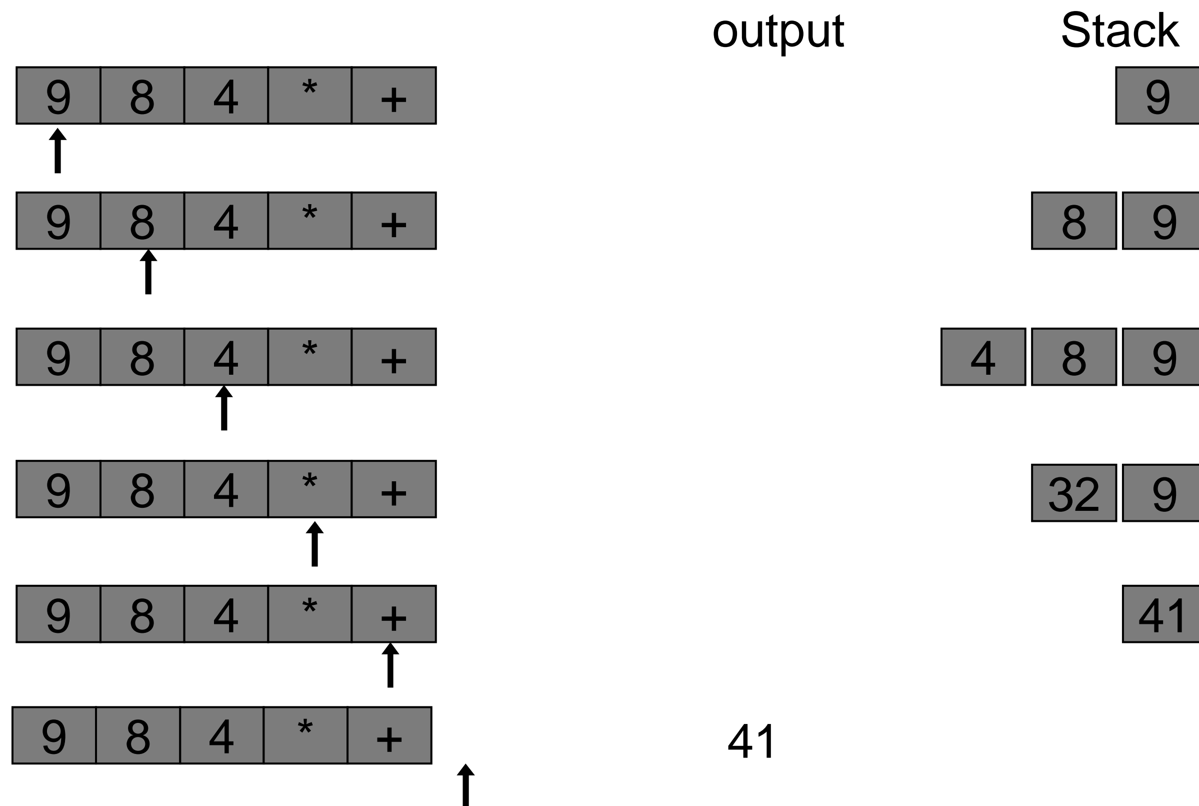


Stack: Esempio d'uso

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

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.



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 = -1;
     dim = X;
     s = new int[X];
  }
 typedef struct Stack_ Stack;
 typedef struct Stack_ * StackPtr;
```

```
bool StackIsEmpty(StackPtr p) {
  if (p->N == -1) return true;
  return false;
}
```

```
void Push(StackPtr p, int x) {
  if (p->N >= p->dim)
     error("Out of stack dimension");
  p->s[p->N] = x;
  p->N = p->N + 1;
}
```

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

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;
```

```
boolean StackIsEmpty(StackPtr p) {
  if (p->s == NULL) return true;
  return false;
}

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

int Pop(StackPtr p) {
  if (p->s == NULL)
     error(“Removing from empty”);
  int r = p->s->data;
  node * t = p->s;
  p->s = p->s->next;
  delete t;
  return r;
}
```

Caratteristiche delle operazioni push e pop

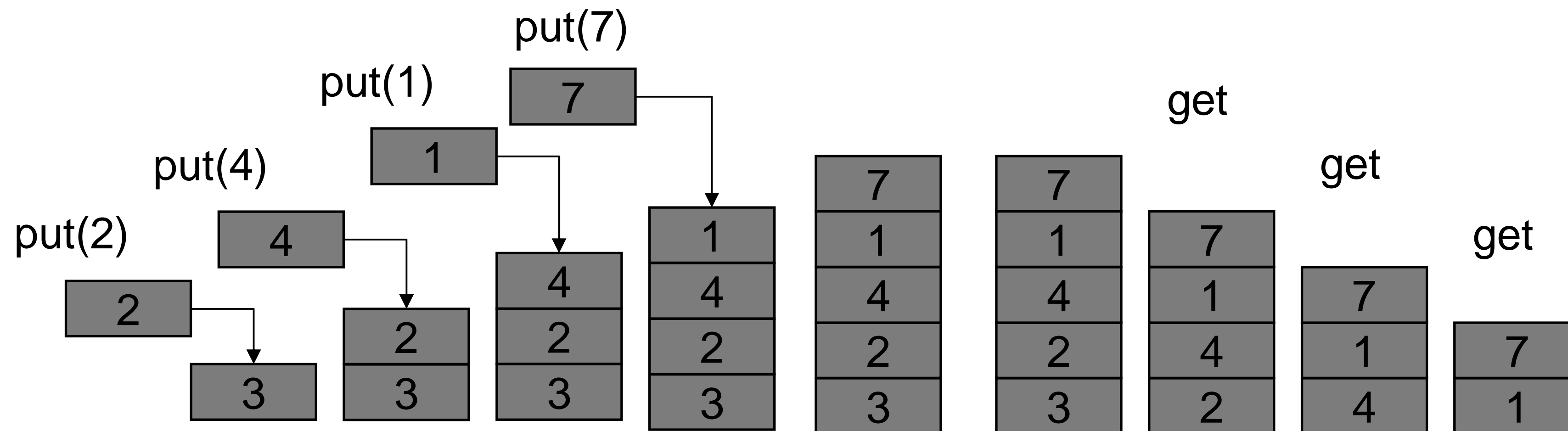
- Le operazioni di ***push*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.
- Le operazioni di ***pop*** usano tempo costante sia nel caso di implementazione con array che con liste concatenate.

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.
  - Piuttoto che eliminare l'ultimo elemento inserito, nelle code viene rimosso l'elemento che è rimasto nella coda più a lungo.
- Sono largamente utilizzate nella vita comune:
  - Coda per comprare un biglietto per andare al cinema.
  - Nei calcolatori per memorizzare istanze, 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 due operazioni base: ***put*** (*inserisci)*, inserisce un nuovo elemento; e ***get*** (*preleva e cancella)* 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);
```

Tipo di Dato Astratto *Coda FIFO*

```
// 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 programme 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 chisura dell'ufficio (17).

Code: Esempio uso

```
int main() {
  Time inttime;
  CodaTimePtr q = new CodaTime();

  LeggiAgenda(q);

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

  delete q;
}
```

Code: Esempio uso

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

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;
  }
}
```

Esercizio: Implememtare la classe Time
E la classe CodaTime

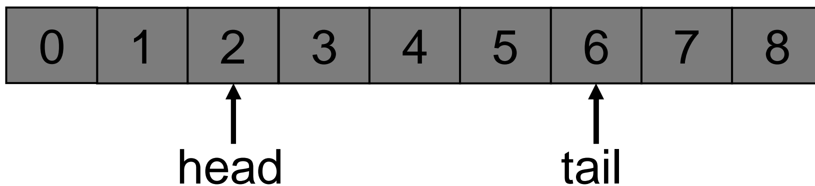
Implementazione dell'ADT Coda FIFO

- 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;
```

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

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



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;
}
```

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;
```

```
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;
}
```

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;
}
```

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 implentazioni di coda FIFO in modo da evitare di avere nella coda elementi duplicati.
  - Hint: l'interfaccia non cambia, cambia l'implementazione dei metodi **get**, e **put.**
  - Esempi in cui 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 restazioni.
- Gli ADT concretizzano il ragionevole principio secondo il quale siamo obbligati a descrvere con precisione I modi in cui manipoliamo I dati.
- Gli ADT possono essere utilizzati per costruire altri ADT.