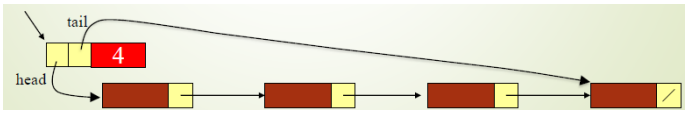
L05.1. ADT QUEUE

Una **queue** (coda) è una sequenza di elementi di un determinato tipo, in cui gli elementi si aggiungono da un lato (**tail**) e si tolgono dall'altro (**head**).
La sequenza viene gestita con la **modalità FIFO** (**First-in-first-out**), ovvero il primo elemento inserito nella sequenza sarà il primo ad essere eliminato.
La coda è una struttura dati lineare a dimensione variabile, si può accedere direttamente solo alla testa (*head*) della lista e non è possibile accedere ad un elemento diverso da head, se non dopo aver eliminato tutti gli elementi che lo precedono (cioè quelli inseriti prima).
Tra le *possibili* implementazioni, le più usate sono realizzate tramite **Lista concatenata** e **Array**, concentriamoci sulla prima.
È possibile utilizzare gli operatori di **rimozione dalla testa** e **aggiunta in coda** definite nella Lista concatenata.
Per motivi di efficienza, conviene avere accesso sia al primo elemento sia all'ultimo ed occorre modificare il tipo lista come un puntatore ad una struct che contiene un intero **numelem** che indica il numero di elementi della coda, un puntatore **head** ad uno **struct node** e un puntatore **tail** ad uno **struct node**.

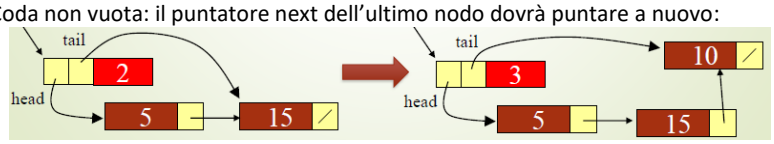


Dobbiamo innanzitutto aggiungere il puntatore tail, poi bisogna modificare gli operatori principali:

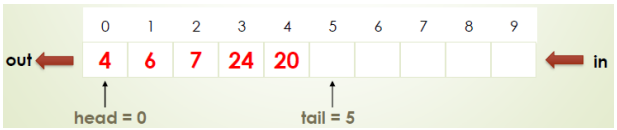
- **removeHead**, deve eventualmente aggiornare entrambi i puntatori head e tail.
  Bisogna prima salvare il puntatore al nodo da eliminare (quello puntato da head), head dovrà quindi puntare al successivo. A questo punto si può deallocare la memoria del nodo da rimuovere, *se la coda aveva un solo elemento, ora è vuota, per cui bisogna porre anche il puntatore tail a NULL*;
- **addListTail**, grazie alla presenza del puntatore tail, non deve più scorrere gli elementi della lista fino all'ultimo e deve eventualmente aggiornare entrambi i puntatori head e tail.
  Dobbiamo innanzitutto creare un nuovo nodo a cui dovrà puntare il puntatore tail, poi bisogna distinguere il caso in cui la coda di input è vuota e il caso in cui non è vuota.

Coda vuota: il puntatore head dovrà puntare al nuovo nodo:

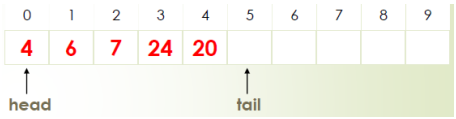
Coda non vuota: il puntatore next dell'ultimo nodo dovrà puntare a nuovo:



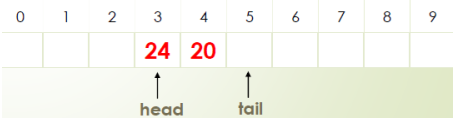
Osserviamo adesso l'implementazione della Queue tramite **Array**.
La coda è implementata come un puntatore ad una **struct queue** che contiene tre elementi, ovvero un array di **MAXQUEUE** elementi, un intero che indica la posizione **head** della coda e un intero che indica la posizione **tail** della coda. Quando la coda si riempie, non è possibile eseguire enqueue.



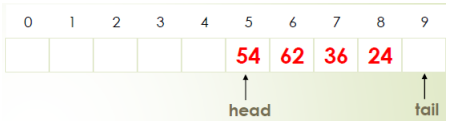
Ma con questa implementazione sorgono alcuni *problemi*.
Se l'array viene gestito normalmente, cioè mantenendo head <= tail, ci sono dei problemi:



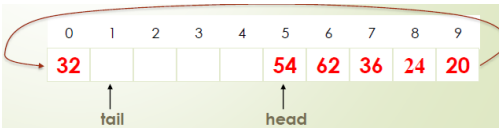
Se rimuoviamo uno alla volta i primi tre elementi in coda otteniamo:



Risultano disponibili solo le posizioni a destra di tail, ma sono libere anche quelle a sinistra di head
Prima soluzione: ad ogni rimozione si compatta l'array nelle posizioni iniziali, con uno shift degli elementi, ma è TROPPO COSTOSO!
Seconda soluzione: si gestisce l'array in modo circolare che in ogni istante, gli elementi della coda si trovano nel segmento head, head+1, ... tail-1, ma non necessariamente head <= tail, infatti, dopo aver inserito in posizione N-1 (ultima posizione dell'array), se c'è ancora spazio in coda, si inseriscono ulteriori elementi a partire dalla posizione 0 (prima posizione dell'array). In questo modo si riesce a garantire che ad ogni istante la coda abbia capacità massima di N-1 elementi.



Supponiamo di voler inserire 20 e 32 in coda:



Adesso tail< head, perché la posizione 0 segue la posizione N-1. In questo ordine circolare il successore di p è (p + 1) % N, ogni volta che si inserisce un elemento tail avanza: tail= (tail+ 1) % N e ogni volta che si rimuove un elemento head avanza: head = (head + 1) % N.
La coda è piena (non si può usare enqueue) se il successore di tail in questo ordine circolare è head, (tail+ 1) % n == head, ***la condizione comporta una locazione vuota necessaria a distinguere la condizione di buffer vuoto da quella di pieno.*** Quando la coda è vuota, head e tail coincidono.

Sintattica	Semantica
Nome del tipo: Queue Tipi usati: Item, boolean	Dominio: insieme di sequenze $S=a_1,\ldots,a_n$ di tipo Item L'elemento **nil** rappresenta la coda vuota
newQueue() → Queue	newQueue() → q • Post: q = nil
isEmptyQueue(Queue) → boolean	isEmptyQueue(s) → b • Post: se q=nil allora b = true altrimenti b = false
enqueue(Queue, Item) → Queue	enqueue(q, e) → q' • Post: q = <a1, …, an> AND q' = <a1, …, an, e>
dequeue(Queue) → Queue	dequeue(q) → q' • Pre: q = <a1, a2, …, an> n>0 • Post: q' = <a2, …, an>

item.h

```
typedef void* Item;

Item inputItem();
void outputItem(Item);
int cmpItem(Item,Item);
Item cloneItem(Item);
void libera(Item const);
```

coda.h

```
#include <stdbool.h>
#include "item.h"

typedef struct coda *Coda;

Coda newQueue();
bool isEmpty(Coda);
bool enqueue(Coda, Item const);
Item dequeue(Coda);
void freeQueue(Coda);
```

coda.c

```
#include <stdlib.h>
#include "coda.h"

struct coda {
   int size;
  struct node *head;
   struct node *tail;
};

struct node {
  Item item;
  struct node *next;
};

Coda newQueue() {
 Coda list = malloc(sizeof(struct coda));
 list->size = 0;
 list->head = NULL;
 list->tail = NULL;
 return list;
}

bool isEmpty(Coda list) {
 if(list != NULL) {
   return list->size == 0;
 } else
   return false;
}

bool enqueue(Coda list, Item const item) {
 if(list != NULL) {
   struct node *nodo = malloc(sizeof(struct node));
   nodo->next = NULL;
   nodo->item = cloneItem(item);
   if(list->tail != NULL)
     list->tail->next = nodo;
   list->tail = nodo;
   if(list->head == NULL)
     list->head = nodo;
   list->size++;
   return true;
 }
 return false;
}

Item dequeue(Coda list) {
 if((list != NULL) && !isEmpty(list)) {
   Item item = list->head->item;
   struct node *tmp = list->head;
   list->head = tmp->next;
   free(tmp);
   list->size--;
   return item;
 } else {
   return NULL;
 }
}

void freeQueue(Coda list) {
 if(list != NULL) {
   while(!isEmpty(list)) {
     free(dequeue(list));
   }
   free(list);
 }
}
```

L05.2. ADT STACK

Una **pila** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi esclusivamente da un unico lato (**top dello stack**). La pila è una struttura dati lineare a dimensione variabile in cui si può accedere direttamente solo al primo elemento della lista, non è possibile accedere ad un elemento diverso dal primo se non dopo aver eliminato tutti gli elementi che lo precedono (inseriti dopo) ed è una Lista gestita con la **modalità LIFO** (**Last-in-first-out**) cioè l'ultimo elemento inserito nella sequenza sarà il primo ad essere eliminato.

Tra le *possibili* implementazioni, le più usate sono realizzate tramite **Array** e **Lista concatenata**, concentriamoci sulla prima.

Lo stack è implementato come un puntatore ad una **struct stack** che contiene due elementi, un array di **MAXSTACK** elementi e un intero che indica la posizione del top dello stack. Quando lo stack si riempie, non è possibile eseguire l'operazione push.

Osserviamo adesso l'implementazione dello Stack tramite **Lista concatenata**.

Il tipo stack è definito come un puntatore ad una struct che contiene un elemento **item** di tipo **list**, non serve più nemmeno l'intero **MAXSTACK** che indica la capienza massima dello stack, anche se abbiamo un solo elemento nella struct, continuiamo a definire il tipo stack come puntatore a **struct stack** per non cambiare la definizione nell'header file.

Sintattica	Semantica
Nome del tipo: Stack Tipi usati: Item, boolean	Dominio: insieme di sequenze $S=a_1,\ldots,a_n$ di tipo Item L'elemento **nil** rappresenta la pila vuota
newStack() → Stack	newStack() → s • Post: s = nil
isEmptyStack(Stack) → boolean	isEmpty(s) → b • Post: se s=nil allora b = true altrimenti b = false
push(Stack, Item) → Stack	push(s, e) → s' • Post: s = <a1, a2, ... an> AND s' = <e, a1, ..., an>
pop(Stack) → Stack	pop(s) → s' • Pre: s = <a1, a2, ..., an> n>0 • Post: s' = <a2, ..., an>
top(Stack) → Item	top(s) → e • Pre: s = <a1, a2, ..., an> n>0 • Post: e = a1

item.h

```
#include <stdbool.h>

typedef void* Item;

Item inputItem();
void outputItem(Item const);
bool cmpItem(Item const, Item const);
Item cloneItem( Item const);
void libera(Item const);
```

pila.h

```
#include "item.h"

typedef struct pila *Pila;

Pila newStack();
bool isEmpty(Pila);
bool push(Pila, Item const);
Item top(Pila);
void pop(Pila);
void freeStack(Pila);
```

pila.c

```
#include <stdlib.h>
#include "pila.h"
#define max_elem 5
#define add_elem 5

struct pila {
  int dim;
  int size;
  Item *items;
};

Pila newStack() {
  Pila pila = malloc(sizeof(struct pila));
  pila->dim = 0;
  pila->size = max_elem;
  pila->items = malloc(max_elem * sizeof(Item));
  return pila;
}

bool isEmpty(Pila pila) {
  if(pila != NULL) {
    return pila->dim == 0;
  } else {
    return false;
  }
}

bool push(Pila pila, Item const item) {
  if((pila != NULL) && (item != NULL)) {
    if(pila->dim >= pila->size) {
      pila->size += add_elem;
      pila->items = realloc(pila->items, pila->size * sizeof(Item));
    }
    pila->items[pila->dim] = cloneItem(item);
    pila->dim++;
    return true;
  } else {
    return false;
  }
}

Item top(Pila pila) {
  if((pila != NULL) && (!isEmpty(pila))) {
    return cloneItem(pila->items[pila->dim-1]);
  } else {
    return NULL;
  }
}

void pop(Pila pila) {
  if((pila != NULL) && (!isEmpty(pila))) {
    pila->dim--;
    free(pila->items[pila->dim]);
    if((pila->size - pila->dim) > add_elem) {
      pila->size -= add_elem;
      pila->items = realloc(pila->items, pila->size * sizeof(Item));
    }
  }
}

void freeStack(Pila pila) {
  if(pila != NULL) {
    while(!isEmpty(pila)) {
      pop(pila);
    }
    free(pila);
  }
}
```