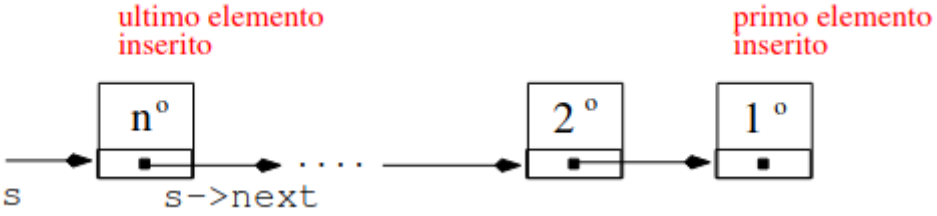
54. Implementazione di TDA con liste concatenate
Rispetto all'implementazione con array, in cui serve sapere la dimensione a priori, con la lista concatenata è possibile incrementare il TDA e aggiungere nuovi elementi fino all'esaurimento della memoria.

Le liste concatenate usando la memoria dinamica permettono di fare le varie operazioni sui TDA in maniera molto efficace. Infatti una lista concatenata è una struttura dati dinamica, la cui struttura si evolve con l'immissione e estrazione di elementi.

54.1 Implementazione di una pila come lista concatenata



Dati: una lista concatenata *s* di *n* elementi
- *s* punta all'ultimo elemento inserito nella pila (inizialmente *NULL*)
  (Opzionalmente un intero *n* con il numero di elementi nella lista)
- l'ultimo elemento della lista contiene il primo elemento inserito.
  - pila vuota: s==NULL
  - pila piena: out of memory

Numero di elementi contenuti nella pila limitato dalla memoria.

! *s* punta costantemente all'ultimo elemento inserito, ovvero il primo da estrarre !

N.B.! Rispetto agli array sono allocati solo gli *n* nodi necessari a contenere gli elementi.

Funzionalità delle funzioni
- *init()*: pone *s=NULL*
- *push(T)*:
  1. alloca un nuovo nodo ad un puntatore *tmp*
  2. copia l'elemento in *tmp->value*
  3. assegna *tmp->next=s*, e *s=tmp*
- *pop()*:
  1. fa puntare un nuovo puntatore *first* al primo nodo: *first=s*
  2. *s* aggira il primo nodo: *s=s->next*
  3. dealloca (l'ex) primo nodo: *delete first*
- t*op(T &)*: restituisce *s->value*
- *deinit()*: ripete *pop()* finché la pila non è vuota

Casi limite: solo lista vuota.

54.1.2 Funzioni
La firma delle funzioni è analoga a quella dell'implementazione con array:
void init(stack &);
void deinit(stack &);
retval push (int, stack &);
retval top (int &, const stack &);
retval pop (stack &);
void print(const stack &);

L’unica differenza è contenuta nella definizione della struttura:

```
struct nodo{
int val;
nodo* next;     //al posto di un array
}
```

Nel *main* definiamo un puntatore alla struttura nodo: ***nodo * s;*** che dovrà essere passato in alcuni casi per riferimento (***T funz(nodo* & s){}***), in altri per valore (***T funz(nodo*s){}***).
Per comodità: *typedef nodo*;*

init() e *deinit()*

```
void init(stack & s){
s=NULL;
}

void deinit(stack & s){
while(!empty(s)){          //verifica che la lista non sia vuota, verificando s==NULL (ritorna bool)
pop(s);                    //metodo burino: while(pop(s)==OK){};
}
}
```

top() e *pop()*

```
retval top(int & n, const stack & s){
retval res;
if(empty(s)){
res=FALSE;
} else {
n=s→val;
/*      nodo * first = s;          //
        s=s→next;                  // se vogliamo unire top e pop
/*      delete first;              //
res=OK;
}
return res;
}

retval pop(stack & s){
retval res;
if(empty(s)){
res=FALSE;
} else {
nodo * first = s;
s=s->next;
delete first;
res=OK;
}
return res;
}
```

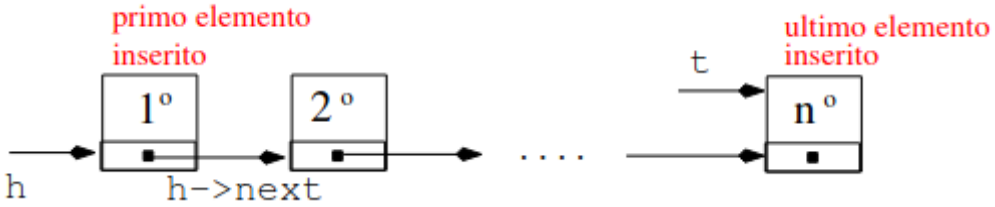
print()

```
void print(const stack& s){
nodo* first=s;
while(first!=NULL){
cout << first->val << " ";
first=fist->next;
}
}
```

push()

```
retval push(int n, stack& s){
retval res;
nodo * np = new (nothrow) nodo;          //uso nothrow così assegna NULL a np se non c'è mem
if(np==NULL){
res=FAIL;
} else {
np->val = n;
np->next=s;
s = np;
res=OK;
}
return res;
}
```

54.2 Code (o queue) implementate con liste concatenate



Dati: una lista concatenata *h* di *n* elementi di tipo *T*, un puntatore *t* all'ultimo elemento

- *h* punta al primo elemento inserito nella coda (inizialmente *NULL*)
- *t* punta all'ultimo elemento inserito nella coda
- Opzionalmente un intero *n* con il numero di elementi nella lista
- coda vuota: *h==NULL*
- coda piena: *out of memory*

Numero di elementi contenuti nella coda limitato dalla memoria.
N.B.! Sono allocati solo gli *n* nodi necessari a contenere gli elementi.

Funzionalità delle funzioni

- *init()*: pone *h=NULL*
- *enqueue(T)*:
  1. alloca un nuovo nodo ad un puntatore *tmp*
  2. copia l'elemento in *tmp->value* e pone *tmp->next=NULL*
  3. (se coda non vuota) assegna *t->next=tmp*, e *t=tmp*
     (se coda vuota) assegna *h=tmp*, e *t=tmp*
- *dequeue()*: come *pop()* della pila con il puntatore *h*
- *first(T &)*: come *top()* della pila con il puntatore *h*
- *deinit()*: ripete *dequeue()* finché la coda non è vuota

Casi limite: solo coda vuota in *dequeue()* e *first()*

55.1.2 Funzioni

Le firme delle funzioni sono le stesse nel caso delle code implementate tramite array:

```
void init (queue &);
void deinit (queue &);
retval enqueue(int,queue &);
retval first(int &,const queue &);
retval dequeue(queue &);
void print (const queue &);
```

In questo caso definiamo due struct (la prima ci servirà nel modulo, la seconda ci servirà nel *main* per definire la queue all'inizio, *queue q;*).

```
struct node {
int val;
node * next;
};

struct queue {
  node * tail;          //sono puntatori!
  node * head;
};
```

init() e *deinit()*

```
void init(const queue & q){
   q.head=NULL;
}

void deinit(queue & q){
while(!empty(q)){          //verifica che q non sia vuoto, verificando q.head == NULL (ritorna bool)
dequeue(q);
}
}
```

enqueue()

```
retval enqueue(int n, queue & q){
retval res;
node * np = new (nothrow) node;
if(np==NULL){
res=FAIL;
} else {
np->val=n;
np->next=NULL;

if(empty(q)){
 q.head=np;
} else {
 q.tail->next=np;
}
q.tail=np;
rest=OK;
}
return res;
}

retval dequeue(queue & q){
retval res;
if(empty(q)){
res=FAIL;
} else {
node * first = q.head;
q.head=q.head->next;
delete first;
res=OK;
}
return res;
}
```

first()

```
retval first(int & n, const queue & q){
retval res;
if(empty(q)){
res=FAIL;
} else {
```

```
n = q.head→val;
// node * first = q.head;          //
// q.head=q.head→next;             //first e dequeue unite
// delete first;                   //
res = OK;
}
return res;
}
```

print()

```
void print(const queue & q){
if(!empty(q)){
node * first = q.head;
do{
cout << first->val << endl;
first = first->next;
} while (first!=NULL);
}
}
```

56. Esempi di utilizzo di pile: da notazione infissa a postfissa

```
for(int i = 0; e[i] != '\0'; i++) {        e contiene l'espressione

   if ((e[i] == '(') || (e[i] == ' ')) continue;  //non fa nulla con spazi vuoti o con )

   if (e[i] == ')') {
    if (Top(s, c)) {                    //con ) se c'è memorizzato qualcosa lo stampa
         Pop(s);
         cout << c << " ";
    }
   }

   if ((e[i] == '+') || (e[i] == '-') || (e[i] == '*') || (e[i] == '/')) {
    if (!Push(s, e[i])) cout << "ERRORE";      //errore se pila piena
   }

   if ((e[i] >= '0') && (e[i] <= '9'))         //se è un numero lo stampa
    cout << e[i];

   if (((e[i] >= '0') && (e[i] <= '9')) && !((e[i+1] >= '0') && (e[i+1] <= '9')))
    cout << " ";       //spazio se il successivo non è un numero
}

// The top level parenthesis are missing
if (Top(s, c)) {
 cout << c;
}
```