UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Ingegneria
"Enzo Ferrari"

Fondamenti di Informatica II

Liste

AGENDA

- **Il concetto di Lista**
- **Definizione di tipo di dato astratto - ADT**
- **ADT Lista**
- **Rappresentazione concreta di Lista**
- **Costruzione dell'ADT elemento lista**

IL CONCETTO DI LISTA

Una lista è una sequenza (multi-insieme finito e ordinato) di elementi di un fissato tipo.

Per denotare strutture a lista useremo la notazione a '[ ]'

L = [elemento, elemento, … , elemento]

ESEMPIO

['a', 'b', 'c'] denota la lista dei caratteri 'a', 'b', 'c'

[5,8,5,21,8] denota la lista di 5 numeri interi

Un lista è un **multi-insieme**, cioè un insieme con elementi che si possono ripetere.

E' un tipo di dato astratto, definito in termini di:

Dominio dei suoi elementi (dominio base)

Operazioni di ***costruzione*** sul tipo lista

Operazioni di ***selezione*** sul tipo lista

L'ADT (ABSTRACT DATA TYPE) LISTA

In generale, un tipo di dato astratto T è definito come:

- Un dominio base D
- Un insieme di funzioni F = {F1, F2, …} sul dominio D
- Un insieme di predicati P = {P1, P2, …} sul dominio D

T = {D, F, P}

Un lista semplice è un tipo di dato astratto tale che:

- D può essere qualunque
- F = {cons, head, tail, emptylist}

  cons: D x List -> List (costruttore in testa)
  head: List -> D (selettore testa della lista)
  tail: List -> List (selettore coda della lista)
  emptylist: -> List (creazione lista 'vuota')
- P = {isempty}

  isempty List -> bool (test di lista vuota)

L'ADT (ABSTRACT DATA TYPE) LISTA

Esempi di funzioni:

- head([6, 7, 11, 21, 3, 6]) = 6
- tail([6, 7, 11, 21, 3, 6]) = ([7, 11, 21, 3, 6])
- cons(6, [7, 11, 21, 3, 6]) = [6, 7, 11, 21, 3, 6]
- L = cons(head(L), tail(L))

Esempi di predicati:

- isempty([6, 7, 11, 21, 3, 6]) = false
- isempty([ ]) = true

Pochi linguaggio forniscono il tipo lista tra quelli predefiniti (LISP, PROLOG). Per gli altri l'ADT lista si costruisce a partire dalle strutture dati esistenti (in C con vettori e puntatori)

L'ADT (ABSTRACT DATA TYPE) LISTA

Concettualmente, le operazioni precedenti costituiscono l'insieme minimo completo per operare sulle liste.

Tutte le altre operazioni, ad esempio inserimento ordinato, concatenamento, ribaltamento, etc, si posono definire in termini delle primitive precedenti.

DEFINIZIONE INDUTTIVA DI LISTA

- Esiste una costante Lista vuota (funzione emptylist)
- È fornito un costruttore (cons) che, dato un elemento e una lista, produce una nuova lista.

Questa caratteristica rende naturale esprimere le **operazioni derivate** (non primitive) mediante algoritmi ricorsivi

RAPPRESENTAZIONE CONCRETA DI LISTA: STATICA

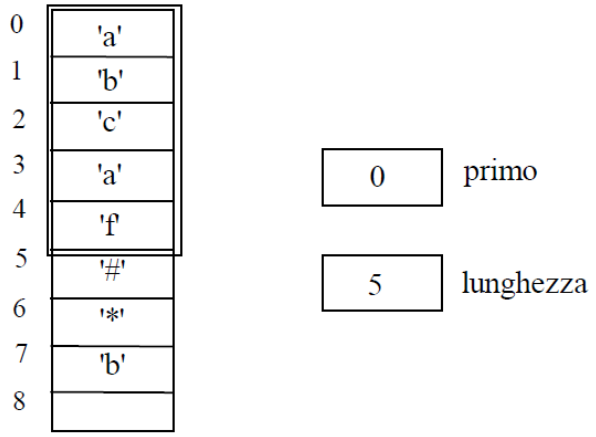
Una prima rappresentazione consiste nell'utilizzare un vettore per memorizzare gli elementi della lista uno dopo l'altro (rappresentazione sequenziale)

- La variabile primo memorizza l'indice del vettore in cui è inserito il primo elemento
- La variabile lunghezza indica da quanti elementi è composta la lista

Le componenti con indice pari o successiva a primo+lunghezza non sono significative.

Esempio

['a', 'b', 'c', 'a', 'f']



Inconvenienti:

- Le dimensioni sono fisse
- Le operazioni di insert e delete sono dispendiose

RAPPRESENTAZIONE CONCRETA DI LISTA: RAPPRESENTAZIONE COLLEGATA

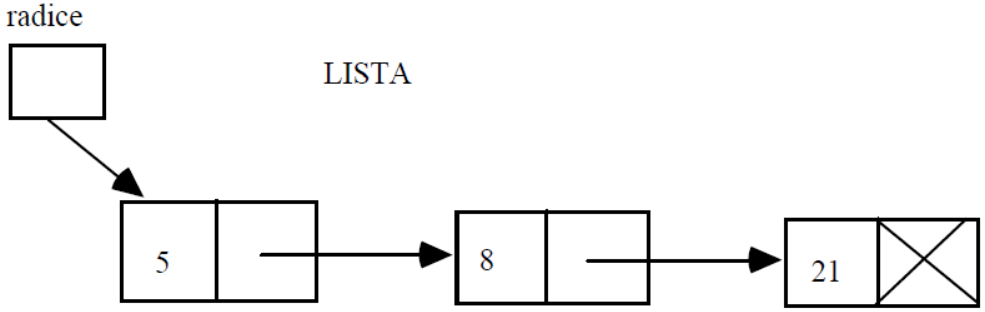
In questa seconda rappresentazione, a ogni elemento si associa l'informazione ('indice', 'riferimento') che permette di individuare la posizione del successivo (rappresentazione collegata)

La sequenzialità degli elementi della lista non è più rappresentata mediante l'adiacenza delle locazioni di memoria.

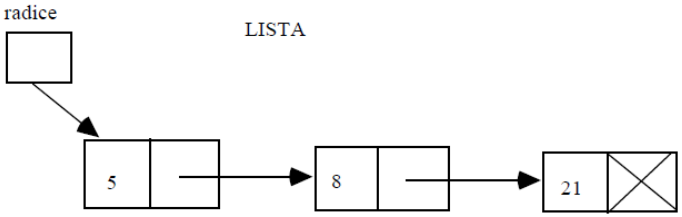
NOTAZIONE GRAFICA

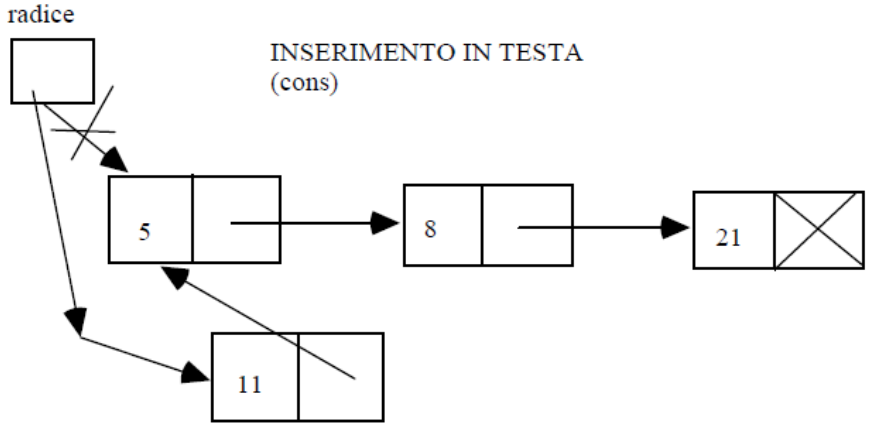
- Elementi della lista come nodi
- Riferimenti (indici) come archi

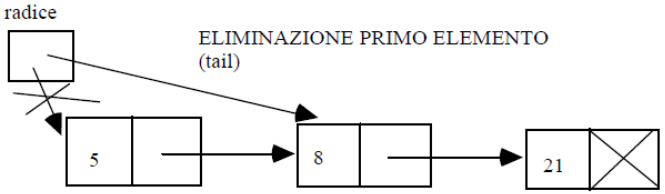
Ad esempio [5, 8, 21] risulta così rappresentata:

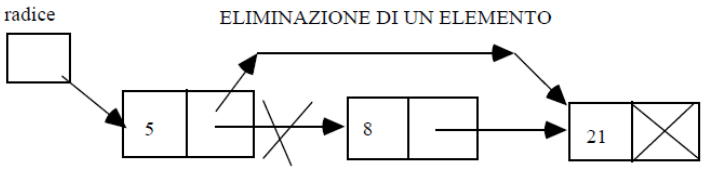


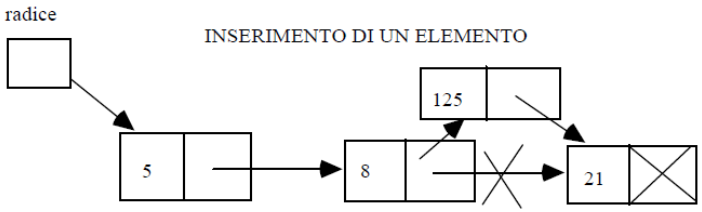
RAPPRESENTAZIONE COLLEGATA: GLI OPERATORI











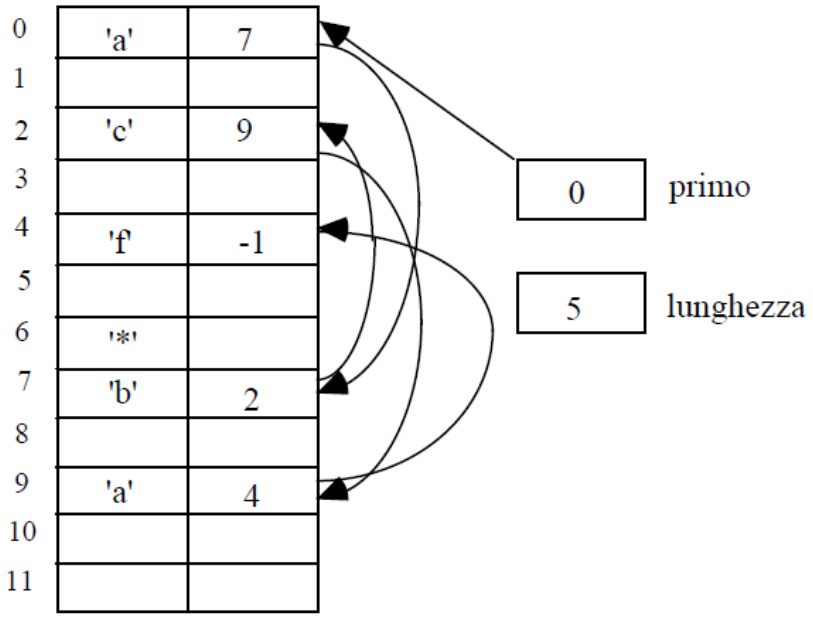
RAPPRESENTAZIONE CONCRETA DI LISTA: IMPLEMENTAZIONE A VETTORI

Ogni elemento del vettore deve mantenere:

- Il valore dell'elemento della lista (dato)
- Un riferimento (indice) al prossimo elemento (next)

Esempio

['a', 'b', 'c', 'a', 'f']



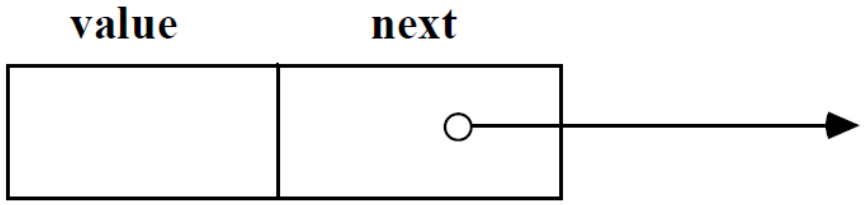
VANTAGGI: cancellazione e inserimento sono più efficienti (non occorre spostare elementi in memoria)

SVANTAGGI: si occupa più spazio e il limite è fisso, e occorre gestire le celle 'libere'

RAPPRESENTAZIONE COLLEGATA MEDIANTE PUNTATORI

Per non avere limiti alla dimensione della lista, si procede attraverso allocazione dinamica della memoria.

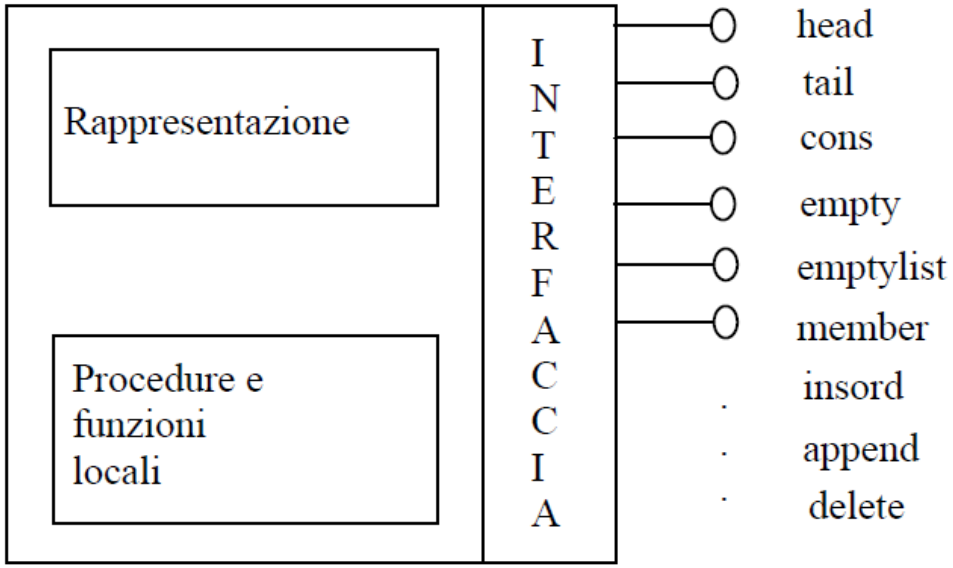
```
struct Item {
    ElemType value;
    struct Item *next;
};
typedef struct Item Item;
```



*Item verrà usato per riferirsi al primo elemento della lista

COSTRUZIONE ADT LISTA

E' conveniente incapsulare la rappresentazione concreta (che utilizza puntatori e strutture) ed esportare solo le dichiarazioni:



Poiché il funzionamento di una lista non dipende dal tipo di elementi di cui è composta, impostiamo una **soluzione generica**

COSTRUZIONE ADT LISTA (2)

- Definiamo un tipo dato `ElemType` per rappresentare il generico tipo di elemento (con le sue proprietà)
- Realizziamo l'ADT (funzioni e predicati) LISTA in termini di `ElemType`

TIPO **ElemType**

```
typedef int ElemType;
```

Su ElemType definiamo funzioni per:

- Verificare la relazione d'ordine tra due elementi
- Verificare l'eguaglianza tra due elementi
- Generare un nuovo elemento tramite copia
- Leggere un elemento da stdin
- Scrivere un elemento su stdout

ADT ElemType

- Dichiarazioni

```
typedef int ElemType;
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
ElemType ReadStdinElem();
void WriteStdoutElem(const ElemType *e);
```

- Definizioni

```
int ElemCompare(const ElemType *e1, const ElemType
*e2){
    return (*e1 > *e2) - (*e1 < *e2); }
ElemType ElemCopy(const ElemType *e){
    return *e;}
```

ADT ElemType (2)

- Definizioni

```
ElemType ReadElem(FILE *f){
    ElemType e;
    fscanf(f, "%d", &e);
    return e; }

ElemType ReadStdinElem(){
    return ReadElem(stdin);}

void WriteElem(const ElemType *e, FILE *f) {
    printf("%d", *e);}

void WriteStdoutElem(const ElemType *e){
    WriteElem(e, stdout);}
```

ADT LISTA

```
struct Item{
    ElemType value;
    struct Item *next;};
typedef struct Item Item;
// primitive
Item* CreateEmptyList(void);
Item* InsertHeadList(const ElemType *e, Item* i);
const ElemType* GetHeadValueList(Item *i);
Item* GetTailList(const Item* i);
Item* InsertBackList(Item* i, const ElemType *e);
void DeleteList(Item* item);
bool IsEmptyList(const Item *i);
// non primitive
void WriteStdoutList(const Item *i);
```

ADT LISTA (2)

```
Item* CreateEmptyList(void){
    return NULL;}

Item* InsertHeadList(const ElemType *e, Item* i){
    Item *t = malloc(sizeof(Item));
    t->value = ElemCopy(e);
    t->next = i;
    return t; }

Const ElemType* GetHeadValueList(Item *i){
    if (IsEmptyList(i)) {
        printf("ERROR: Alla funzione 'GetHeadList()' è stata passata
                        una lista vuota (NULL pointer).\n");
        exit(1); }
    else { return &i->value; }
}
```

ADT LISTA (3)

```
Item* GetTailList(const Item* i){
    if (IsEmptyList(i)) {
        printf("ERROR: Alla funzione 'GetTail()' è stata passata una
                        lista vuota (NULL pointer).\n");
        exit(2);
    }
    else { return i->next; } }


Item* InsertBackList(Item* i, const ElemType *e) {
    Item* n = InsertHeadList(e, CreateEmptyList());
    if (IsEmptyList(i)) { return n; }

    Item* tmp = i;
    while (!IsEmptyList(GetTailList(tmp))) {
        tmp = GetTailList(tmp);}
    tmp->next = n;
    return i;
}
```

ADT LISTA (4)

```
void DeleteList(Item* item) {
    while (!IsEmptyList(item)) {
        Item* tmp = item;
        item = item->next;
        ElemDelete(&tmp->value);
        free(tmp); }
}

bool IsEmptyList(const Item *i){
    return i == NULL;}

// non primitive
void WriteList(const Item *i, FILE *f) {
    printf("[");
    while (!IsEmptyList(i)) {
        WriteElem(&i->value, f);
        i = GetTailList(i);
        if (!IsEmptyList(i)) printf(", ");}
    printf("]\n"); }

void WriteStdoutList(const Item *i) {
    WriteList(i, stdout); }
```

COSTRUZIONE LISTA DA PRIMITIVE

```
Item* CreateListFromVector(const int *v, size_t v_size) {
    Item *list = CreateEmptyList();
    for (size_t i = 0; i < v_size; ++i) {
        list = InsertBackList(list, &v[i]);
    }
    return list;
}

int main(void) {

    int v[] = { 1,2,3,4,5,6,7,8,9 };
    size_t v_size = sizeof(v) / sizeof(int);
    Item *list = CreateListFromVector(v, v_size);
    Item* tmp = list;
    while (tmp) {
        ElemType e = tmp->value;
        // to do
        tmp = tmp->next;
    }
    DeleteList(list);
    return EXIT_SUCCESS;
}
```

INSERIMENTO ORDINATO IN LISTA

```
Item* InsOrdRec(const ElemType *e, Item *i) {
    if (IsEmptyList(i) || ElemCompare(GetHeadValueList(i), e) >= 0) {
        return InsertHeadList(e, i); }

    Item* tmp = InsertHeadList(GetHeadValueList(i), InsOrdRec(e, GetTailList(i)));
    free(i);
    return tmp;
}

Item* InsOrd(const ElemType *e, Item *i) {
    if (IsEmptyList(i) || ElemCompare(GetHeadValueList(i), e) >= 0) {
        return InsertHeadList(e, i);  }
    Item* root = i;
    Item* prev = CreateEmptyList();
    Item* new_item = InsertHeadList(e, CreateEmptyList());
    while (!IsEmptyList(i) && ElemCompare(GetHeadValueList(i), e) < 0) {
        prev = i;
        i = GetTailList(i); }
    prev->next = new_item;
    new_item->next = i;
    return root;
}
```

INSERIMENTO ORDINATO IN LISTA (MAIN)

```
int main(void)
{
    Item* i = CreateEmptyList();
    Item* i_rec = CreateEmptyList();
    ElemType e;
    // Acquisizione di elementi da stdin
    do {
        printf("Introdurre un valore intero: ");
        e = ReadStdinElem();
        i = InsOrd(&e, i);
        i_rec = InsOrdRec(&e, i_rec);
    } while (e != 0); // L'acquisizione termina quando viene inserito lo zero.

    WriteStdoutList(i);
    WriteStdoutList(i_rec);
    DeleteList(i);
    DeleteList(i_rec);
    return EXIT_SUCCESS;
}
```

FUNZIONE IS MEMBER

```
bool IsMember(const ElemType *e, Item* i) {
    while (!IsEmptyList(i)) {
        if (ElemCompare(e, GetHeadValueList(i)) == 0) {
            return true; }
        i = GetTailList(i);
    }
    return false;
}

bool IsMemberRec(const ElemType *e, Item* i) {
    if (IsEmptyList(i)) {
        return false; }

    if (ElemCompare(e, GetHeadValueList(i)) == 0) {
        return true;}

    return IsMemberRec(e, GetTailList(i));
}
```

FUNZIONE LENGHT

```
int Lenght(Item* i) {
    int n = 0;
    while (!IsEmptyList(i)) {
        n++;
        i = GetTailList(i);
    }
    return n;
}

int LenghtRec(Item* i) {
    if (IsEmptyList(i)) {
        return 0; }

    return 1 + LenghtRec(GetTailList(i));
}
```

FUNZIONE APPEND

APPEND(L1, L2) =

L2 se empty(L1)

CONS(HEAD(L1),APPEND(TAIL(L1), L2)) altrimenti

```
Item* AppendRec(Item* i1, Item* i2) {
    if (IsEmptyList(i1)) {
        return i2;
    }

    Item* tmp = InsertHeadList(GetHeadValueList(i1),
                               AppendRec(GetTailList(i1), i2));
    free(i1);
    return tmp;
}
```

FUNZIONE COPY

COPY(L) =

L	se empty(L)
CONS(HEAD(L),COPY(TAIL(L))	altrimenti

```
Item* CopyRec(Item* i) {
    if (IsEmptyList(i)) {
        return i;
    }
    return InsertHeadList(GetHeadValueList(i), CopyRec(GetTailList(i)));
}
```

FUNZIONE REMOVE

$$
\text{REMOVE}(e,L) = \begin{cases} L & \text{se empty}(L) \\ \text{TAIL}(L) & \text{se } e = \text{head}(L) \\ \text{CONS}(\text{HEAD}(L), \text{REMOVE}(e, \text{TAIL}(L))) & \text{altrimenti} \end{cases}
$$

```
Item* RemoveRec(const ElemType* e, Item* i) {
    if (IsEmptyList(i)) {
        return i;}
    if (ElemCompare(e, GetHeadValueList(i)) == 0) {
        Item* tmp = GetTailList(i);
        free(i);
        return tmp;}
    Item* tmp = InsertHeadList(GetHeadValueList(i), RemoveRec(e, GetTailList(i)));
    free(i);
    return tmp;
}
```