TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Linguaggio C: Liste

Valeria Cardellini

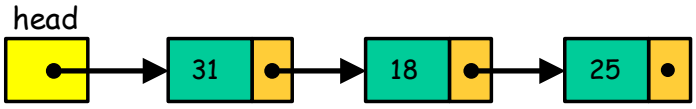
Corso di Calcolatori Elettronici
A.A. 2019/20

Argomenti

- Definizione di lista in C
- Operazioni su lista: inserimento, cancellazione, ricerca di un elemento

Lista concatenata

- Struttura dati in cui ogni nodo ha un collegamento (link) ad un altro nodo: lista concatenata (*linked list*)
- Il collegamento è un puntatore con l'indirizzo di memoria dell'altro nodo
  - Si accede alla lista per mezzo di un puntatore al suo primo nodo (*head*)
  - Si accede ai nodi successivi per mezzo del puntatore memorizzato in ogni nodo
  - Il puntatore dell'ultimo nodo è impostato a NULL
- In una lista i dati sono memorizzati in modo dinamico



Strutture auto-referenzianti

- Struttura *auto-referenziante* (self-referencing): la struttura contiene un puntatore ad una struttura dello stesso tipo
- Definiamo il nodo di una lista (il tipo del dato non è specificato)

```
struct listNode {
  <type> data; // Ogni nodo contiene un dato
  struct listNode *next; // Puntatore al prossimo nodo
};
```

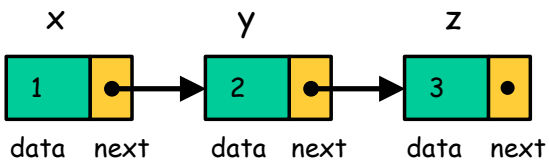
  - `next` è un puntatore al tipo di oggetto che si sta dichiarando

Esempio: strutture auto-referenzianti

- Esempio:

```
struct listNode {
  int data; // Ogni nodo contiene un intero
  struct listNode *next; // Puntatore al prossimo nodo
} x, y, z;

x.data = 1;
y.data = 2;
z.data = 3;
x.next = &y;
y.next = &z;
z.next = NULL;
```



Esempio: strutture auto-referenzianti

- Attenzione: i nodi delle liste concatenate non sono generalmente memorizzati in modo contiguo

```
//Vedi esempio in slide precedente
struct listNode *ptr = &x; /* ptr punta alla struttura x */
ptr++; /* ptr punta alla locazione di memoria
          ptr + sizeof(x) */
ptr = ptr->next;   /* ptr punta ad y */
```

Gestione di liste semplici

- Esaminiamo due modi alternativi per la gestione di liste semplici

1. Direttamente con struct e puntatori
   - Codice sorgente: lista.c
2. Usando typedef per definire il tipo di dato per un nodo della lista
   - Codice sorgente: lista_typedef.c

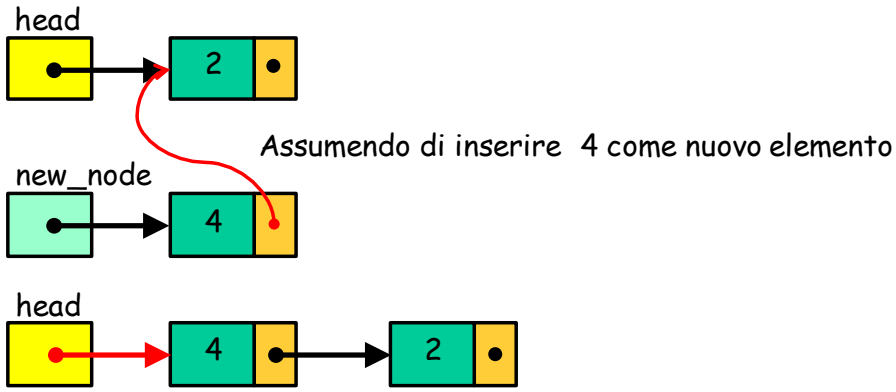
Allocazione di memoria

- Se `new_node` è una variabile di tipo `struct listNode *`, chiamando la funzione `malloc` si ottiene dal sistema una porzione di memoria adeguata a contenere un nodo e si inizializzano i relativi campi

```
new_node = malloc(sizeof(struct listNode));
if (new_node == NULL) {
   printf("Impossibile allocare memoria!\n");
   exit(EXIT_FAILURE); }
new_node->data = new_value;
new_node->next = NULL;
```

Creazione di un nodo ed inserimento in testa

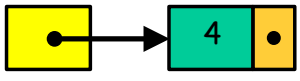
- Si alloca la nuova struct listNode da inserire in testa
- Si memorizza il valore nel nuovo nodo
- Si fa puntare il nuovo nodo alla precedente testa della lista
- Si fa puntare la testa della lista al nuovo nodo



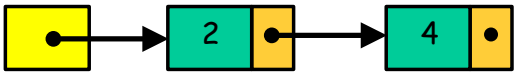
Creazione di un nodo ed inserimento in testa

```
void insert_head(struct listNode **head_ptr, int new_value)
{
  struct listNode *new_node; // Puntatore al nodo da inserire
  new_node = malloc(sizeof(struct listNode));
  if (new_node == NULL) {
    printf("Impossibile allocare memoria!\n");
    exit(EXIT_FAILURE);
  }
  new_node->data = new_value;
  new_node->next = *head_ptr; // Inserimento in testa
  *head_ptr = new_node; // Aggiornamento puntatore alla testa
}
```

Assumendo new_value = 4



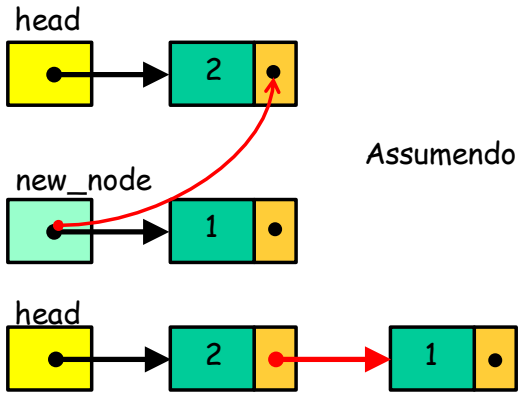
Assumendo new_value = 2



Codice sorgente: lista.c

Creazione di un nodo ed inserimento in coda

- Si alloca la nuova struct listNode da inserire in coda
- Si memorizza il valore nel nuovo nodo
- Se la lista è vuota, si aggiorna il puntatore alla testa
- Altrimenti si scorre la lista fino a raggiungere l'ultimo nodo e si inserisce in coda il nuovo nodo, aggiornando il puntatore del nodo che lo precede



Assumendo di inserire 1 come nuovo elemento

Creazione di un nodo ed inserimento in coda

```
void append(struct listNode **head_ptr, int new_value){
  struct listNode *current;   /* Puntatore al nodo corrente */
  struct listNode *new_node;  /* Puntatore al nuovo nodo da
appendere */
  new_node = malloc(sizeof(struct listNode));
  if (new_node == NULL) {
    printf("Impossibile allocare memoria!\n");
    exit(EXIT_FAILURE);       }
  new_node->data = new_value;
  new_node->next = NULL;
  current = *head_ptr;
  if (current == NULL) {      /* Lista vuota */
    *head_ptr = new_node;     /* Aggiornamento del puntatore
alla testa della lista */
    return;    }
```

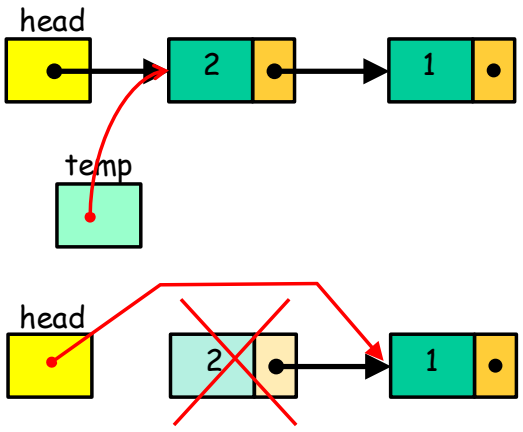
Codice sorgente: lista.c

Creazione di un nodo ed inserimento in coda

```
  /* Scansione della lista fino all'ultimo nodo */
  while (current->next  != NULL)
    current = current->next;
    current->next  = new_node; /* Inserimento del nuovo nodo */
}
```


Rimozione del nodo in testa alla lista

- Si salva il puntatore al primo nodo da rimuovere
- Si fa puntare la testa della lista al secondo nodo
- Si dealloca (usando `free`) lo spazio del primo nodo



Rimozione del nodo in testa alla lista

La funzione restituisce 1 se la rimozione va a buon fine, 0 altrimenti

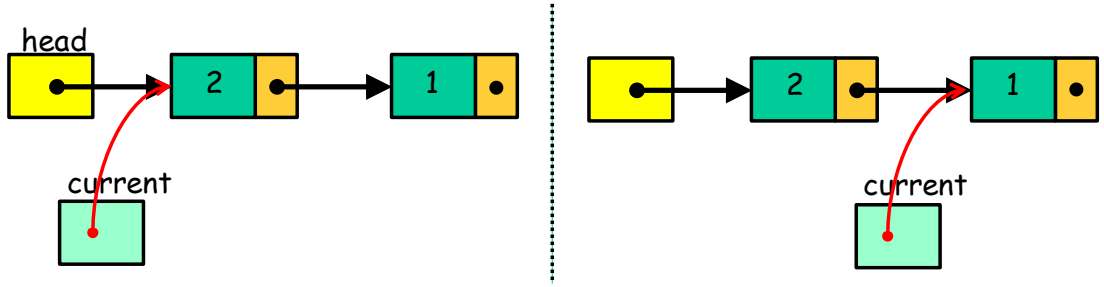
```
int delete_head(struct listNode **head_ptr)
{
  struct listNode *temp;
  if (*head_ptr == NULL) /* Lista vuota */
    return 0;

  temp = *head_ptr; /* Salva il puntatore al primo nodo da
rimuovere */
  /* Cancellazione del primo nodo */
  *head_ptr = current->next;
  free(temp);
  return 1;
}
```

Codice sorgente: lista.c

Stampa della lista

- Si scorre la lista nodo per nodo, stampando il valore del nodo corrente
- Si aggiorna il puntatore al nodo corrente con quello al nodo successivo, finché non si raggiunge la fine della lista (NULL)



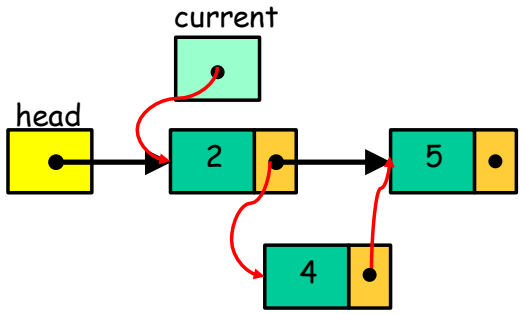
Stampa della lista

```
void print_list(struct listNode *head)
{
  struct listNode *current;
  if (head == NULL) /* La lista è vuota */
    printf("La lista è vuota\n\n");
  else {
    printf("La lista è:\n");
    /* Stampa nodo per nodo fino alla fine della lista */
    for (current=head; current != NULL; current=current->next)
      printf("%d --> ", current->data);
    printf("NULL\n\n");
  }
}
```


Creazione di un nodo ed inserimento ordinato

- Inserimento ordinato dei valori (ordinamento crescente)
  - In un ciclo, occorre scorrere la lista fino a trovare il nodo prima del quale deve essere inserito il nuovo nodo, ossia fin quando il valore da inserire risulta maggiore del valore del nodo corrente
    - ✓ Si usano un puntatore, `current`, che punta al nodo prima del quale inserire il nuovo nodo
  - Se la lista è vuota, bisogna far puntare la testa della lista al nuovo nodo, altrimenti inserire il nuovo nodo dopo `current`

Assumendo di inserire 4 come nuovo nodo



Creazione di un nodo ed inserimento ordinato

```
void insert_sorted(struct listNode **head_ptr, int new_value)
{
  struct listNode *current; /* Puntatore al nodo corrente */
  struct listNode *new_node; /* Puntatore al nuovo nodo da inserire */

  new_node = malloc(sizeof(struct listNode));
  if (new_node == NULL) {
    printf("Impossibile allocare memoria!\n");
    exit(EXIT_FAILURE);
  }
  new_node->data = new_value;
  current = *head_ptr;
  if (current == NULL) { //La lista è vuota, inserisci in testa
    new_node->next = *head_ptr;
    *head_ptr = new_node;
    return;
  }
```

Codice sorgente: lista.c

Creazione di un nodo ed inserimento ordinato

```
  /* Inserimento in modo ordinato */
  while (current->next != NULL && current->next->data < new_value)
    current = current->next;
    if (*head_ptr == current && current->data > new_value) {
      //Elemento da inserire in testa
      new_node->next = current;
      *head_ptr = new_node;
    }
    else {
      new_node->next = current->next;
      current->next = new_node;
    }
}
```

Codice sorgente: lista.c

Rimozione di un nodo

- Rimozione di un nodo
  - Se è da rimuovere la testa della lista
    - ✓ Si salva il puntatore al nodo da rimuovere
    - ✓ Si fa puntare la testa della lista al secondo nodo
    - ✓ Si dealloca lo spazio del primo nodo
  - Altrimenti, in un ciclo occorre scorrere la lista fino a trovare il nodo da rimuovere
    - ✓ Si usano due puntatori, `previous` e `current`, che puntano rispettivamente al nodo precedente e al nodo corrente
  - Se il nodo viene trovato
    - ✓ Si salva il puntatore al nodo da rimuovere
    - ✓ Si fa puntare il nodo precedente al nodo successivo a quello da rimuovere
    - ✓ Si dealloca lo spazio del nodo

Rimozione di un nodo

Il valore del nodo da rimuovere viene passato come parametro in ingresso
La funzione restituisce 1 se la rimozione va a buon fine, 0 altrimenti

```
int delete(struct listNode **head_ptr, int value)
{
  struct listNode *current;  /* Puntatore al nodo corrente */
  struct listNode *previous; /* Puntatore al nodo precedente */
  if (*head_ptr == NULL) /* Lista vuota */
    return 0;

  current = *head_ptr;
  if (current->data == value) { /* Cancellazione del primo nodo */
    *head_ptr = current->next; /* Aggiorna il puntatore alla testa */
    free(current);
    return 1;
  }
```

Codice sorgente: lista.c

Rimozione di un nodo

```
  /* Scansione della lista a partire dal secondo nodo */
  previous = current;
  current = current->next;
  while (current != NULL) {
    if (current->data == value) {
      previous->next = current->next; /* Aggiornamento del
puntatore al nodo successivo nel nodo che precede quello da
cancellare */
      free(current);
      return 1;
    }
    previous = current;
    current = current->next;
  }
  return 0;
}
```

Codice sorgente: lista.c

- Analizziamo la seconda versione in cui si usa typedef
  - Il valore del nodo è di tipo char

Tipo di dato lista

```
typedef <tipo> dataType;     // definito dal programmatore
struct listNode {
  dataType data;
  struct listNode *next;
};
typedef struct listNode ListNode; // sinonimo di struct listNode
typedef ListNode *List;           // sinonimo di ListNode *
```

In modo equivalente:

```
typedef <tipo> dataType;     // definito dal programmatore
typedef struct listNode {
  dataType data;
  struct listNode *next;
} ListNode;
typedef ListNode *List;
```

Codice sorgente: lista_typedef.c

Tipo di dato lista

In modo equivalente:

```
typedef <tipo> dataType;     // definito dal programmatore
typedef struct ListNode *List;
typedef struct listNode {
  dataType data;
  List next;
} ListNode;
```

In modo equivalente:

```
typedef <tipo> dataType;     // definito dal programmatore
typedef struct listNode ListNode;
typedef struct ListNode *List;
struct listNode {
  dataType data;
  List next;
};
```

Allocazione di memoria

- Se head è una variabile di tipo `List` (`List head;`) chiamando la funzione `malloc`

  `head = (List)malloc(sizeof(listNode));` oppure

  `head = (List)malloc(sizeof(*head));`

  - Si ottiene dal sistema una porzione di memoria adeguata a contenere un `listNode` e si assegna l'indirizzo a `head`

Creazione di una lista con un nodo

```
typedef char itemType;
...
List newNode(itemType value) {
  List head;
  head = (List)malloc(sizeof(listNode));
  if (head != NULL) {
    head->item = value;
    head->next = NULL;
  }
  else
    printf("%c not inserted. No memory available.\n", value);
  return head;
}
```

Creazione di un nodo ed inserimento in testa

```
void insertHead(List *s, char value)
{
   List new;
   new = malloc(sizeof(ListNode));
   if (new != NULL) {    /* Is space available? */
      new->data = value;
      new->next = *s;
      *s = new;
   }
   else
      printf("%c not inserted. No memory available.\n", value);
}
```

Codice sorgente: lista_typedef.c

Rimozione dalla testa della lista

```
char deleteHead(List *s)
{
   List temp;
   char value;

   temp = *s;

   if (temp != NULL) {
     value = temp->data;
     *s = temp->next;
     free(temp);
     return value;
   }
   return '\0';
}
```

Codice sorgente: lista_typedef.c

Creazione di un nodo ed inserimento ordinato

```
/* Insert a new value into the list in sorted order */
void insertSorted(List *s, char value)
{
   List new, previous, current;

   new = malloc(sizeof(ListNode));   /* Create node */
   if (new != NULL) {    /* Is space available? */
      new->data = value;
      new->next = NULL;
      previous = NULL;
      current = *s;

      /* Loop to find the correct location in the list */
      while (current != NULL && value > current->data) {
         previous = current;          /* walk to ...   */
         current = current->next;  /* ... next node */
      }
```

Codice sorgente: lista_typedef.c

Creazione di un nodo ed inserimento ordinato

```
      /* Insert new node at beginning of list */
      if (previous == NULL) {
         new->next = *s;
         *s = new;
      }
      else {  /* Insert new node between previous and current */
         previous->next = new;
         new->next = current;
      }
   }
   else
      printf("%c not inserted. No memory available.\n", value);
}
```

Rimozione di un nodo

```
/* Delete a list element and return the deleted value */
char delete(List *s, char value)
{
   List previous, current, temp;

   /* Delete first node */
   if (value == (*s)->data) {
      temp = *s;         /* Hold onto node being removed */
      *s = (*s)->next;   /* De-thread the node */
      free(temp);        /* Free the de-threaded node */
      return value;
   }
   else {
      previous = *s;
      current = (*s)->next;
```

Codice sorgente: lista_typedef.c

Rimozione di un nodo

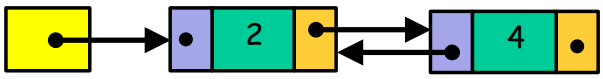
```
      /* Loop to find the correct location in the list */
      while (current != NULL && current->data != value) {
         previous = current;          /* walk to ...   */
         current = current->next;   /* ... next node */
      }

      /* Delete node at current */
      if (current != NULL) {
         temp = current;
         previous->next = current->next;
         free(temp);
         return value;
      }
   }

   return '\0';
}
```

Liste doppiamente concatenate

- Rispetto ad una lista semplicemente concatenata, nella lista doppiamente concatenata (*doubly linked list*)
  - Aggiungiamo ad ogni nodo della lista un puntatore al nodo precedente




```
struct listNode {
  int data;
  struct node *next; /* puntatore al nodo successivo */
  struct node *prev; /* puntatore al nodo precedente */
};
```


Esercizi

- Trovare l'*n*-esimo nodo in una lista semplice contando a partire dalla testa della lista ed essendo *n* >= 1 un parametro in ingresso
  - n=1: il primo nodo
  - n=2: il secondo nodo
- Cancellare l'*n*-esimo nodo in una lista semplice contando a partire dalla testa della lista ed essendo *n* >= 1 un parametro in ingresso
- Trovare l'*n*-esimo nodo in una lista semplice contando a partire dalla coda della lista ed essendo *n* >= 1 un parametro in ingresso
  - n=1: l'ultimo nodo
  - n=2: il penultimo nodo

Esercizi

- Cancellare l'*n*-esimo nodo in una lista semplice contando a partire dalla coda della lista ed essendo *n* >= 1 un parametro in ingresso
- In una lista doppiamente collegata
  - Inserire un nodo in testa
  - Inserire un nodo dopo la prima occorrenza di un nodo il cui valore è passato come parametro in ingresso
  - Cancellare un nodo il cui valore è passato come parametro in ingresso
  - Stampare la lista in entrambe le direzioni