LE STRUTTURE DATI DINAMICHE

L'allocazione dinamica

Talvolta è necessario allocare spazio di memoria durante l'esecuzione di un programma oppure creare variabili di durata temporanea.

Esempio: Leggere e memorizzare dei numeri interi inseriti da tastiera finché non si inserisce lo 0.

L'allocazione dinamica

Esiste una zona di memoria riservata a cui è possibile accedere per allocare dati durante l'esecuzione del programma.

Le variabili statiche e dinamiche sono memorizzate in zone di memoria differenti.

L'allocazione dinamica

L'allocazione dinamica avviene tra mite uso di puntatori ed è eseguita da apposite funzioni contenute nella libreria standard stdlib.h.

void *malloc(int n);

Alloca dinamicamente una zona di memoria pari a n byte e restituisce un puntatore a void che punta alla zona di memoria allocata, contenente un tipo di dato indefinito.

L'allocazione dinamica

L'allocazione dinamica avviene tramite uso di puntatori ed è eseguita da apposite funzioni.

void *calloc(int n, int dim);

Alloca dinamicamente una zona di memoria pari a n*dim, adatta a contenere n variabili ampie dim e la inizializza a 0.
Restituisce un puntatore a void che punta alla zona di memoria allocata, contenente un tipo di dato indefinito.

L'allocazione dinamica

L'allocazione dinamica avviene tramite uso di puntatori ed è eseguita da apposite funzioni.

void *realloc(void *p, int n);

Alloca dinamicamente una zona di memoria pari a n byte e vi colloca il contenuto della zona di memoria puntata da p, tagliando l'eventuale eccesso in caso di spazio insufficiente.
Restituisce p o un nuovo puntatore a void, in caso debba essere usata una nuova zona di memoria.

L'allocazione dinamica

ATTENZIONE: Tutte le funzioni che consentono l'allocazione dinamica restituiscono un puntatore NULL in caso di allocazione non riuscita (memoria insufficiente o altro tipo di errore). Questa eventualità deve essere gestita per non incorrere in errori.

Due funzioni utili

La funzione sizeof(tipo_dato)

Riceve un tipo di dato e ne restituisce la dimensione espressa in byte.

La funzione typedef oggetto nome

Assegna un alias all'oggetto, che da quel momento può essere invocato attraverso nome.

L'allocazione dinamica

Esempi:

```
int *p;
p=(int *)malloc(sizeof(int));
*p=28;
```

Alloca dinamicamente una zona di memoria pari alla dimensione di un intero e la fa puntare da p.

```
char *p;
p=(char *)malloc(80);
p="albero";
```

Alloca dinamicamente una zona di memoria pari ad 80 byte (cioè 80 caratteri).

L'allocazione dinamica

Esempi:

```
struct studente {char nome[80];
                 int voto;} *p;
p=(struct studente*)malloc(sizeof(struct studente));
```

Alloca dinamicamente una zona di memoria pari alla dimensione di una struttura di tipo studente e la fa puntare da p.

L'allocazione dinamica

Esempi:

```
struct studente {char nome[80];
                 int voto;};
typedef struct studente st;
st *p;
p=(st *)malloc(sizeof(st));
```

La de-allocazione

La memoria allocata dinamicamente persiste fino al termine del programma o finché non viene esplicitamente liberata dal programmatore.

E' possibile liberare lo spazio di memoria precedentemente allocato dinamicamente attraverso la funzione

```
free (p);
```

dove p è stato precedentemente allocato attraverso malloc, calloc o realloc.

Le strutture dati dinamiche

Sono strutture dati che possono variare la loro dimensione durante l'esecuzione del programma e possono contenere dati eterogenei. In questo corso studieremo:

Le liste concatenate

Le pile

Le code

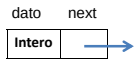
Le liste concatenate

Sono sequenze lineari (messe in fila indiana) di unità di informazione (strutture ricorsive), in cui inserimenti e cancellazioni possono avvenire in qualunque punto della lista.

Le strutture ricorsive

Sono strutture contenenti un puntatore ad oggetto dello stesso tipo.

```
struct elemento{int dato;
                struct elemento *next;};
```



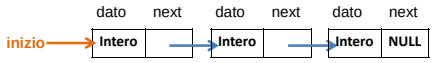
Le liste concatenate

Ogni elemento della lista è detto nodo e si crea al bisogno, mediante allocazione dinamica.

I puntatori sono detti link.

E' necessario avere un puntatore al primo nodo che indichi l'inizio della lista e che non venga spostato, altrimenti si perde la possibilità di accedere ai dati.

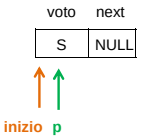
E' opportuno porre a NULL il puntatore dell'ultimo elemento in modo da segnalare la fine della lista.



Assegnazione valori ad un nodo

Dopo avere creato il nodo desiderato, si procede come nelle strutture accedute mediante puntatori.

```
struct referendum{char voto;
                  struct referendum *next;}*p, *inizio;
typedef struct referendum r;
p=(r*)malloc(sizeof(r));
p->voto='S';
p->next=NULL;
inizio=p;
```



Spostarsi in una lista

Spostarsi di un elemento

```
p=inizio;
p=p->next;
```

Si usa un puntatore ausiliario per evitare di perdere l'inizio della lista.

Scorrere l'intera lista

```
p=inizio;
while(p)
        p=p->next;
```

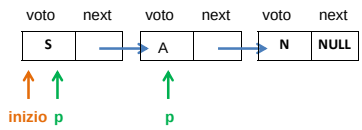
Modifica valori di un nodo

Dopo avere individuato il nodo desiderato, si procede come nelle strutture accedute mediante puntatori.

E' necessario usare un puntatore ausiliario che scorra la lista fino a posizionarsi nel nodo desiderato.

Esempio:

```
p=inizio;
p=p->next;
p->voto='A';
```



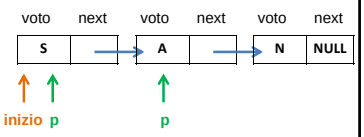
In alternativa è possibile assegnare i valori dei dati tramite scanf o copia di valori contenuti in altri dati.

Visualizzazione valori di un nodo

Dopo avere individuato il nodo desiderato, mediante l'uso di un puntatore ausiliario, si procede come nelle strutture accedute mediante puntatori.

Esempio:

p=inizio;
p=p→next;
printf("%c", p->voto);



A

Visualizzazione di una intera lista

```
#include<stdio.h>
#include<stdlib.h>
struct elemento{int dato;
                struct elemento *next;};
typedef struct elemento ele;
void stampa (ele * );
main()
{…
stampa (inizio);
 … }
```

```
stampa (ele * ini)
{while(ini)
     {printf("%d",ini->dato);
      ini=ini->next;}
}
```

ATTENZIONE: Il puntatore ini è una variabile locale. L'inizio della lista non viene spostato agendo su ini.

Contare gli elementi di una lista

```
#include<stdio.h>
#include<stdlib.h>
struct elemento{int dato;
                struct elemento *next;};
typedef struct elemento ele;
int conta (ele * );
main()
{int n; …
n=conta (inizio);
 … }
```

```
int conta (ele * ini)
{int x=0;
while(ini)
     {x++;
      ini=ini->next;}
return (x);}
```

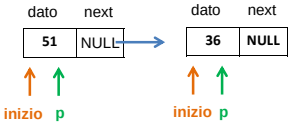
Le pile

Le pile o liste LIFO (Last Input First Output) sono particolari liste concatenate in cui inserimenti e cancellazioni avvengono solo in testa.

Creazione di una pila

Per creare una pila è necessario:

1. Creare un singolo elemento ed inserire i dati
2. Se l'elemento creato è il primo posizionare l'inizio.
   Altrimenti collegare col precedente e spostare l'inizio
1. Ripetere da 1.



Creazione di una pila

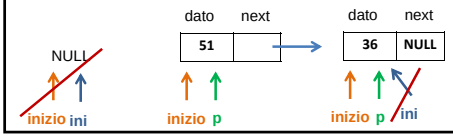
```
#include<stdio.h>
#include<stdlib.h>
struct elemento{int dato;
                struct elemento *next;};
typedef struct elemento ele;
ele * creapila(ele *, int );
main()
{ele *inizio=NULL;
…
inizio=creapila(inizio, n); … }
```

```
ele * creapila (ele * ini, int a)
{ele *p;
p=(ele *)malloc(sizeof(ele));
p->dato=a;
p->next=ini;
return(p);}
```

Creazione di una pila

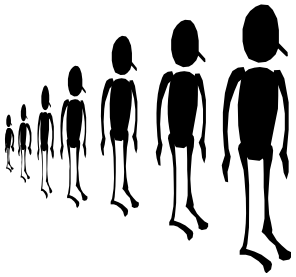
```
…
main()
{ele *inizio=NULL;
…
inizio=creapila(inizio, n); … }
```

```
ele * creapila (ele * ini, int a)
{ele *p;
p=(ele *)malloc(sizeof(ele));
p->dato=a;
p->next=ini;
return(p);}
```



Le code

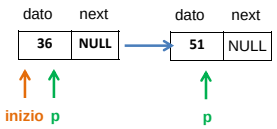
Le code o liste FIFO (Fist Input First Output) sono particolari liste concatenate in cui gli inserimenti avvengono in coda e le eliminazioni in testa.



Creazione di una coda

Per creare una coda è necessario:

1. Creare un singolo elemento ed inserire i dati
2. Se l'elemento creato è il primo posizionare l'inizio.
   Altrimenti scorrere fino alla fine della lista e collegare con l'elemento creato.
1. Ripetere da 1.



Creazione di una coda

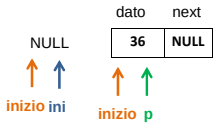
```
#include<stdio.h>
#include<stdlib.h>
struct elemento{int dato;
        struct elemento *next;};
typedef struct elemento ele;
ele * creacoda(ele *, int );
main()
{ele *inizio=NULL;
…
inizio=creacoda(inizio, n);
… }
```

```
ele * creacoda (ele * ini, int a)
{ele *p,*aux;
p=(ele *)malloc(sizeof(ele));
p->dato=a;
p->next=NULL;
if(!ini)
        return(p);
else
{aux=ini;
while(aux->next)
        aux=aux->next;
aux->next=p;
return(ini);}
```

Creazione di una coda

```
…
main()
{ele *inizio=NULL;
…
inizio=creacoda(inizio, n);
… }
```

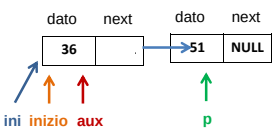
```
ele * creacoda (ele * ini, int a)
{ele *p,*aux;
p=(ele *)malloc(sizeof(ele));
p->dato=a;
p->next=NULL;
if(!ini)
        return(p);
else
{aux=ini;
while(aux->next)
        aux=aux->next;
aux->next=p;
return(ini);}
```



Creazione di una coda

```
…
main()
{ele *inizio=NULL;
…
inizio=creacoda(inizio, n);
… }
```

```
ele * creacoda (ele * ini, int a)
{ele *p,*aux;
p=(ele *)malloc(sizeof(ele));
p->dato=a;
p->next=NULL;
if(!ini)
        return(p);
else
{aux=ini;
while(aux->next)
        aux=aux->next;
aux->next=p;
return(ini);}
```



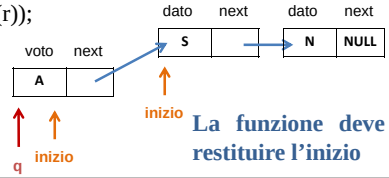
Inserimento nodo in lista concatenata

L'inserimento di un nodo in una lista può avvenire in testa (LIFO), in coda (FIFO) o in un punto intermedio della lista.

E' bene deputare gli inserimenti ad apposite funzioni.

Inserimento in testa

```
q=(r *)malloc(sizeof(r));
q->voto='A';
q->next=inizio;
inizio=q;
```

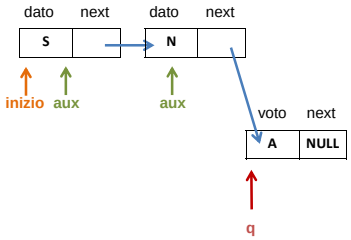


La funzione deve restituire l'inizio

Inserimento in coda

```
q=(r *)malloc(sizeof(r));
q->voto='A';
q->next=NULL;
aux=inizio;

while(aux->next)
        aux=aux->next;

aux->next=q;
```

La funzione non deve restituire nulla perché l'inizio non è stato spostato.



Inserimento nodo dopo un elemento assegnato

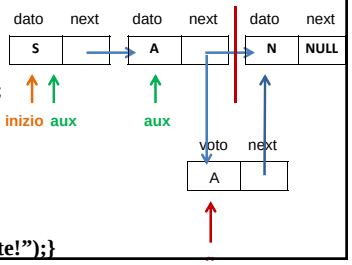
1. Scorro la lista fino a posizionarmi nel punto desiderato.

   E' necessario usare un puntatore ausiliario che scorra la lista fino a posizionarsi nel nodo in cui fare l'inserimento senza perdere l'inizio.
2. Se il punto desiderato è stato trovato
   a) Creo il nuovo elemento
   b) Creo i collegamenti.

   Per non perdere i collegamenti è necessario collegare prima il nuovo elemento al successivo della lista e poi il precedente al nuovo elemento.

Inserimento nodo dopo un elemento assegnato

```
void inserisci_dopo(r* inizio, int info, int dove)
{r* q, *aux;
aux=inizio;
while(aux && (aux->dato!=dove))
        aux=aux->next;
if(aux)
   {q=(r *)malloc(sizeof(r));
    q->voto=info;
    q->next=aux->next;
    aux->next=q;}
else
    printf("Elemento assente!");}
```



Inserimento nodo prima di un elemento assegnato

1. Scorro la lista (mediante puntatore ausiliario) fino a posizionarmi nel nodo precedente a quello desiderato.
2. Se il punto desiderato è stato trovato
   a) Creo il nuovo elemento
   b) Creo i collegamenti.

   Per non perdere i collegamenti è necessario collegare prima il nuovo elemento al successivo della lista e poi il precedente al nuovo elemento.

   Se il punto desiderato è il primo elemento effettuo un inserimento in testa.

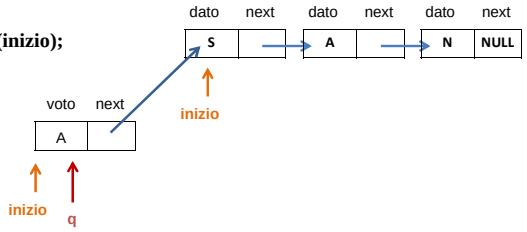
   Altrimenti stampo un messaggio.

Inserimento nodo prima di un elemento assegnato

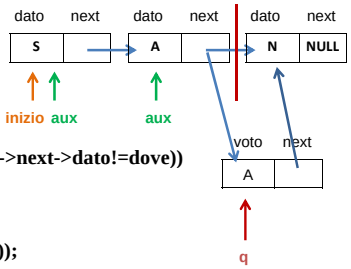
```
r* inserisci_prima(r* inizio, int info, int dove)
{r *q,*aux;
if(inizio->voto==dove)
    {q=(r *)malloc(sizeof(r));
    q->voto=info;     q->next=inizio;     inizio=q;}
else
        {aux=inizio;
        while(aux->next && (aux->next->dato!=dove))
                aux=aux->next;

        if(aux->next)
          {q=(r *)malloc(sizeof(r));
           q->voto=info;          q->next=aux->next;          aux->next=q;}
        else
           printf("Elemento assente!");}
return (inizio);}
```

Inserimento nodo prima di un elemento assegnato

```
if(inizio->dato==dove)
    {q=(r *)malloc(sizeof(r));
    q->voto=info;      q->next=inizio;      inizio=q;}
else
  {…}
return(inizio);
```



Inserimento nodo prima di un elemento assegnato




```
if …
else
 {aux=inizio;
 while(aux->next && (aux->next->dato!=dove))
        aux=aux→next;
 if(aux->next)
    {q=(r *)malloc(sizeof(r));
     q->voto=info;    q->next=aux->next;      aux->next=q;}
  else
    printf("Elemento assente!");}
return (inizio);}
```

Cancellazione di un elemento assegnato

1. Se l'elemento da cancellare è il primo sposto l'inizio e libero l'area di memoria.
2. Altrimenti scorro la lista (mediante puntatore ausiliario) fino a posizionarmi nel nodo precedente a quello desiderato.
3. Se il punto desiderato è stato trovato
   a) Posiziono un puntatore nell'area da cancellare
   b) Creo i collegamenti
   c) Libero l'area di memoria

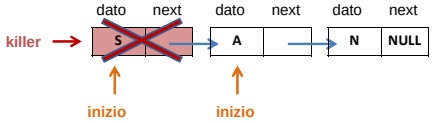
   Altrimenti stampo un messaggio.
4. Restituisco l'inizio.

Cancellazione di un elemento assegnato

```
r* cancella(r* inizio, int dove)
{r *killer,*aux;
if(inizio->dato==dove)
    {killer=inizio;        inizio=inizio->next;      free(killer);}
else
        {killer=inizio;
        while( killer->next && (killer->next->dato!=dove) )
                killer=killer->next;

        if(killer->next)
          {aux=killer->next;       killer->next=aux->next;
           free(aux); }
        else
            printf("Elemento assente!");}
return (inizio);}
```

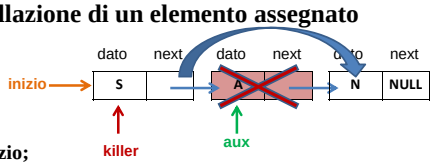
Cancellazione di un elemento assegnato

```
if(inizio->dato==dove)
    {killer=inizio;     inizio=inizio->next;   free(killer);}
else
  {…}
return(inizio);
```



Cancellazione di un elemento assegnato

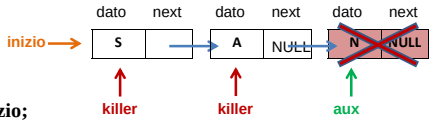
```
if …
else
```




```
        else
        {killer=inizio;
        while(killer->next && (killer->next->dato!=dove))
                killer=killer→next;

        if(killer->next)
          {aux=killer->next;    killer->next=aux->next;
           free(aux); }
        else
            printf("Elemento assente!");}
return (inizio);}
```

Cancellazione di un elemento assegnato

```
if …
else
        else
        {killer=inizio;
        while (killer->next && (killer->next->dato!=dove) )
                killer=killer→next;

        if(killer->next)
           {aux=killer->next;    killer->next=aux->next;
            free(aux);  }
        else
            printf("Elemento assente!");}
return (inizio);}
```



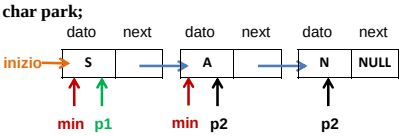
Ordinamento di una lista (per selezione)

Ho necessità di due puntatori p1 e p2 che puntino a due elementi successivi da confrontare e di un terzo puntatore min che punti all'elemento minimo.

1. Posiziono min all'inizio della lista e considero il dato puntato come minimo. Posiziono p1 all'inizio della lista
2. Posiziono p2 nell'elemento successivo a p1. Scorro la lista usando p2.
3. Se trovo un elemento minore di quello puntato da min effettuo lo scambio.
4. Sposto p1 di una posizione e ripeto da 2.

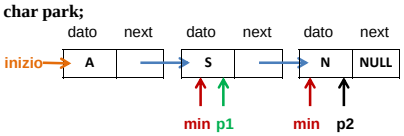
Ordinamento di una lista (per selezione) 1

```
void ordina(r* inizio)
{r *p1,*p2,*min;        char park;
p1=inizio;
while( p1->next)
        {p2=p1->next;
        min=p1;
        while(p2)
                {if(p2->dato < min->dato)
                        min=p2;
                p2=p2->next;}

        if(min!=p1)
           {park=p1->dato;      p1->dato=min->dato;      min->dato=park;}
        p1=p1->next;}
}
```



Ordinamento di una lista (per selezione) 2

```
void ordina(r* inizio)
{r *p1,*p2,*min;        char park;
p1=inizio;
while( p1->next)
        {p2=p1->next;
        min=p1;
        while(p2)
                {if(p2->dato < min->dato)
                        min=p2;
                p2=p2->next;}

        if(min!=p1)
           {park=p1->dato;      p1->dato=min->dato;      min->dato=park;}
        p1=p1->next;}
}
```



Ordinamento di una lista (per selezione) 3

```
void ordina(r* inizio)
{r *p1,*p2,*min;        char park;
p1=inizio;
while( p1->next)
        {p2=p1->next;
        min=p1;
        while(p2)
                {if(p2->dato < min->dato)
                        min=p2;
                p2=p2->next;}

        if(min!=p1)
           {park=p1->dato;      p1->dato=min->dato;      min->dato=park;}
        p1=p1->next;}
}
```