Gestione di liste in C

Strutture dinamiche

Gli array ci permettono di memorizzare un insieme di dati dello stesso tipo

– Deve essere noto staticamente il numero massimo di dati da memorizzare

– C'è uno spreco di memoria ogni qual volta la quantità di dati effettivamente da memorizzare sia inferiore al numero massimo previsto

È utile poter costruire strutture dati capaci di crescere al crescere delle necessità del programma..... e di liberare memoria quando non è più necessaria

Strutture Dinamiche

Il C offre due importanti funzioni (in stdlib.h)

– malloc

  Richiede come parametro la dimensione dell'oggetto creato

  Restituisce un puntatore all'oggetto

– free

  Richiede come parametro il puntatore a un oggetto creato da malloc

  Rende la memoria utilizzata dall'oggetto disponibile per nuovi oggetti

Si usa lo *heap*

Strutture Dinamiche

Vantaggi:

– Si creano solo i contenitori che servono per la particolare istanza del problema
– Aumenta la flessibilità d'uso

Svantaggi:

– L'uso è più complesso
– Si possono creare dangling pointers

Liste

Sono le strutture dinamiche più semplici

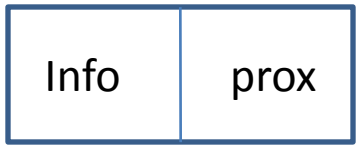
Una lista è composta da un insieme di nodi

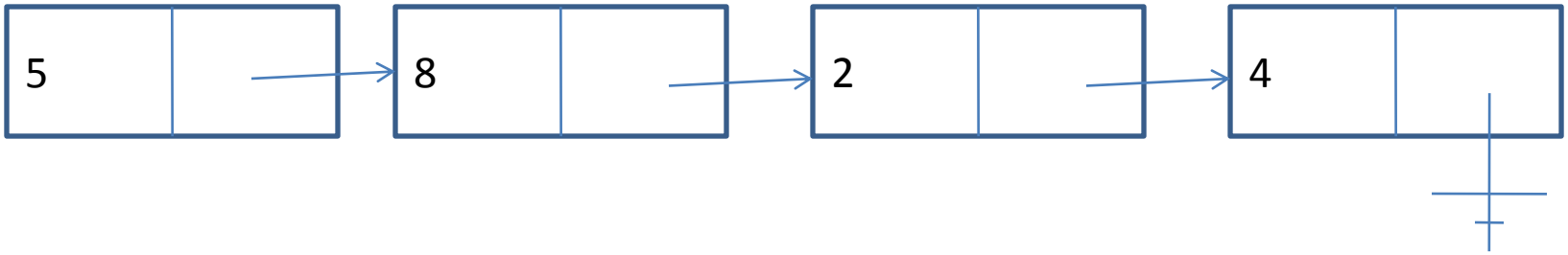
Ciascun nodo contiene:

- Un dato
- Il puntatore al nodo successivo

Nodi e liste

Nodo





Lista contenente 4 elementi

Le liste in C

```
typedef struct Nodo {
        TipoElemento info;
        struct Nodo *prox;
                        } Nodo;

typedef Nodo *Lista;
```

Inizializzazione

```
Lista inizializza() {
   return NULL;
}
```

Esempio

```
Lista lista;
// codice vario
lista = inizializza();
```

Inizializzazione – versione 2

```
void inizializza(Lista *l) {
    *l = NULL;
}
```

Esempio

```
Lista lista;
// codice vario
inizializza(&lista);
```

Controllo lista vuota

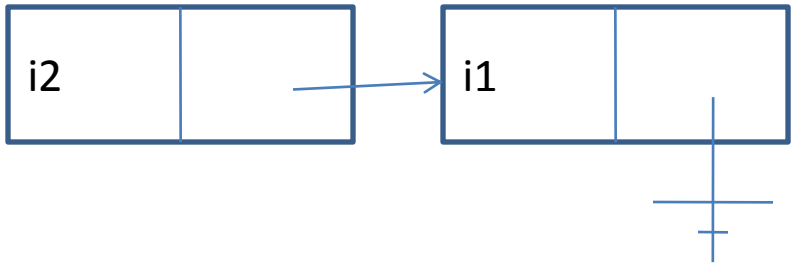
```
boolean listaVuota(Lista l) {
   return l == NULL;
}
```

Esempio

```
Lista lista;
// codice vario
if(listaVuota(lista)) ...
```

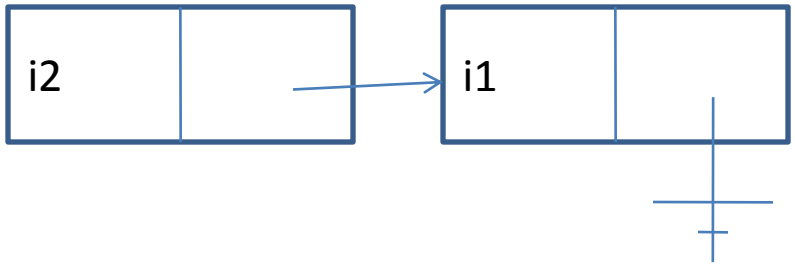
Inserimento in testa

```
Lista inserisciInTesta(Lista l, TipoElemento el){
    Nodo *temp;
    temp = malloc(sizeof(Nodo));
    temp–>info = el;
    temp–>prox = l;
    return temp;
}

Esempio
Lista lista=NULL;
TipoElemento i1, i2;
// codice vario
lista = inserisciInTesta(lista, i1);
lista = inserisciInTesta(lista, i2);
```



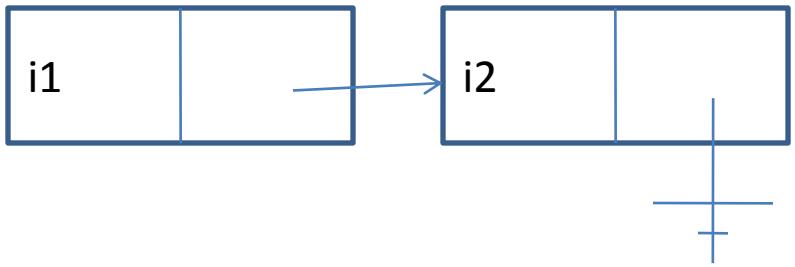
Inserimento in testa – versione 2

```
void inserisciInTesta(Lista *l, TipoElemento el){
    Nodo *temp;
    temp = malloc(sizeof(Nodo));
    temp–>info = el;
    temp–>prox = *l;
    *l=temp;
}

Esempio
Lista lista=NULL;
TipoElemento i1, i2;
// codice vario
inserisciInTesta(&lista, i1);
inserisciInTesta(&lista, i2);
```



Inserimento in coda

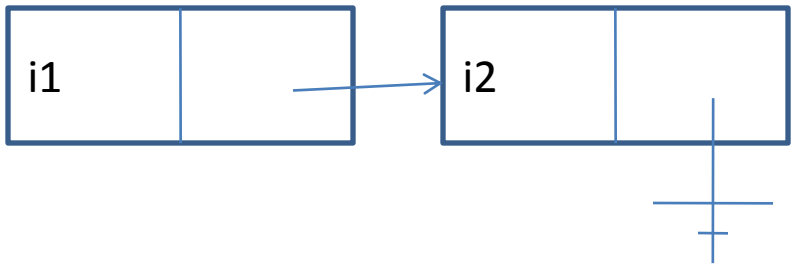
```
Lista inserisciInCoda(Lista l, TipoElemento el) {
        Nodo *temp;
        if (listaVuota(l)) {
                temp = malloc(sizeof(Nodo));
                temp->info = el;
                temp->prox = NULL;
        } else {
                // Inserire ciclo per posizionarsi sull'ultimo elemento con prox==NULL
                temp->prox = malloc(sizeof(Nodo));
                temp->prox->info = el;
                temp->prox->prox = NULL;
                temp=l;
        }
         return temp;
}
```




```
Esempio
Lista lista=NULL;
TipoElemento i1, i2;
// codice vario
lista = inserisciInCoda(lista, i1);
lista = inserisciInCoda(lista, i2);
```

Inserimento in coda – versione 2

```
void inserisciInCoda(Lista *l, TipoElemento el) {
	Nodo *temp;
	if (listaVuota(*l)) {
		temp = malloc(sizeof(Nodo));
		temp->info = el;
		temp->prox = NULL;
		*l = temp;
	} else {
		 // Inserire ciclo per posizionarsi sull'ultimo elemento con prox==NULL
		temp->prox = malloc(sizeof(Nodo));
		temp->prox->info = el;
		temp->prox->prox = NULL;
	}
}
```

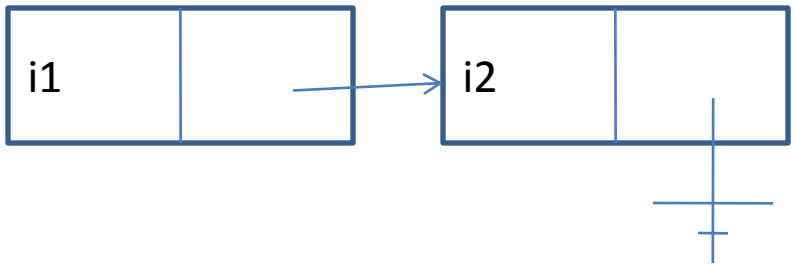


Esempio

```
Lista lista=NULL;
TipoElemento i1, i2;
// codice vario
inserisciInCoda(&lista, i1);
inserisciInCoda(&lista, i2);
```

Inserimento in coda - ricorsiva

```
Lista inserisciInCoda(Lista l, TipoElemento el) {
	Nodo *temp;
	if (listaVuota(l)) {
		temp = malloc(sizeof(Nodo));
		temp–>info = el;
		temp–>prox = NULL;
	} else {
		l->prox = inserisciIncoda(l–>prox, el);
		temp=l;
	}
	return temp;
}
```




```
Esempio
Lista lista=NULL;
TipoElemento i1, i2;
// codice vario
lista = inserisciInCoda(lista, i1);
lista = inserisciInCoda(lista, i2);
```

Cancellazione di un elemento

```
Lista cancella(Lista l, TipoElemento el) {
	Nodo *temp, *puntCorrente, *puntPrecedente;
	if(listaVuota(l)) return l;
		if(l->info==el) {
				temp = l;
				l = l->prox;
				free(temp);
		} else {
				puntPrecedente=NULL;
				puntCorrente=l;
				while (puntCorrente!=NULL && puntCorrente->info!=el) {
				puntPrecedente = puntCorrente;
				puntCorrente=puntCorrente->prox;
		}
	if(puntCorrente!=NULL && puntCorrente->info==el) {
		temp=puntCorrente;
		puntPrecedente->prox = puntCorrente->prox;
		free(temp);
		}
	}
	return l;
}
```

Cancellazione di un elemento – versione 2

```
void cancella(Lista *l, TipoElemento el) {
	Nodo *temp, *puntCorrente, *puntPrecedente;
	if(listaVuota(*l)) return;
		if((*l)->info==el) {
				temp = *l;
				*l = (*l)->prox;
				free(temp);
		} else {
				puntPrecedente=NULL;
				 puntCorrente=l;
				while (puntCorrente!=NULL && puntCorrente->info!=el) {
				puntPrecedente = puntCorrente;
				puntCorrente=puntCorrente->prox;
		}
	if(puntCorrente!=NULL && puntCorrente->info==el) {
		temp=puntCorrente;
		puntPrecedente->prox = puntCorrente->prox;
		free(temp);
		}
	}
}
```

Esercizi

1. Si scriva la funzione cancella per il caso delle liste ordinate (si sfrutti l'ordinamento per semplificare la ricerca)
2. Si scriva una funzione cancellaTutti che cancella tutte le occorrenze di un dato elemento da una lista
3. Si scriva una funzione cerca che restituisce true se un dato elemento è presente nella lista passata come parametro
4. Si riscriva la funzione di cancellazione in forma ricorsiva