10 Liste

Una **struct** (struttura) **autoreferenziale** ha un membro puntatore che punta a una struttura dello stesso tipo, chiamato **link**, e che serve a creare una *catena* (lista) di nodi collegati tra loro.

```
1  struct nodo {
2    int dato;
3    struct nodo *nextPtr;
4  } n1, n2, n3;
```

Listing 17: Esempio di una struct

Definizione 10.1 (Struttura dinamica)**.** *È una struttura dati che può **variare** la sua dimensione a tempo di esecuzione, aumentando e diminuendo. Alcuni esempi sono proprio le **liste**, oltre che le pile, le code e gli alberi binari.*

Note 10.0.1. *L'ultimo nodo di una lista conterrà nel link **null**.*

10.1 Confronto tra liste e array

Liste	**Array**
Contiene sequenze di dati	Contiene sequenze di dati
Create dinamicamente a tempo di esecuzione e la dimensione non può essere prevista a tempo di compilazione	Creati staticamente a tempo di esecuzione e la dimensione deve essere calcolabile a tempo di compilazione
La dimensione è variabile	La dimensione è costante e tutti gli elementi sono allocati a tempo di definizione
Diventa piena solo quando termina la memoria disponibile nell'heap	Diventa pieno quando sono pieni tutti gli elementi
Non sono memorizzate in celle contigue	Sono in celle contigue
Possono essere manipolate senza spostare elementi	Per manipolarli bisogna spostare gli elementi

10.2 Operazioni sulle liste

10.2.1 Insert

Viene codificata come segue:

```
1  void insert(NodoPtr *lPtr, int val) {
2    // Alloco nuovo nodo
3    NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4    if (nuovoPtr != NULL) {
5      // Inizializzo nodo
6      nuovoPtr->dato = val;
7      nuovoPtr->prossimoPtr = NULL;
8      NodoPtr precedentePtr = NULL;
9      NodoPtr correntePtr = *lPtr;
10     while (correntePtr != NULL && val > correntePtr->dato) {
11       precedentePtr = correntePtr;
12       correntePtr = correntePtr->prossimoPtr;
13     }
14     if (precedentePtr == NULL) {
15       // Inserimento all'inizio della lista
16       nuovoPtr->prossimoPtr = *lPtr;
17       *lPtr = nuovoPtr;
18     }
19     else {
20       // Inserimento tra due nodi
```

```
21          precedentePtr->prossimoPtr = nuovoPtr;
22          nuovoPtr->prossimoPtr = correntePtr;
23        }
24      }
25      else {
26        puts("Memoria esaurita");
27      }
28    }
```

Listing 18: Inserimento in una lista

10.2.2 Delete

Viene codificata come segue:

```
1  void delete(NodoPtr *lPtr, int val) {
2    if (*lPtr != NULL) {
3      if (val == (*lPtr)->dato) {
4        NodoPtr tempPtr = *lPtr;
5        *lPtr = (*lPtr)->prossimoPtr;
6        free(tempPtr);
7      }
8      else {
9        NodoPtr precedentePtr = *lPtr;
10       NodoPtr correntePtr = (*lPtr)->prossimoPtr;
11       while (correntePtr != NULL && correntePtr->dato != val) {
12         precedentePtr = correntePtr;
13         correntePtr = correntePtr->prossimoPtr;
14       }
15       if (correntePtr != NULL) {
16         NodoPtr tempPtr = correntePtr;
17         precedentePtr->prossimoPtr = correntePtr->prossimoPtr;
18         free(tempPtr);
19       }
20     }
21   }
22 }
```

Listing 19: Cancellazione in una lista

10.2.3 Verifica se è vuota

```
1  int is_empty(NodoPtr lPtr) {
2    return lPtr == NULL;
3  }
```

Listing 20: Verificare se la lista è vuota

10.3 Liste particolari

10.3.1 Pile

Definizione 10.2. *Una **pila** è una lista in cui inserimenti e cancellazioni possono essere fatte solo sulla testa della lista (politica **LIFO**).*

Le operazioni che possono essere eseguite sulle pile sono:

- **Push**: inserimento di un nuovo nodo in testa

```
1      void push(NodoPtr *topPtr, int val) {
2        // alloco nuovo nodo
3        NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4        if (nuovoPtr != NULL) { // Spazio disponibile
5          // inizializzo nodo
6          nuovoPtr->dato = val;
7          nuovoPtr->prossimoPtr = *topPtr;
8          *topPtr = nuovoPtr;
9        }
```

```
10        else {
11          puts("Memoria esaurita");
12        }
13      }
```

- **Pop**: cancellazione di un elemento in testa

```
1       int pop(NodoPtr *topPtr) {
2         int val = (*topPtr)->dato;
3         NodoPtr tempPtr = *topPtr;
4         *topPtr = (*topPtr)->prossimoPtr;
5         free(tempPtr);
6         return val;
7       }
```

- **is_empty**, **stampa_pila**

10.3.2 Code

Definizione 10.3. *Una **pila** è una lista in cui inserimenti e cancellazioni possono essere fatte solo alla fine della lista (politica **FIFO**).*

Le operazioni che possono essere eseguite sulle pile sono:

- **Enqueue**: inserimento di un nuovo nodo alla fine della coda

```
1       void enqueue(NodoPtr *testaPtr, NodoPtr * codaPtr, int val) {
2         // alloco nuovo nodo
3         NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4         if (nuovoPtr != NULL) { // Spazio disponibile
5           // inizializzo nodo
6           nuovoPtr->dato = val;
7           nuovoPtr->prossimoPtr = NULL;
8           if (is_empty(*testaPtr)) {
9             *testaPtr = nuovoPtr;
10          }
11          else {
12            (*codaPtr)->prossimoPtr = nuovoPtr;
13            *codaPtr = nuovoPtr;
14          }
15        }
16        else {
17          puts("Memoria esaurita");
18        }
19      }
```

- **Dequeue**: cancellazione di un elemento in testa

```
1       int dequeue(NodoPtr *testaPtr, NodoPtr * codaPtr) {
2         int val = (*testaPtr)->dato;
3         NodoPtr tempPtr = *testaPtr;
4         *testaPtr = (*testaPtr)->prossimoPtr;
5         if (*testaPtr == NULL) {
6           *codaPtr = NULL;
7         }
8         free(tempPtr);
9         return val;
10      }
```

- **is_empty**, **stampa_coda**