UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento
di Fisica
e Astronomia
"Ettore Majorana"

L-30 CLASSE DELLE LAUREE IN SCIENZE E TECNOLOGIE FISICHE
CORSO DI LAUREA IN FISICA
PROGRAMMAZIONE AD OGGETTI E BIG DATA

SANDRO FIORETTO

REALIZZAZIONE IN C DELL'OGGETTO LISTA

SOMMARIO

L'OBIETTIVO DELLA RELAZIONE È PRESENTARE L'OGGETTO "LISTA", REALIZZATO IN C IN DUE VERSIONI: UNA CLASSICA IN CUI VIENE CREATO UN NUOVO ELEMENTO AD OGNI VALORE INSERITO E UNA PIÙ VELOCE PERCHÉ UTILIZZA, ASSIEME A TUTTE LE ALTRE LISTE DELLO STESSO TIPO, UN UNICO SPAZIO IN MEMORIA PREALLOCATO.

ANNO ACCADEMICO 2022/2023 - 3° ANNO

Indice

1 Cos'è la lista in informatica

La lista è una struttura dati dinamica. A differenza dell'array può facilmente cambiare numero di elementi e i suoi elementi non sono necessariamente contigui in memoria.
La complessità delle seguenti funzioni della lista non dipende dalla sua dimensione:

- inserimento e rimozione in testa alla lista
- inserimento in coda alla lista

ma vi dipende nel caso di:

- rimozione in coda alla lista
- inserimento e rimozione all'interno della lista
- ricerca di elementi interni alla lista

È perciò particolarmente utile quando il numero di elementi non è noto a priori e non si devono effettuare molte operazioni con elementi interni alla lista.

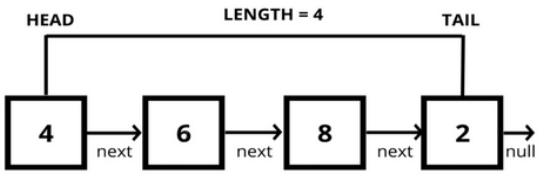


Figura 1: Schema della lista. Gli oggetti non sono contigui in memoria ma ciascuno contiene l'informazione su come raggiungere il successivo.

2 Tipi di lista

Le liste di tutti i tipi vengono istanziate attraverso la funzione $malloc\_list$, internamente tuttavia sono fornite 6 classi derivate, 3 per ciascun tipo. I due tipi di lista sono $list\_dynamic$ e $list\_table$.

La $list\_dynamic$ è la versione più classica della lista: la lista occupa una quantità di memoria proporzionale al numero di elementi contenuti. Ha il vantaggio rispetto alla versione successiva di occupare solamente la memoria necessaria per contenere i suoi elementi, tuttavia l'inserimento di ciascun elemento richiede un'allocazione in memoria, il ché rende la $list\_dynamic$ più lenta degli array o della sua controparte $list\_table$.

La $list\_table$ è una versione in cui viene preallocata una quantità di memoria arbitraria (detta $table$ o tabella) all'istanziamento della prima lista e nella quale vengono salvati gli elementi di tutte le liste della stessa natura. Ciascuna table infatti deve avere elementi della stessa dimensione quindi potrà essere condivisa solo da liste che che contengono lo stesso tipo di dato (sarà istanziata una table per le liste che contengono interi, una per i float etc.).
Il vantaggio di questa versione rispetto alla precedente è che lavorare con la table è più veloce che lavorare con puntatori; rispetto agli array è che le liste contenute nella table possono avere numero di elementi variabile.
Lo svantaggio è che quando la memoria preallocata si riempie la table deve essere ricopiata in una zona di memoria più grande, il ché rallenta temporaneamente il programma. Per decidere come ingrandire la table sono forniti due tipi di resize:

- $tpe\_resize\_default$: è il tipo di resize della table se non è stato specificato diversamente. La table si espande automaticamente quando si cerca di inserire elementi oltre la sua capienza.
- $type\_resize\_manual$: le funzioni di inserimento tornano errore se la tabella è già piena e questa va espansa manualmente attraverso la funzione $resize\_table$.

Dimensione e tipo di resize della table possono essere specificati in fase di allocazione della lista attraverso $malloc\_list\_specify\_table$.
Si noti che per ragioni di compatibilità con liste preesistenti, $malloc\_list\_specify\_table$ imposta il tipo di resize e la dimensione della tabella solo se questa non esiste ancora. In caso esistesse, il tipo di resize può essere modificato con $change\_resize\_table$, la dimensione con $resize\_table$.

Le 3 classi derivate per ciascun tipo sono:

- lista di elementi singoli
- lista di array
- lista generica

infatti queste tre categorie differiscono nel codice tra di loro ma racchiudono tutti i tipi di lista istanziabili (int, float, array_float, generic, ...).

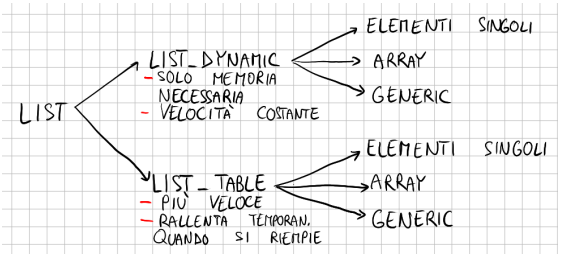


Figura 2: Schema dei tipi di lista. Tutte le liste sono comunque istanziate con la funzione $malloc\_list$ mentre il differenziamento tra tipi è interno.

2.1 Tipi di dati contenuti

I tipi che l'oggetto può contenere, stabiliti all'istanziamento di ciascuna lista, sono "INT", "LONG INT", "LONG LONG INT", "UNSIGNED", "LONG UNSIGNED", "LONG LONG UNSIGNED", "FLOAT", "DOUBLE", "CHAR", "SIGNED CHAR", "PVOID", loro array e il tipo "GENERIC".
"GENERIC" è un tipo di dato con dimensione variabile individuato da un puntatore all'indirizzo di memoria in cui è contenuto e un intero unsigned in cui è salvata la sua dimensione, ottenibile ad esempio con la funzione standard di C $sizeof$.

Lavorare con array di dati permette di non dover traversare la lista un elemento alla volta, ma N alla volta, con N numero di elementi di ciascun array. All'istanziamento della lista, è fornita la possibilità di scegliere il numero di valori contenuti in ciascuno dei suoi elementi. Si noti che questo numero è fissato all'istanziamento della lista e deve essere lo stesso per tutti gli array in essa contenuti.

È importante sottolineare che quando un elemento viene inserito all'interno della lista viene creata una sua copia locale. Ciò permette di mantenere invariato l'elemento salvato nella lista pur modificando la variabile attraverso cui è stato inserito (presa ad esempio in input da $insert\_first$).

Poiché C non permette l'overloading delle funzioni, sarebbero richieste tante funzioni di inserimento ed estrazione quanti sono i tipi contenuti nella lista. Piuttosto si è scelto di utilizzare la union all_type, che contiene tutti i tipi di dato conservabili nell'oggetto lista.
Le funzioni di inserimento e di estrazione prendono in input elementi di tipo all_type, per cui è importante castare ad all_type tali variabili.
Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

Ad esempio nel caso di inserimento ed estrazioni di valori di tipo float, si avranno le seguenti funzioni:

```
insert_first(plista, (all_type) f, ...)
extract_first(plista, (all_type)((pvoid)&f), ...)
```

3 Manuale d'uso della lista

Di seguito sono riportate le funzioni membro fornite e come si utilizzano.

3.1 Istanziamento della lista

Esempi di allocazione di liste:

- list_dynamic che contiene int singoli:

```
plist=malloc_list(type_list_dynamic, "INT", 1);
```

- list_dynamic che contiene array di 5 float:

```
plist=malloc_list(type_list_dynamic, "FLOAT", 5);
```

- list_table che contiene generic:

```
plist=malloc_list(type_list_table, "GENERIC", ...);
```

- list_table, specificando le caratteristiche della tabella:

```
plist=malloc_list_specify_table("INT",
                                3,
                                type_resize_manual,
                                100);
```

* ... vuol dire che gli argomenti successivi non servono per questo tipo di lista

a.
```
pvoid malloc_list(type_list type_list,
                  pchar type_string,
                  unsi dim_array);
```

crea nuove liste e ne restituisce l'indirizzo come void*. Può essere usata per creare **liste di tutti i tipi**, dipendentemente dai parametri inseriti in input. Gli argomenti sono rispettivamente:

- type_list: tipo di lista da istanziare, da scegliere tra $type\_list\_dynamic$ e $type\_list\_table$, con le differenze riportate in Sezione 2;
- type_string: è una stringa che corrisponde al tipo del dato da contenere. Deve essere una tra: "INT", "LONG INT", "LONG LONG INT", "UNSIGNED", "LONG UNSIGNED", "LONG LONG UNSIGNED", "FLOAT", "DOUBLE", "CHAR", "SIGNED CHAR", "PVOID" "GENERIC";
- dim_array: numero di valore che contiene ciascun elemento della lista. Si inserisca 1 per liste con valori singoli e un numero maggiore di uno per liste che contengono array.
  Si noti che la dimensione scelta in fase di allocazione deve essere la stessa per tutti gli array inseriti successivamente nella lista;
- return: puntatore alla nuova lista, NULL altrimenti.

b. `pvoid malloc_list_specify_table(pchar type_string, unsi dim_array, type_resize type_resize, unsi dim_table);`

che alloca **esclusivamente** liste di tipo ***list_table*** e permette di specificare, nel caso in cui la tabella non fosse già esistente (per evitare errori con altre liste che vi appartenessero):

- type_resize: tipo di resize da impostare per la tabella. Va scelto tra $type\_resize_default$ e $type\_resize\_manual$, con le differenze riportate in Sezione 2;
- dim_table: numero massimo di elementi delle liste contenute nella tabella;
- return: puntatore alla nuova lista, NULL altrimenti.

Se si vuole cambiare il tipo di resize di una tabella già esistente si può usare $change\_resize\_table$, mentre per cambiarne la dimensione $resize\_table$ (si rimanda a Sezione 3.2 per maggiori informazioni).

3.2 Modifica della tabella

Esempi di modifiche della tabella:

- cambio il tipo di resize:

```
change_resize(plist, type_resize_manual);
```

- ridimensiono la tabella se c'è abbastanza spazio:

```
unsigned* n_occupied;
get_info_table(plist,
               NULL, /* non ci interessa al momento */
               &n_occupied);
resize_table(plist, n_occupied + 10);
```

La precedente funzione $malloc\_list\_specify\_table$ specifica solo le caratteristiche di tabelle non ancora create. Se ho precedentemente istanziato una lista con $malloc\_list$ e adesso voglio cambiare le caratteristiche della tabella in cui è contenuta, posso usare:

a.
```
int change_resize_table(pvoid plist,
                        type_resize type_resize);
```

cambia il tipo di resize della tabella che contiene plist.

- plist: lista contenuta nella tabella che si vuole modificare;
- type_resize: tipo di resize da impostare per la tabella che contiene plist. Va scelto tra $type\_resize\_default$ e $type\_resize\_manual$, con le differenze riportate in Sezione 2;
- return: 1 se modificato correttamente, 0 altrimenti.

b.
```
int resize_table(pvoid plist, unsi n_entries);
```

ridimensiona la tabella se possibile.

- plist: lista contenuta nella tabella che si vuole modificare;
- n_entries: numero di elementi complessivi della tabella, dopo che è stata ridimensionata;
- return: 1 se ridimensionata correttamente, 0 altrimenti, ad esempio se n_entries è minore del numero di elementi delle liste contenute.

c.
```
int get_info_table(pvoid plist,
                   punsi pn_entries,
                   punsi pn_occupied);
```

fornisce delle informazioni sulla tabella che contiene plist. Può essere usata ad esempio per sapere quanto la tabella può essere rimpicciolita.

- plist: lista contenuta nella tabella di cui si vogliono conoscere le informazioni;
- pn_entries: indirizzo in cui scrivere il numero di elementi complessivi della tabella;
- pn_occupied: indirizzo in cui scrivere il numero di elementi occupati della tabella;
- return: 1 se tutto va bene, 0 altrimenti

3.3 Inserimento in lista

Esempi di inserimento in lista:

- Inserimento in lista float:

```
float f=3.2;
insert_first(plist, (all_type)f,...);
```

- Inserimento in seconda posizione in una lista array di int:

```
int array[5]={1,2,3,4,5};
insert_first(plist,
             (all_type)((pvoid)array),
             0, /* la size non importa
                   per liste di array */
             2);
```

- Inserimento in lista generic:

```
insert_first(plist,
             (all_type)((pvoid)&var),
             sizeof(var));
```

* ... vuol dire che gli argomenti successivi non servono per questo tipo di lista

Le funzioni di inserimento prendono in input variabili di tipo all_type, che è una union che contiene tutti i tipi di dato contenibili.
È importante castare le variabili ad all_type. Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

a. `int insert_first(pvoid plist, all_type value, unsi size);`

   inserisce un elemento in cima alla lista.

   - plist: indirizzo della lista al cui inizio inserire l'elemento
   - value: valore dell'elemento da inserire, da castare ad all_type
   - size: deve essere rispettivamente:
     - dato "GENERIC": dimensione del dato da inserire
     - altri: non ha importanza
   - return: 1 se tutto va bene, 0 altrimenti.

b. `int insert_last(pvoid plist, all_type value, unsi size);`

inserisce un elemento in coda alla lista.

- plist: indirizzo della lista al cui termine inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad all_type
- size: deve essere rispettivamente:
  - dato "GENERIC": dimensione del dato da inserire
  - altri: non ha importanza
- return: 1 se tutto va bene, 0 altrimenti.

c. `int insert_nth(pvoid plist, all_type value, unsi size, unsi n);`

inserisce un elemento all'n-esima posizione della lista. Si noti che le posizioni sono contate a partire da 1, n=1 vuol dire inserire l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.

- plist: indirizzo della lista al cui termine inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad all_type
- size: deve essere rispettivamente:
  - dato "GENERIC": dimensione del dato da inserire
  - altri: non ha importanza
- n: indice della lista in cui inserire l'elemento
- return: 1 se tutto va bene, 0 altrimenti.

3.4 Estrazione dalla lista

Esempi di estrazione dalla lista:

- Estrazione da lista float:

```
float* pf;
extract_first(plist, (all_type)((pvoid)pf),...);
```

- Estrazione del secondo elemento da una lista array di int:

```
int* parray;
unsi* psize;
extract_first(plist,
              (all_type)((pvoid)&parray),
              psize,
              2);
```

- Estrazione da lista generic:

```
void* pvar;
unsi  size;
extract_first(plist,
              (all_type)(&pvar),
              &size);
```

  dove in questo caso dobbiamo fornire un void* perché non sappiamo a priori la dimensione dell'elemento estratto quindi ci pensa il programma ad allocare la memoria necessaria e salvarla in pvar.

* ... vuol dire che gli argomenti successivi non servono per questo tipo di lista

Le funzioni di estrazione prendono in input variabili di tipo all_type, che è una union che contiene tutti i tipi di dato contenibili.
È importante castare le variabili ad all_type. Si noti che:

- la union non contiene tutti i tipi di puntatore ma solo void*, per cui quando vogliamo fornire alla funzione un puntatore dobbiamo prima castarlo a void* e poi ad all_type;
- castare una variabile a union dà warning quando si compila con –pedantic; tuttavia si è scelto di mantenere questa scelta per la sua facilità di utilizzo.

a. `int extract_first(pvoid plist, all_type pvalue, punsi psize);`

estrae un elemento dalla cima della lista.

- plist: indirizzo della lista al cui inizio inserire l'elemento
- pvalue: indirizzo in cui scrivere il valore dell'elemento estratto, da castare a pvoid se è un puntatore di tipo diverso e ad all_type
- psize: nel caso di dato "GENERIC", indirizzo in cui sarà scritta la size dell'elemento estratto
- return: 1 se tutto va bene, 0 altrimenti.

b. `int extract_last(pvoid plist, all_type pvalue, punsi psize);`

estrae l'ultimo elemento della lista.

- plist: indirizzo della cui coda estrarre l'elemento
- pvalue: indirizzo in cui scrivere il valore dell'elemento estratto, da castare a pvoid se è un puntatore di tipo diverso e ad all_type
- psize: nel caso di dato "GENERIC", indirizzo in cui sarà scritta la size dell'elemento estratto
- return: 1 se tutto va bene, 0 altrimenti.

c. `int extract_nth(pvoid plist, all_type pvalue, punsi psize, unsi n);`

estrae l'elemento all'n-esima posizione della lista. Si noti che le posizioni sono contate a partire da 1, n=1 vuol dire estrarre l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.

- plist: indirizzo della cui coda estrarre l'elemento
- pvalue: indirizzo in cui scrivere il valore dell'elemento estratto, da castare a pvoid se è un puntatore di tipo diverso e ad all_type
- psize: nel caso di dato "GENERIC", indirizzo in cui sarà scritta la size dell'elemento estratto
- n: indice nella lista dell'elemento da estrarre
- return: 1 se tutto va bene, 0 altrimenti.

3.5 Altre funzioni utili

a.
```
int search_first(pvoid plist,
                 all_type value_searched, unsi size_searched,
                 all_type pvalue_found,   punsi psize_found,
                 pcustom_compare pinput_compare);
```

trova la prima occorrenza dell'elemento cercato e la scrive in pvalue_found, in particolare:

- value_searched: valore con cui confrontare l'elemento da cercare, da castare ad all_type
- size_searched: dimensione dell'elemento da cercare, utile per cercare elementi di tipo generic
- pvalue_found: indirizzo in cui scrivere il valore dell'elemento trovato
- psize_found: indirizzo in cui scrivere la dimensione dell'elemento trovato (nel caso di liste con tipo di dato generic)
- pinput_compare: funzione del tipo

  ```
  int pcustom_compare(all_type value1, unsi size1,
                      all_type value2, unsi size2);
  ```

  con cui ciascun elemento della lista viene confrontato (come value1) con l'elemento da cercare (come value2). La funzione deve tornare 0 quando i due elementi sono uguali, e in quel caso l'elemento trovato viene scritto in pvalue_found e psize_found.
- return: 1 se lo ha trovato, 0 altrimenti

b.
```
int print_list(pvoid plist, pcustom_print pinput_print);
```

stampa informazioni sulla lista nel formato:

```
type_list: ...
type_data: ...
Numero di elementi: ...
```

e i primi 5 elementi della lista se è fornita pinput_print. In particolare:

- plist: lista da stampare
- pinput_print: funzione del tipo

  ```
  int pcustom_print(all_type value, unsi size);
  ```

  con cui stampare un elemento della lista. Deve tornare 1 se tutto va bene, 0 altrimenti.
- return: 1 se tutto va bene, 0 altrimenti

4 Manuale tecnico

4.1 Tipi di lista

Abbiamo visto in Sezione 2 che l'oggetto consiste di 6 classi derivate. In realtà le classi sono molto più di queste, e in particolare esiste una classe derivata per ogni tipo di dato contenibile.

La creazione di queste classi è realizzata in modo programmatico da uno script Bash a partire dalle classi: $list\_dynamic\_BASETYPE$, $list\_dynamic\_array\_BASETYPE$, $list\_dynamic\_generic$, $list\_table\_BASETYPE$, $list\_table\_array\_BASETYPE$, $list\_table\_generic$.
Lo script ricopia le classi con $BASETYPE$ tante volte quanti sono i tipi base da contenere (char, int, float, ...) e sostituisce nel codice:

- il termine BASETYPE con il nome del tipo base scritto con gli underscore (ad es. long_int, signed_char)
- il termine MEMBERTYPE con il nome del rispettivo membro della union (ad es. c per i char, f per i float, ...)
- il termine MEMBERTYPE_SPACES con il nome del tipo base scritto con gli spazi (ad es. long int, signed char), che serve ad esempio quando calcolo sizeof(long int)

Per questioni di velocità si è scelto di evitare chiamate di funzioni a cascata, ma per ciascuna funzione esiste un array di puntatori a funzioni che collega il tipo di lista alla rispettiva funzione nella classe derivata.

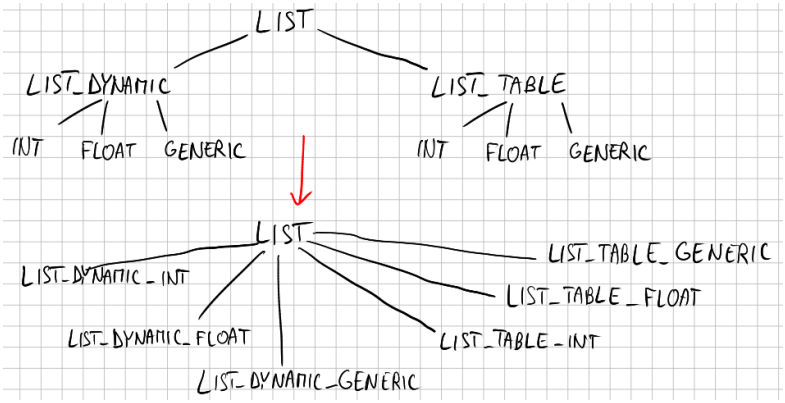


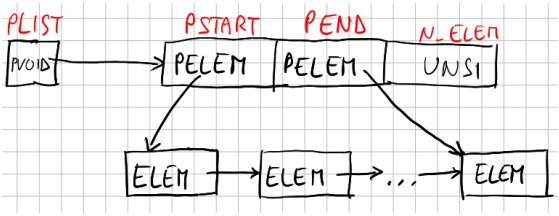
Figura 3: Schema di chiamata delle funzioni. Si è scelto il metodo in basso per risparmiare una chiamata a funzione. Ciò implica avere array di funzioni più lunghi ma sempre dell'ordine della decina di funzioni, il ché non dovrebbe essere un problema nella maggior parte dei casi.

4.2 Lista dinamica

Illustrerò di seguito la struttura della lista dinamica e come sono programmate le funzioni base. Siccome le classi derivate sono molto simili tra di loro, prenderò come esempio la classe $list\_dynamic\_BASETYPE$, sottolineando le differenze con le altre classi quando necessario.

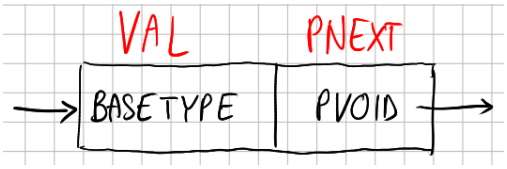
4.2.1 Struttura

Il funzionamento della lista dinamica si basa su una struttura centrale che contiene informazioni su come raggiungere il primo e l'ultimo elemento della lista e il numero di elementi contenuti.
Ciascun elemento contiene poi l'informazione che permette di raggiungere il successivo.



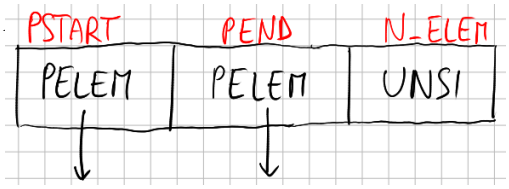
La struct $elem\_BASETYPE$ (o $elem\_generic$ in modo molto simile):

```
typedef struct _elem_BASETYPE {
  BASETYPE val;
  struct _elem_BASETYPE *pnext;
} elem_BASETYPE;
```



sono i "mattoncini" di cui è fatta la lista. Ciascuno contiene un valore di tipo BASETYPE e il puntatore all'elemento successivo.
La struct $list\_dynamic\_BASETYPE$:

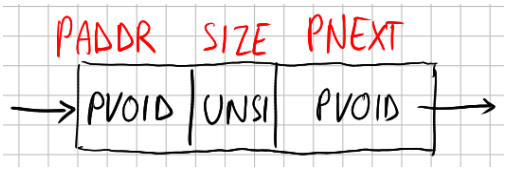
```
typedef struct _list_dynamic_BASETYPE
  pelem_BASETYPE pstart;
  pelem_BASETYPE pend;
  unsi n_elem;
} list_dynamic_BASETYPE;
```



contiene le informazioni sulla lista vera e propria, cioè l'indirizzo del primo e dell'ultimo elemento e il numero di elementi contenuti.

Nel caso di $elem\_generic$ cambia leggermente la struct:

```
typedef struct _elem_generic {
  pvoid paddr;
  unsi size;
  struct _elem_generic *pnext;
} elem_generic;
```



ma il principio è del tutto analogo.

Nel caso di $elem\_array\_BASETYPE$ l'implementazione è più complicata perché gli elementi hanno size variabile in base alla dimensione degli array contenuti, quindi non possiamo usare le struct di C per essi.
La struct che contiene le informazioni sulla lista è simile, ma contiene anche la dimensione dell'array.
La memoria occupata da ciascun elemento è calcolata manualmente (invece di sizeof(elem)) e pari a $size\_array * sizeof(BASETYPE) + sizeof(pvoid)$.
$elem\_BASETYPE->pnext$ è sostituito invece dalla macro GET_PNEXT:

```
#define GET_PNEXT(pvalue) \
    (*((ppvoid)(((pBASETYPE) pvalue )+size_array )))
```

che partendo dall'inizio dell'elemento ($pvalue$) si sposta di $sizeof(BASETYPE)*size\_array$ e legge il pezzo di memoria successivo come void*.

4.2.2 Funzioni di inserimento ed estrazione

Le funzioni di inserimento ed estrazione sono simili per i 3 tipi singolo, array e generic. Cambia sostanzialmente come scrivere elementi della lista e come restituire gli elementi all'utente.

Le funzioni di inserimento prendono in input:

- tipi base: un valore BASETYPE (int, float, char, ...) e lo copiano direttamente con l'assegnamento di C: pnew_elem→val = value_input
- tipo generic: un puntatore pvoid che contiene l'indirizzo dell'elemento da copiare e un unsigned corrispondente alla size dell'elemento.
  Viene allocata la memoria per contenere il valore (pnew_elem→paddr = malloc(size)) e poi copiato il valore con memcpy(pnew_elem→paddr, pvalue_input, size)
- tipo array: un puntatore pvoid che contiene l'indirizzo di partenza dell'array.
  A differenza del caso precedente, l'elemento della lista è già della grandezza sufficiente a contenere l'array, quindi basta ricopiarlo con memcpy(pnew_elem, pvalue_input, sizeof_array), dove la sizeof_array è salvata nella struct della lista.

Le funzioni di estrazione prendono in input:

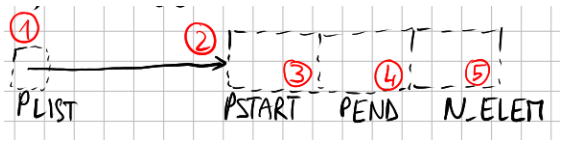
- tipi base: un puntatore pBASETYPE (ad esempio pint, pfloat, ...) e restituiscono il valore direttamente con l'assegnamento di C: *pvalue_input = pelem_to_extract→val
- tipo generic: un puntatore ppvoid in cui scrivere l'indirizzo del valore estratto (*ppvalue_input = pelem_to_extract→paddr) e un unsigned* in cui scrivere la size dell'elemento estratto (*psize = pelem_to_extract→size).
  Un metodo alternativo sarebbe prendere direttamente un pvoid e copiare il valore da estrarre all'indirizzo contenuto nel pvoid. Questo metodo prevede però che l'utente allochi la memoria necessaria cioè che sappia a priori quanto è la size dell'elemento da estrarre, il ché non è detto dato che la lista può contenere elementi di size arbitraria
- tipo array: un puntatore ppvoid in cui alloco la memoria per l'array (*ppvalue_input = malloc(sizeof(array)) e copio l'array con memcpy (memcpy(*ppvalue_input, pelem→val, sizeof(array))).
  A differenza del caso generic non posso scrivere direttamente l'indirizzo dell'array in ppvalue_input perché l'array è contenuto nell'elemento della lista e nella stessa zona di memoria del puntatore all'elemento successivo. Piuttosto devo invece ricopiare l'array e liberare l'intero elemento.

Come detto in precedenza comunque, tutti questi possibili tipi sono contenuti nella union all_type, in modo che sia sufficiente che le funzioni prendano in input valori di tipo all_type e che l'utente casti la variabile (int, pfloat, ppvoid, ...) a all_type.

4.2.3 malloc_list

```
pvoid malloc_list_dynamic_BASETYPE(unsi dim_array);
```

- dim_array: numero di valori contenuto in ciascun elemento della lista
- return: l'indirizzo della nuova lista istanziata, NULL se non è andato a buon fine



1 creo pvoid plist

2 plist = malloc(sizeof(list_dynamic_BASETYPE))

3 pstart = NULL

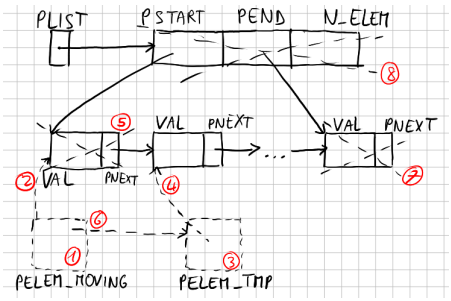
4 pend = NULL

5 n_elem = 0

4.2.4 free_list

```
void free_list_dynamic_BASETYPE(pvoid plist);
```

- plist: indirizzo della lista da liberare
- return: niente



1 creo pelem_moving

2 pelem_moving = plist→pstart

3 creo pelem_tmp

4 pelem_tmp = pelem_moving→pnext

5 free(pelem_moving)

6 pelem_moving = pelem_tmp e vado avanti eliminando gli altri elementi finché pelem_moving→pnext == NULL

7 libero l'ultimo elemento

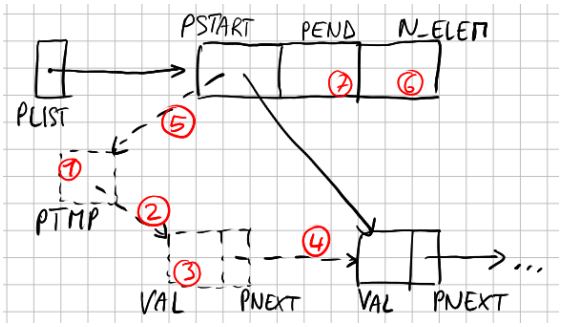
8 libero la struttura della lista

NB: nel caso di list_dynamic_generic devo anche liberare il valore contenuto (free(pelem_moving→paddr) prima di free(pelem_moving)

4.2.5 insert_first

```
int insert_first_dynamic_BASETYPE(pvoid plist,
                                  all_type value,
                                  unsi size);
```

- plist: lista alla cui cima inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco temporaneo ptmp

2 ptmp = malloc(sizeof(elem))

3 scrivo il valore preso in input in ptmp→val

4 ptmp→pnext = plist→pstart

5 plist→pstart = ptmp

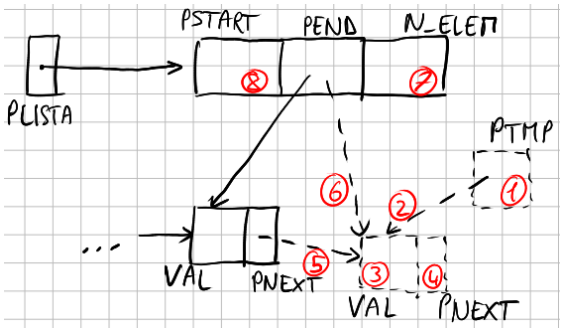
6 n_elem++

7 if(n_elem == 1) plist→pfine = ptmp

4.2.6 insert_last

```
int insert_last_dynamic_BASETYPE(pvoid plist,
                                 all_type value,
                                 unsi size);
```

- plist: lista alla cui coda inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco temporaneo ptmp

2 ptmp = malloc(sizeof(elem))

3 scrivo il valore preso in input in ptmp→val

4 ptmp→pnext = NULL

5 plist→pend→pnext = ptmp

6 plist→pend = ptmp

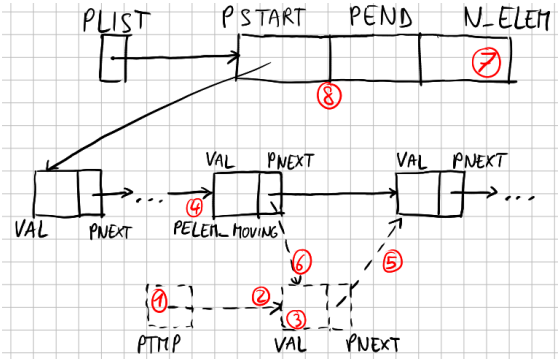
7 n_elem++

8 if(n_elem == 1) plist→pstart = ptmp

4.2.7 insert_nth

```
int insert_nth_dynamic_BASETYPE(pvoid plist,
                                all_type value,
                                unsi size,
                                unsi n);
```

- plist: lista in cui inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- indice a cui inserire l'elemento. Le posizioni sono contate a partire da 1: n=1 vuol dire inserire l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco temporaneo ptmp

2 ptmp = malloc(sizeof(elem))

3 scrivo il valore preso in input in ptmp→val

4 raggiungo l'(n-1)-esimo elemento pelem_moving con un for

5 ptmp→pnext = pelem_moving→pnext

6 pelem_moving→pnext = ptmp

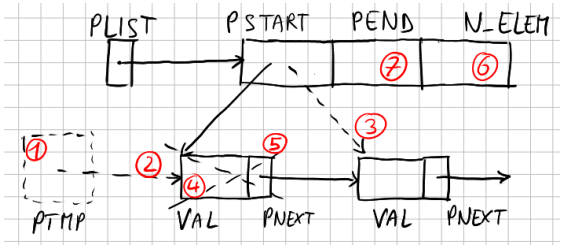
7 n_elem++

8 faccio check sulla lunghezza della lista per sapere se aggiornare pstart o pend

4.2.8 extract_first

```
int extract_first_dynamic_BASETYPE(pvoid plist,
                                   all_type pvalue,
                                   punsi psize);
```

- plist: lista dal cui inizio estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco temporaneo ptmp

2 ptmp = plist→pstart

3 plist→pstart = ptmp→pnext

4 restituisco valore contenuto in ptmp

5 free(ptmp)

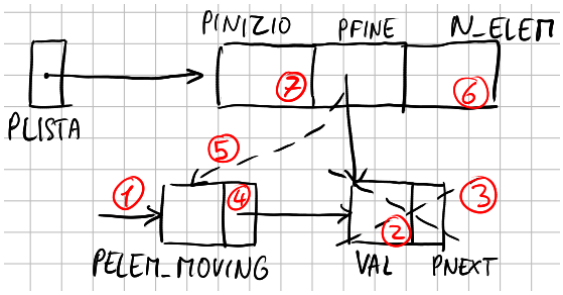
6 n_elem–

7 if(n_elem == 1) plist→pfine = NULL

4.2.9 extract_last

```
int extract_last_dynamic_BASETYPE(pvoid plist,
                                  all_type pvalue,
                                  punsi psize);
```

- plist: lista dalla cui coda estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- return: 1 se tutto va bene, 0 altrimenti



1 raggiungo il penultimo elemento pelem_moving con un for che continua finché pelem_moving→pnext == plist→pend

2 restituisco valore contenuto in plist→pend

3 free(plist→pend)

4 pelem_moving→pnext = NULL

5 plist→pend = pelem_moving

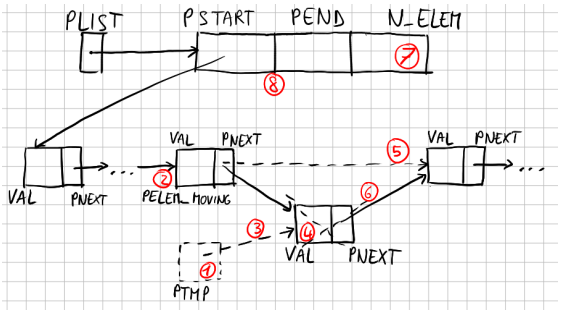
6 n_elem–

7 if(n_elem == 1) plist→pstart = NULL

4.2.10 extract_nth

```
int extract_nth_dynamic_BASETYPE(pvoid plist, all_type pvalue, punsi psize, unsi n);
```

- plist: lista da cui estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- indice da cui estrarre l'elemento. Le posizioni sono contate a partire da 1: n=1 vuol dire estrarre l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco ptmp

2 raggiungo il penultimo elemento pelem_moving con un for

3 ptmp = pelem_moving→pnext

4 restituisco il valore contenuto in ptmp

5 pelem_moving→pnext = ptmp→pnext

6 free(ptmp)

7 n_elem–

8 faccio check sulla lunghezza della lista per sapere se aggiornare pstart o pend

4.2.11 search_first

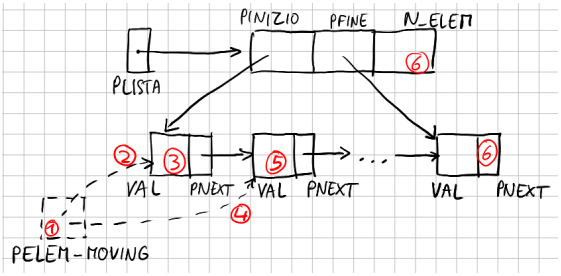
```
int search_first_dynamic_BASETYPE(pvoid plist,
                                  all_type value_searched, unsi size_searched,
                                  all_type pvalue_found,   punsi psize_found,
                                  pcustom_compare pinput_compare);
```

- plist: indirizzo della lista in cui cercare l'elemento
- value_searched: valore con cui confrontare l'elemento da cercare, da castare ad all_type
- size_searched: dimensione dell'elemento da cercare, utile per cercare elementi di tipo generic
- pvalue_found: indirizzo in cui scrivere il valore dell'elemento trovato
- psize_found: indirizzo in cui scrivere la dimensione dell'elemento trovato (nel caso di liste con tipo di dato generic)
- pinput_compare: funzione del tipo

  ```
  int pcustom_compare(all_type value1, unsi size1,
                      all_type value2, unsi size2);
  ```

  con cui ciascun elemento della lista viene confrontato (come value1) con l'elemento da cercare (come value2). La funzione deve tornare 0 quando i due elementi sono uguali, e in quel caso l'elemento trovato viene scritto in pvalue_found e psize_found.

- return: 1 se lo ha trovato, 0 altrimenti



1 creo elemento pelem_moving con cui scorrere la lista

2 pelem_moving = plist→pstart

3 se pelem_moving→val è uguale a value_searched secondo pinput_compare, lo scrivo in pvalue_found ed esco

4 pelem_moving = pelem_moving + pelem_moving→pnext

5 confronto pelem_moving e value_searched, se non sono uguali vado avanti

6 se pelem_moving == NULL, cioè sono arrivato alla fine della lista, ritorno 0 perché non l'ho trovato

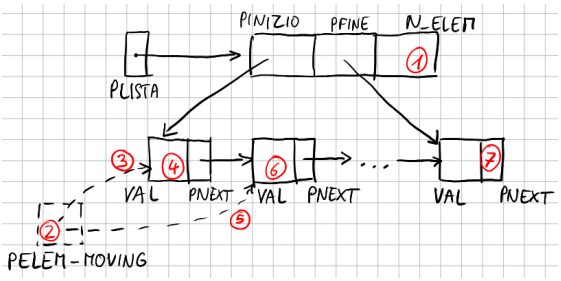
4.2.12 print_list

```
int print_list_dynamic_BASETYPE(pvoid plist,
                                pcustom_print pinput_print);
```

- plist: lista da stampare
- pinput_print: funzione del tipo

  ```
  int pcustom_print(all_type value, unsi size);
  ```

  con cui stampare un elemento della lista. Deve tornare 1 se lo stampa correttamente, 0 altrimenti.
- return: 1 se tutto va bene, 0 altrimenti



1 stampo le informazioni sulla lista

2 creo elemento pelem_moving con cui scorrere la lista

3 pelem_moving = plist→pstart

4 stampo pelem_moving con pinput_print

5 pelem_moving = pelem_moving + pelem_moving→pnext

6 stampo pelem_moving e ripeto

7 quando pelem_moving == NULL, cioè sono arrivato alla fine della lista, ritorno 1

4.3 Lista con tabella

Il funzionamento della $list\_table$ si basa su una grande zona di memoria preallocata (tabella o $table$) in cui coesistono più liste.
Ogni lista è individuata da un unsigned corrispondente all'indice della tabella in cui si trova il suo primo elemento. Ciascun elemento contiene il valore salvato e un unsigned corrispondente all'indice dell'elemento successivo, fino all'ultimo elemento che ha per idx_next 0.

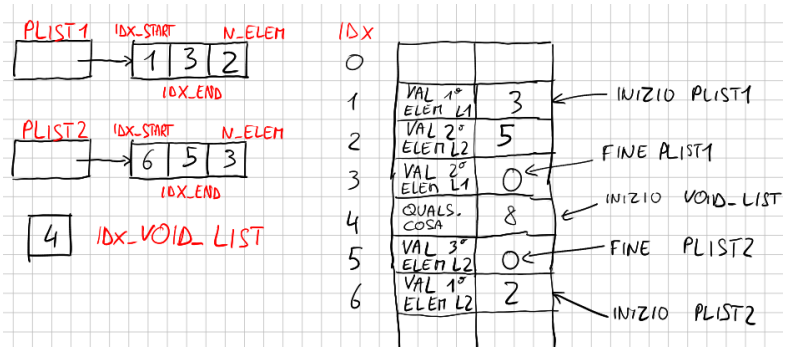


Figura 4: In questo esempio la tabella contiene due liste, che iniziano rispettivamente all'indice 1 e 6, con 2 e 3 elementi.

In realtà la tabella contiene un'ulteriore lista, detta $void\_list$ (che nell'esempio precedente inizia all'indice 4), che permette di ottenere il primo elemento libero della tabella.
Quando ad esempio devo inserire un elemento posso inserirlo nell'indice corrispondente a idx_void_list e spostare idx_void_list all'indice successivo (nel caso precedente l'indice 8).

Come per la lista dinamica, utilizzeremo le struct di C per i tipi base e generic mentre gestiremo la memoria manualmente per i tipi array (dato che i loro elementi hanno dimensione variabile in base al numero di elementi dell'array contenuto).

4.3.1 Struttura tipi base e generic

Le due strutture sono molto simili, cambia essenzialmente come salvare e restituire i dati salvati nella lista.
Ciascuna classe base e la classe generic contengono due variabili static: $ptable$, l'indirizzo alla tabella di quella classe, e idx_void_list, l'indice di partenza della $void\_list$ di quella specifica tabella.

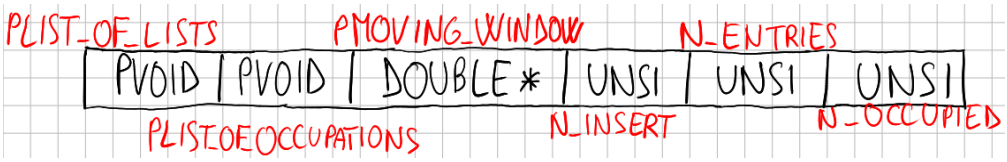
$ptable$ è inizializzata a NULL e allocata solo quando viene creata la prima lista del tipo corrispondente.
idx_void_list è inizializzata a 1 perché quando la tabella è stata appena creata è una lista che scorre tutti gli indici $1 \to 2 \to 3 \to ....$

Altre informazioni sulla tabella che bisogna conservare sono: il $type\_resize$, la $list\_of\_lists$ (una lista dinamica che contiene le liste contenute nella tabella, si veda Sezione 4.3.3), la $list\_of\_occupations$, la $pmoving\_window$ e n_insert (per i quali si rimanda a Sezione 4.3.5), il numero di elementi complessivi della tabella $n\_entries$ e il numero di elementi occupati attualmente $n\_occupied$.

Poiché il fine tabella è individuato dal valore 0, il primo elemento della tabella è inutilizzabile per le liste, per cui si è scelto di mettere il $type\_resize$, che è un unsigned molto piccolo, nell'idx_next dell'elemento 0.

Le altre informazioni della lista sono contenute nella struct:

```
typedef struct _table_info_BASETYPE{
  pvoid  plist_of_lists;
  pvoid  plist_of_occupations;
  double pmoving_window[DIM_MOVING_WINDOW];
  unsi   n_insert;
  unsi   n_entries;
  unsi   n_occupied;
} table_info_BASETYPE;
```



che è posta in memoria subito prima della tabella. Si è scelto di non rendere anche questa struct static perché vi si deve accedere molto meno spesso che idx_void_list, e così viene allocato il suo spazio di memoria solo quando venga creata la tabella.

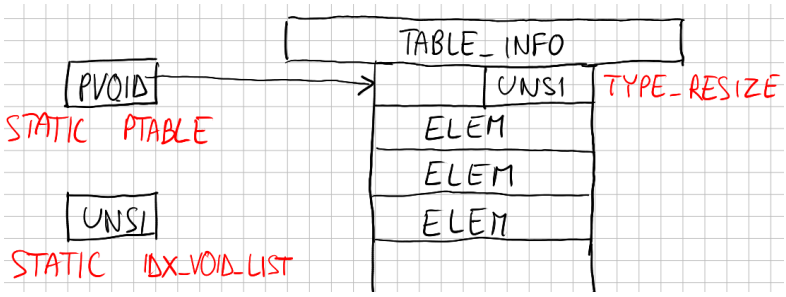
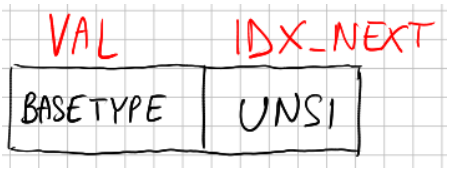


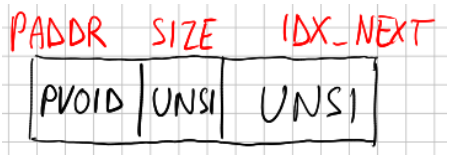
Figura 5: Schema di tutte le informazioni da salvare della tabella: due variabili static, idx_void_list e $ptable$ (che punta all'elemento 0 della tabella), il $type\_resize$ (contenuto nell'idx_next dell'elemento 0, la struct $table\_info$ (salvata in memoria subito prima della tabella).

Come si vede dalla figura precedente la tabella è composta da $n\_entries + 1$ (perché l'elemento 0 non è utilizzabile) elementi, del tipo:

```
typedef struct _elem_table_BASETYPE{
  BASETYPE val;
  unsi     idx_next;
 } elem_table_BASETYPE;
```

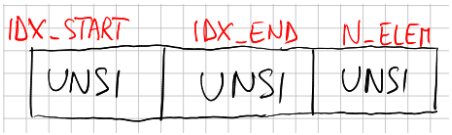



```
typedef struct _elem_table_generic{
  pvoid paddr;
  unsi size;
  unsi idx_next;
 } elem_table_generic;
```



Ciascuna lista infine è individuata dalla struct:

```
typedef struct _list_table_BASETYPE {
  unsi idx_start;
  unsi idx_end;
  unsi n_elem;
 } list_table_BASETYPE;
```



dove

idx_start e idx_end sono rispettivamente l'indice nella tabella del primo e dell'ultimo elemento della lista.

4.3.2 Struttura tipi array

La gestione di questi tipi è più complicata. Ci sono due problematiche:

- non possiamo usare le struct di C per gli elementi della tabella dato che questi hanno dimensione variabile in base al numero di elementi contenuti in ciascun array;
- siccome gli elementi di ciascuna tabella devono avere tutti la stessa size, ci servono tante tabelle quante sono le liste di array di lunghezza diversa da salvare.

La prima problematica è risolta come nella lista dinamica: la memoria è calcolata manualmente e pari a $size\_array * sizeof(BASETYPE) + sizeof(unsi)$.
$elem \rightarrow idx\_next$ è sostituito invece dalla macro GET_IDX_NEXT:

```
#define GET_IDX_NEXT(pvalue) \
  (*((punsi)((pchar) pvalue )+sizeof_array )))
```

che a partire da pvalue si sposta del sizeof_array e legge l'elemento successivo come unsigned.
Un'altra macro utilizzata è GET_NEXT_ELEM:

```
#define GET_NEXT_ELEM(pvalue, i) \
  (((pchar) pvalue ) + (sizeof_array + sizeof(unsi))* i)
```

che permette di spostarsi di i indici nella tabella a partire da pvalue. Questa macro sostituisce l'analogo con le struct:
$elem = pvalue + i$ diventa $elem = GET\_NEXT\_ELEM(pvalue, i)$.

La seconda problematica richiede necessariamente l'indicizzazione di più tabelle, ciascuna con la propria idx_list_void e le proprie info.
Per ogni classe di tipo array (array_int, array_float, ...) ho una variabile static pptables che è inizializzata a NULL ma punta a una lista dinamica di tabelle quando viene creata la prima lista del tipo corrispondente.

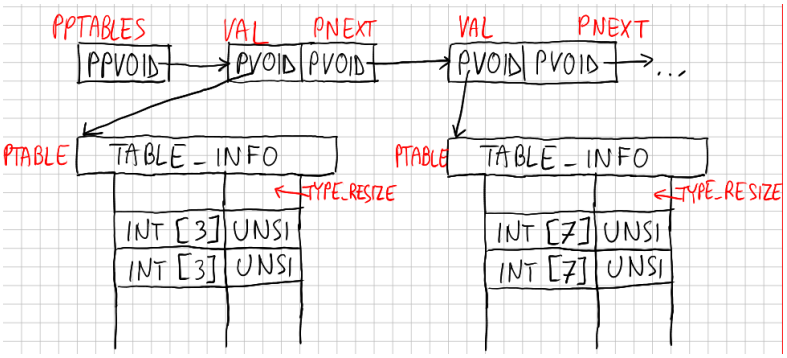


Figura 6: Esempio di *pptables*: contiene gli indirizzi alle tabelle che contengono array di size diversa.

Le informazioni della tabella contengono anche idx_list_void e $sizeof\_array$:

```
typedef struct _table_info_array_BASETYPE {
  pvoid  plist_of_lists;
  pvoid  plist_of_occupations;
  double pmoving_window[DIM_MOVING_WINDOW];
  unsi   n_insert;
  unsi   sizeof_array;
  unsi   idx_void_list;
  unsi   n_entries;
  unsi   n_occupied;
} table_info_array_BASETYPE;
```

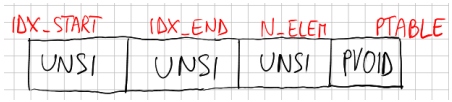
Inoltre, mentre prima lo static ptable puntava all'elemento 0 della tabella, in questo caso pptables contiene gli indirizzi della struct $table\_info$, e l'elemento 0 si può ottenere da:

```
ptable_start = ((table_info) ptable) + 1;
```

Si è scelta questa convenzione perché in ogni caso bisogna accedere a $table\_info$ (ad esempio per ottenere idx_void_list che serve quasi sempre) mentre prima spesso ci interessava solo l'indirizzo dell'elemento 0.

Quando una seconda lista dello stesso tipo viene creata, si cerca nella lista delle tabelle se c'è una tabella con uguale dimensione: se non c'è viene aggiunta alla lista un'ulteriore tabella.
Ovviamente questa è un'operazione dispendiosa ma che va fatta solo in fase di allocazione: una volta trovata o creata la tabella corrispondente, infatti, questa viene aggiunta alla struct della lista:

```
typedef struct _list_table_array_BASETYPE {
  unsi idx_start;
  unsi idx_end;
  unsi n_elem;
  pvoid ptable;
} list_table_array_BASETYPE;
```



Avere un numero di tabelle crea un'ulteriore problematica. Le tabelle non sono statiche, ma possono essere create, distrutte o sostituite (ad esempio in fase di resize).
Tuttavia nel momento in cui cambiamo l'indirizzo di una tabella, tutte le liste in essa contenuta contengono ancora l'indirizzo della tabella precedente, che quindi va modificato. Per questo si è implementata la $list\_of\_lists$.

4.3.3 list_of_lists

La $list\_of\_lists$ è una lista dinamica che contiene gli indirizzi di tutte le liste contenute in una tabella. Ha l'obiettivo di risolvere due problemi (che sorgono principalmente nel caso di resize):

- aggiornare l'idx_start e idx_end delle liste contenute nella tabella quando queste vengono spostate (ad esempio a seguito di un rimpicciolimento) e questo è un problema presente nel caso di tipi singoli, generic e array;
- nel caso di tipi array, aggiornare l'indirizzo della tabella in cui è contenuta nella lista (plist→ptable) quando questa viene sostituita (ad esempio a seguito di qualsiasi operazione di resize).

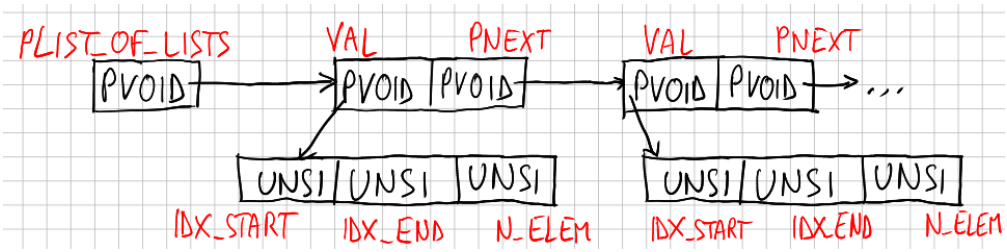


Figura 7: Esempio di $list\_of\_lists$: contiene gli indirizzi alle struct delle liste contenute nella tabella in modo da poter modificare idx_start, idx_end o $ptable$ (nel caso di tipi array) in caso di necessità (ad esempio dopo resize).

4.3.4 resize manuale

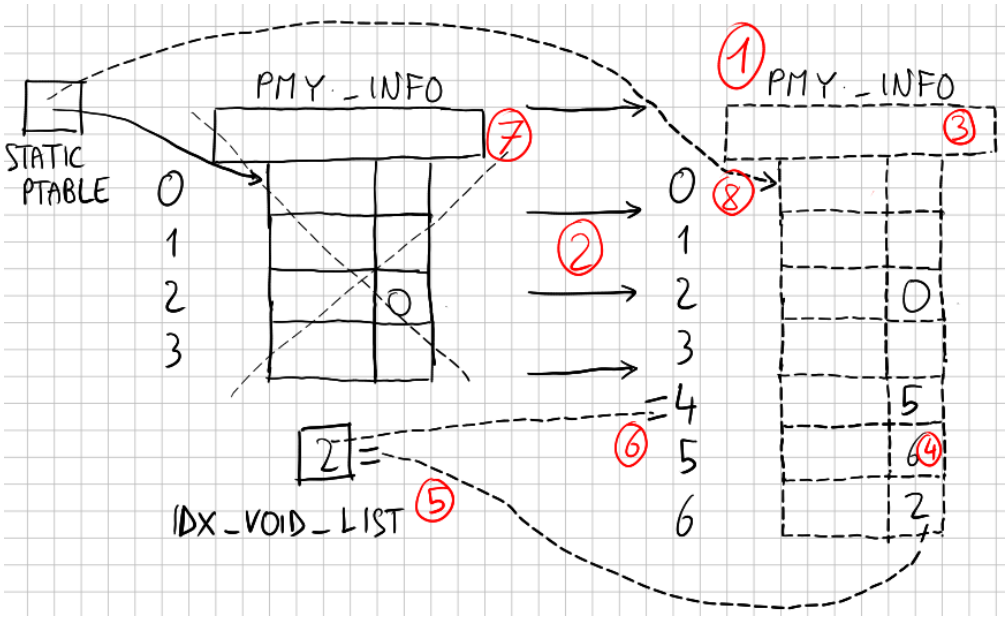
È previsto che la tabella venga ampliata automaticamente quando riempita (se è stato scelto type_resize_default) o che in ogni caso possa essere ridimensionata manualmente attraverso la funzione:

```
int resize_table_table_BASETYPE(pvoid plist, unsi n_entries);
```

- plist: lista contenuta nella tabella da modificare. È utile per le liste di tipo array (in cui vengono gestite più tabelle) per specificare quale ridimensionare
- n_entries: numero complessivo di elementi della tabella dopo il ridimensionamento
- return: 1 se tutto va bene, 0 altrimenti

Di seguito sia pmy_info la struct contenente le info sulla tabella. Si hanno 3 casi:

- $n\_entries < pmy\_info \rightarrow n\_occupied$
  la funzione ritorna 0 perché per ridimensionare la tabella dovrebbe rimuovere elementi da qualche lista
- $n\_entries > pmy\_info \rightarrow n\_entries$
  La tabella deve essere ingrandita. Questo caso è più semplice del successivo perché possiamo usare memcpy sulla vecchia tabella e le liste contenute mantengono intatti i loro idx_start e idx_end. Si ha in particolare:



1 pnew_table = malloc(sizeof(table_info) + sizeof(elem) * (n_entries + 1))
alloco la nuova tabella (con n + 1 elementi perché l'elemento 0 è inutilizzabile)

2 memcpy(pnew_table, ptable, sizeof(table_info) + sizeof(elem) * ptable→n_entries)
copio tutta la tabella vecchia, comprese le informazioni

3 aggiorno le informazioni della nuova tabella (n_entries = nuovo_n_entries)

4 a partire dall'elemento con indice n_entries + 1, scrivo come idx_next l'indice dell'elemento successivo in modo da creare una lista che termina all'ultimo elemento

5 metto come idx_next dell'ultimo elemento idx_void_list

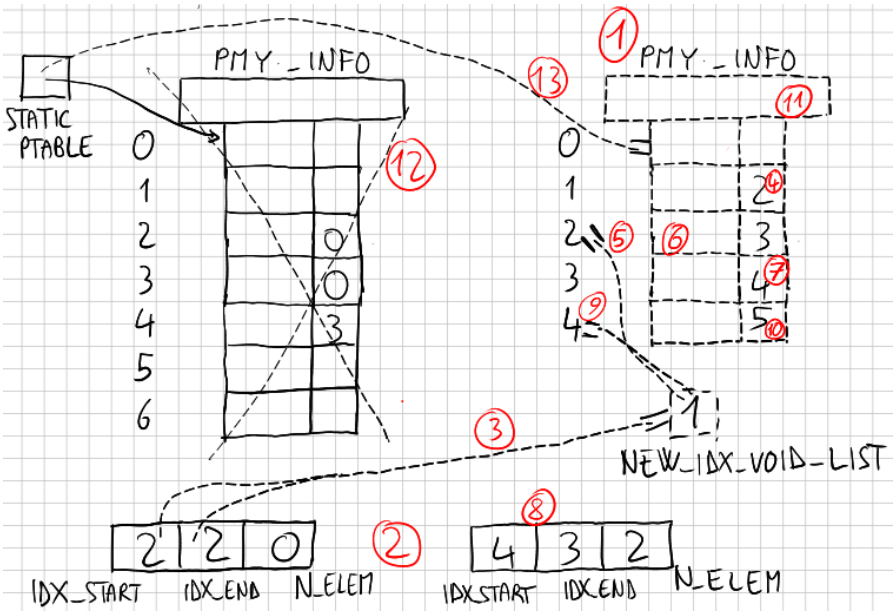
6 sposto idx_void_list all'indice iniziale della lista così creata (n_entries + 1)

7 free(ptable)

8 ptable = ((ptable_info)pnew_table) + 1

NB: nel caso di liste con array il punto 8 cambia leggermente perché non ho ptable come variabile static ma devo anche scorrere tutte le liste contenute in pmy_info→plist_of_lists e aggiornare il loro campo ptable

- $n\_entries < pmy\_info \rightarrow n\_entries$
  La tabella va rimpicciolita. Questo caso è più complicato dei precedenti perché le liste devono cambiare i loro idx_start e idx_end (basti pensare a una lista che ha idx_start compreso tra n_entries e pmy_info→n_entries).
  Si procede come di seguito:



1 pnew_table = malloc(sizeof(table_info) + sizeof(elem) * (n_entries + 1))
alloco la nuova tabella (con n + 1 elementi perché l'elemento 0 è inutilizzabile)

2 copio le liste contenute in ptable (le estraggo da ptable→plist_of_lists). La copia si differenzia tra:
liste con zero elementi, ad esempio quella a sinistra
liste con più elementi, ad esempio quella a destra

3 per copiare la lista con zero elementi basta prendere come idx_start e idx_end l'idx_void_list corrente (1 nell'esempio)

4 cambio l'idx_next del nuovo primo elemento della lista

5 aumento di 1 idx_void_list dato che tutti gli elementi successivi sono al momento liberi

6 per copiare la lista a destra, che ha due elementi, parto dall'elemento con indice idx_void_list, copio il primo elemento della lista, passo all'elemento con indice idx_void_list + 1, copio il secondo elemento della lista e così via.
Alla fine avrò ricopiato la lista in modo ordinato dall'alto al basso dall'indice idx_void_list all'indice idx_void_list + plist_copiata→n_elem

7 scrivo IDX_FINE_LISTA come idx_next dell'ultimo elemento della lista copiata

8 plist_copiata→idx_start = idx_void_list
plist_copiata→idx_end = idx_void_list + plist_copiata→n_elem

9 idx_void_list += plist_copiata→n_elem + 1

10 dopo aver copiato tutte le liste, metto come idx_next di tutti gli elementi l'indice dell'elemento seguente, così da creare una lista che parte da idx_void_list e termina nell'ultimo elemento, che ha per idx_next IDX_FINE_LISTA

11 aggiorno le informazioni della nuova tabella (n_entries = nuovo_n_entries,, copio pmoving_window (per la quale si rimanda a Sezione 4.3.5)

12 free(ptable)

13 ptable = ((ptable_info)pnew_table) + 1

4.3.5 resize automatico

Il resize automatico avviene quando si cerca di inserire un elemento oltre la capienza della tabella e il suo tipo di resize è type_resize_default.
In generale si possono scegliere due strade:

- aumentare la tabella di una quantità fissa ogni volta che si riempie;
- utilizzare un meccanismo predittivo e ingrandire la tabella proporzionalmente a quanto velocemente si riempie.

È stata scelta la seconda strada, che richiede la raccolta di statistiche sull'utilizzo della tabella.
Di seguito vedremo come viene implementata nel programma, l'idea comunque è che ogni tot inserimenti viene salvato il numero di elementi occupati n_occupied in una finestra mobile e calcolata la media mobile corrispondente a quel numero di inserimenti.
Le varie medie mobili sono salvate in una lista dinamica assieme al numero di inserimenti corrispondente.
Quando la tabella si riempie uso i dati salvati nella lista per calcolare un fit lineare.
Infine, basta scegliere il numero di inserimenti futuro in cui, secondo la nostra statistica, la tabella sarà di nuovo piena e prendere il corrispondente n_occupied.
Si è scelto di prendere n_insert_futuro = 3 * n_insert, di modo che se ci fossero solo insert (coefficiente della retta = 1), il numero di elementi della tabella sarebbe moltiplicato per 3, mentre per velocità di occupazione minori la tabella cresce sempre meno.
Le informazioni aggiunte alla tabella a questo scopo sono pmoving_window, plist_of_occupations e n_insert.
Vediamo come viene implementato in C:

1. in fase di creazione della tabella alloco:

   ```
   plist_of_occupations=malloc_list(type_list_dynamic, "DOUBLE", 2);
   ```

2. Ogni volta che viene effettuata un'operazione di inserimento (inclusa la malloc perché riserva uno spazio per la lista istanziata), n_insert viene aumentato di 1.

3. 1. Se ((n_insert % OCCUP_FREQ) == 0), cioè a passi multipli di OCCUP_FREQ (uguale a 100 e definito in defines_typedef.hidden), faccio

      ```
      pmoving_window[idx_moving_window%DIM_MOVING_WINDOW]=n_occupied
      ```

      dove idx_moving_window = n_insert / OCCUP_FREQ è l'indice che aumenta di 1 ogni volta che viene aggiunto un elemento all'array. DIM_MOVING_WINDOW è uguale a 5 e definito in defines_typedef.hidden.
   2. se idx_moving_window è maggiore di DIM_MOVING_WINDOW allora l'array è pieno e posso calcolare la media mobile;
   3. aggiungo alla plist_of_occupations la coppia (n_insert, mean_n_occupied)

4 quando la tabella si riempie ho due casi (siano n_elem il numero di elementi della plist_of_occupations, MIN_OCCUP_STAT il numero minimo di elementi per effettuare il fit (uguale a 5 e definito in defines_typedef.hidden)):

   1 $n\_elem < MIN\_OCCUP\_STAT$
   ho troppi pochi dati per fare il fit, aumento semplicemente la tabella di 3 volte (scelta arbitraria perché si presume che la tabella sia piccola dato che si è riempita prima di accumulare abbastanza statistica)

   2 $n\_elem > MIN\_OCCUP\_STAT$
   in una funzione ausiliaria estraggo le coppie (n_insert, n_occupied) da plist_of_occupations e calcolo i coefficienti della best fit lineare.
   Lancio

   ```
   resize_table_table_generic(NULL, a * n_insert * 3 + b);
   ```

5 quando ridimensiono la tabella (automaticamente o manualmente) devo memcopiare pmoving_window della vecchia tabella in quello della nuova tabella e assegnare la stessa plist_of_occupations in modo da poter usare la stessa statistica.

4.3.6 Differenze tra tipi base, generic e array

Le funzioni riportate di seguito sono analoghe per i tre tipi, cambia:

- come salvare e restituire i dati contenuti nella lista, per cui si rimanda alla Sezione 4.2.2;
- come muoversi lungo la tabella, che avviene attraverso le struct di C nei casi di tipi base e generic e con le macro GET_IDX_NEXT e GET_NEXT_ELEM nel caso di tipi array, per le quali si rimanda a 4.3.2;
- come ottenere l'indirizzo della tabella, che è contenuto nella variabile static ptable nei tipi base e generic e nella struct della lista (plist→ptable) nel caso dei tipi array.

Ulteriori differenza saranno presentate nei vari casi specifici.

4.3.7 Notazione

Mentre nella lista dinamica si lavorava sempre con puntatori, adesso molte operazioni diventano assegnamenti tra unsigned. Indico l'assegnamento con una freccia che termina con un uguale:

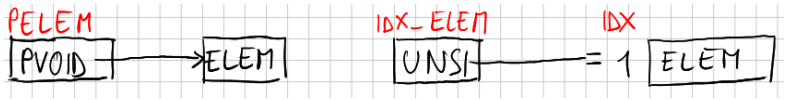


Figura 8: Notazione utilizzata di seguito: la freccia indica l'assegnamento di una variabile pvoid, la freccia con uguale indica l'assegnamento di un unsigned.

4.3.8 malloc_list

Sono disponibili due funzioni:

```
pvoid malloc_list_table_BASETYPE(unsi dim_array);
```

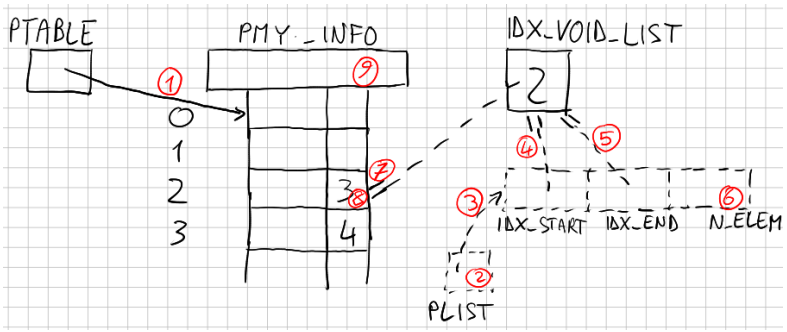
- dim_array: numero di valori contenuto in ciascun elemento della lista
- return: l'indirizzo della nuova lista istanziata, NULL se non è andato a buon fine

che essenzialmente chiama, con i parametri di default type_resize_default e TABLE_DEFAULT_DIM:

```
pvoid malloc_list_specify_table_table_BASETYPE(
        unsi dim_array,
        type_resize type_resize,
        unsi dim_table);
```

- dim_array: numero di valori contenuto in ciascun elemento della lista
- type_resize: tipo di resize della tabella, qualora questa non esistesse ancora
- dim_table: numero di elementi complessivo della tabella, qualora questa non esistesse ancora
- return: l'indirizzo della nuova lista istanziata, NULL se non è andato a buon fine

type_resize e dim_table vengono applicati solo se la tabella non esiste ancora per evitare incompatibilità con altre liste che abitassero la stessa tabella.



1 if(ptable == NULL) creo tabella (si veda 4.3.9)

2 creo puntatore alla nuova lista plist

3 plist = malloc(sizeof(list_table))

4 if(idx_void_list != IDX_FINE_LISTA) (cioè c'è ancora spazio) plist→idx_start = idx_void_list

5 plist→idx_end = idx_void_list

6 plist→n_elem = 0

7 idx_void_list = (pfirst_elem = (ptable + idx_void_list))→idx_next

8 pfirst_elem→idx_next = IDX_FINE_LISTA

9 (pmy_info→n_occupied)++ dove pmy_info = ((ptable_info) ptable) - 1

NB: nel caso di tipi array il numero 1 diventa:

- if(pptable == NULL) creo lista di tabella
- if(!search_first) creo una nuova tabella e la aggiungo alla lista, altrimenti uso la ptable trovata. search_first (Sezione 4.2.11) trova, se c'è, la tabella che contiene array di uguale dimensione

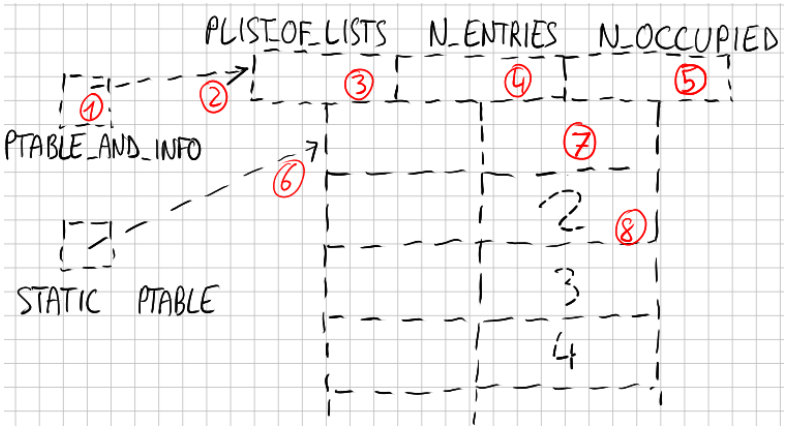
e si aggiunge

- plist→ptable = ptable

4.3.9 create_table

```
pvoid create_table_BASETYPE(type_resize type_resize, unsi dim);
```

- type_resize: tipo di resize della tabella
- dim: numero di elementi complessivo della tabella
- return: l'indirizzo della tabella, NULL se non è andato a buon fine. Il return dell'indirizzo è importante per le classi table_array_BASETYPE in cui vanno gestite diverse tabelle.



1 creo puntatore a nuova tabella ptable_and_info

2 ptable_and_info = malloc(sizeof(table_info + sizeof(elem) * (dim + 1))).
dim è preso in input e alloco dim+1 elementi perché il primo è inutilizzabile dato che IDX_FINE_LISTA = 0

3 ptable_and_info→plist_of_lists = malloc_list(type_list_dynamic, "PVOID", 1)

4 ptable_and_info→n_entries = dim

5 ptable_and_info→n_occupied = 0

6 ptable = (pvoid)(ptable_and_info + 1)

7 ((pelem)ptable)→idx_next = type_resize

8 scrivo i+1 come idx_next dell'elemento i-esimo in modo che la *void_list* inizi dall'indice 1 e finisca nell'ultimo elemento della lista, che ha per idx_next IDX_FINE_LISTA

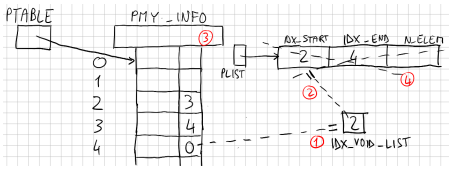
NB: nel caso di tipi array, devo aggiungere i seguenti passaggi:

- ptable_and_info→sizeof_array = sizeof_array (con sizeof_array preso in input)
- insert_first(pptables, (all_type)ptable, 0)

4.3.10 free_list

```
void free_list_table_BASETYPE(pvoid plist);
```

- plist: indirizzo della lista da liberare
- return: niente



1 (ptable + plist→idx_end)→idx_next = idx_void_list

2 idx_void_list = plist→idx_start
(così ho messo la lista da liberare in cima alla $void\_list$

3 aggiorno le info della tabella
(n_occupied -= plist→n_elem; tolgo la lista da list_of_lists)

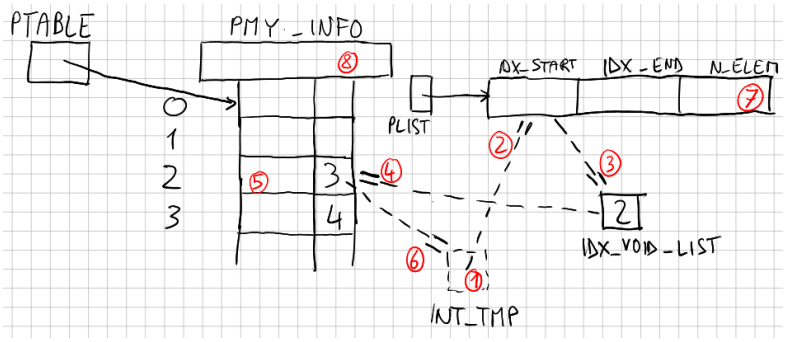
4 free(plist)

NB: nel caso di list_table_generic devo anche liberare gli indirizzi contenuti in ciascun elemento, quindi scorro sulla lista con pelem_moving finché pelem_moving→idx_next == IDX_FINE_LISTA e faccio free(pelem_moving→paddr)

4.3.11 insert_first

```
int insert_first_table_BASETYPE(pvoid plist, all_type value, unsi size);
```

- plist: lista alla cui cima inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco int_tmp

2 int_tmp = plist→idx_start
(salvo l'indice di quello che diventerà il secondo elemento)

3 plist→idx_start = idx_void_list

4 idx_void_list = ((pnew_elem = (ptable + idx_void_list))→idx_next
(l'indice del successivo elemento della $void\_list$)

5 scrivo il valore preso in input in pnew_elem→val

6 pnew_elem→idx_next = int_tmp
(che contiene il vecchio indice iniziale della lista)

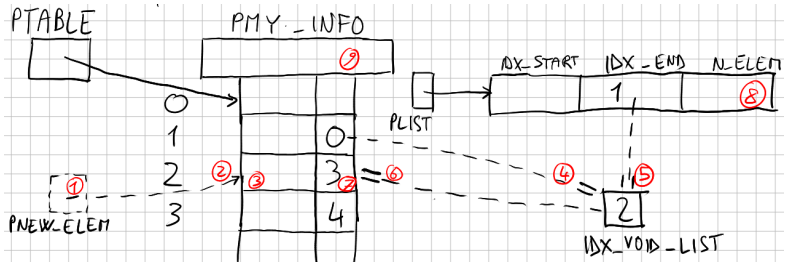
7 n_elem++

8 ((((ptable_info) ptable) - 1)→n_occupied)++

4.3.12 insert_last

```
int insert_last_table_BASETYPE(pvoid plist, all_type value, unsi size);
```

- plist: lista alla cui coda inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco pvoid pnew_elem

2 pnew_elem = ptable + idx_void_list

3 scrivo il valore preso in input in pnew_elem→val

4 (ptable + plist→idx_fine)→idx_next = idx_void_list
(aggiorno l'idx_next del penultimo elemento della lista)

5 plist→idx_end = idx_void_list

6 idx_void_list = pnew_elem→idx_next
(l'indice del successivo elemento della $void\_list$)

7 pnew_elem→idx_next = IDX_FINE_LISTA
(essendo pnew_elem l'ultimo elemento della lista)

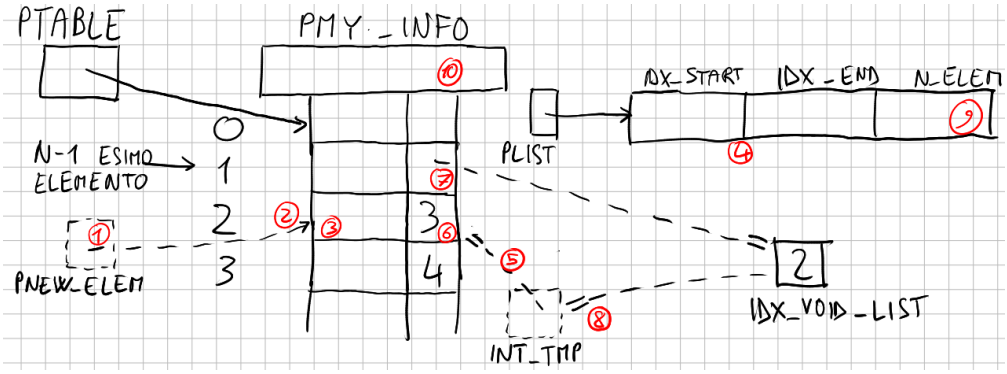
8 n_elem++

9 ((((ptable_info) ptable) - 1)→n_occupied)++

4.3.13 insert_nth

```
int insert_nth_dynamic_BASETYPE(pvoid plist, all_type value, unsi size, unsi n);
```

- plist: lista in cui inserire l'elemento
- value: valore dell'elemento da inserire, da castare ad (all_type)
- size: utile solo nel caso del tipo GENERIC, in cui rappresenta la size dell'elemento da inserire
- indice a cui inserire l'elemento. Le posizioni sono contate a partire da 1: n=1 vuol dire inserire l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.
- return: 1 se tutto va bene, 0 altrimenti



1 creo buco pvoid pnew_elem

2 pnew_elem = ptable + idx_void_list

3 scrivo il valore preso in input in pnew_elem→val

4 faccio check sulla lunghezza per sapere se aggiornare idx_start o idx_end (e porli uguali a idx_void_list)

5 creo buco int_tmp e int_tmp = pnew_elem→idx_next

6 raggiungo l'elemento (n-1)esimo con un for (chiamiamolo pelem_moving) e aggiorno pnew_elem→idx_next = pelem_moving→idx_next

7 pelem_moving→idx_next = idx_void_list
(aggiorno l'idx_next dell'(n-1)esimo elemento)

8 idx_void_list = int_tmp
(l'indice del successivo elemento della $void\_list$)

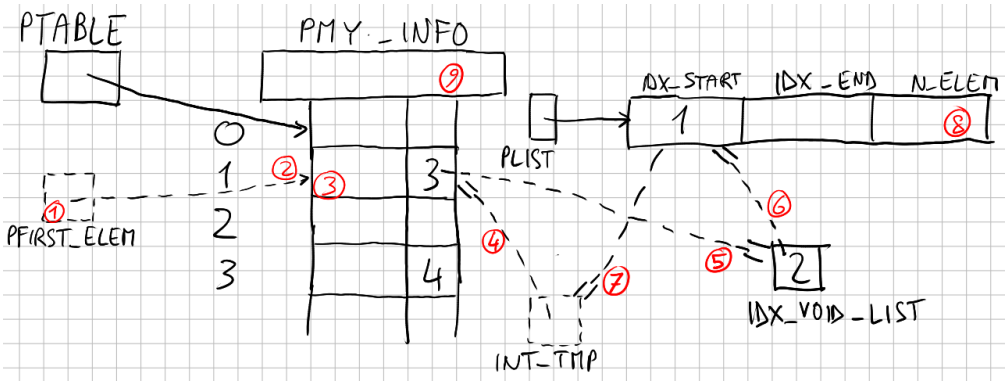
9 n_elem++

10 ((((ptable_info) ptable) - 1)→n_occupied)++

4.3.14 extract_first

```
int extract_first_table_BASETYPE(pvoid plist, all_type pvalue, punsi psize);
```

- plist: lista dal cui inizio estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- return: 1 se tutto va bene, 0 altrimenti



1 creo pfirst_elem (primo elemento della lista cioé elemento da estrarre)

2 pfirst_elem = ptable + plist→idx_start

3 restituisco valore contenuto in pfirst_elem

4 creo buco int_tmp e vi salvo pfirst_elem→idx_next (che sarà poi il nuovo idx_start della lista)

5 pfirst_elem→idx_next = idx_void_list
(perché diventerà il primo elemento della $void\_list$)

6 idx_void_list = plist→idx_start

7 plist→idx_start = int_tmp

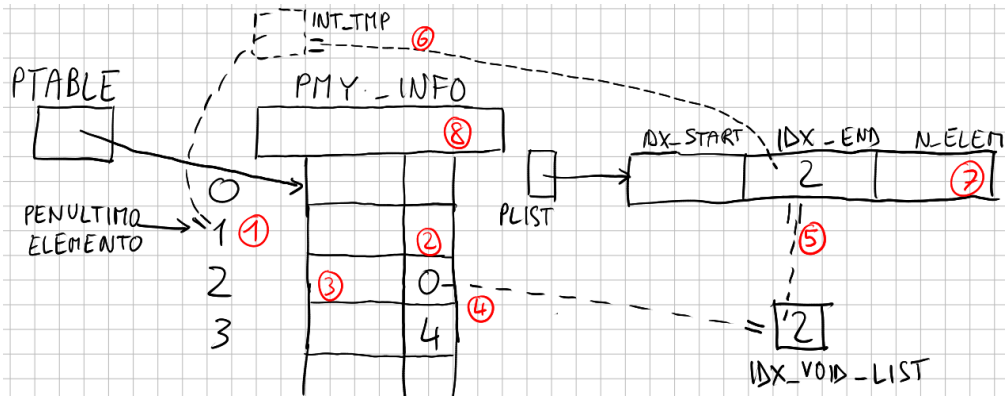
8 n_elem–

9 ((((ptable_info) ptable) - 1)→n_occupied)–

4.3.15 extract_last

```
int extract_last_table_BASETYPE(pvoid plist, all_type pvalue, punsi psize);
```

- plist: lista dalla cui coda estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- return: 1 se tutto va bene, 0 altrimenti



1 raggiungo il penultimo elemento pelem_moving con un for che continua finché pelem_moving→idx_next == plist→idx_end e salvo il suo indice in int_tmp

2 pelem_moving→idx_next = IDX_FINE_LISTA e sposto pelem_moving all'ultimo elemento della lista
(pelem_moving = ptable + plist→idx_end)

3 restituisco valore contenuto in pelem_moving

4 pelem_moving→idx_next = idx_void_list
(perché diventerà il primo elemento della $void\_list$)

5 idx_void_list = plist→idx_end

6 plist→idx_end = int_tmp
(indice del penultimo elemento, che adesso diventa l'ultimo)

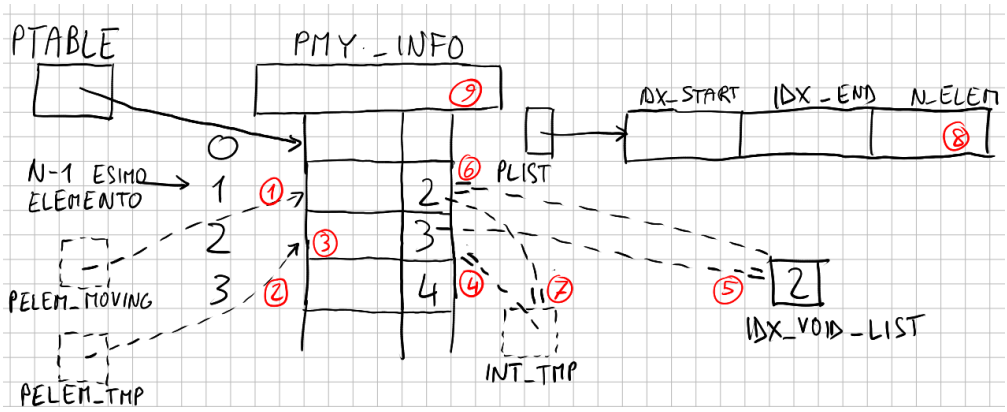
7 n_elem–

8 ((((ptable_info) ptable) - 1)→n_occupied)–

4.3.16 extract_nth

```
int extract_nth_dynamic_BASETYPE(pvoid plist,
                                 all_type pvalue,
                                 punsi psize,
                                 unsi n);
```

- plist: lista da cui estrarre l'elemento
- pvalue: indirizzo in cui scrivere l'elemento estratto, da castare a pvoid e poi ad all_type
- psize: nel caso di dati GENERIC, indirizzo in cui scrivere la size del dato estratto
- indice da cui estrarre l'elemento. Le posizioni sono contate a partire da 1: n=1 vuol dire estrarre l'elemento in cima alla lista, n=2 dopo il primo elemento e cosi' via.
- return: 1 se tutto va bene, 0 altrimenti



1 raggiungo con un for l'(n-1)esimo elemento e lo salvo in pelem_moving

2 salvo in pelem_tmp l'n-esimo elemento (quello da estrarre)

3 restituisco valore contenuto in pelem_tmp

4 creo buco int_tmp e vi salvo pelem_tmp→idx_next
(l'indice dell'(n+1)esimo elemento)

5 pelem_tmp→idx_next = idx_void_list (perché diventerà il primo elemento della $void\_list$)

6 idx_void_list = pelem_moving→idx_next

7 pelem_moving→idx_next = int_tmp

8 n_elem–

9 ((((ptable_info) ptable) - 1)→n_occupied)–

4.3.17 search_first e print_list

```
int search_first_table_BASETYPE(pvoid plist,
                                all_type value_searched, unsi size_searched,
                                all_type pvalue_found,   punsi psize_found,
                                pcustom_compare pinput_compare);
```

- plist: indirizzo della lista in cui cercare l'elemento
- value_searched: valore con cui confrontare l'elemento da cercare, da castare ad all_type
- size_searched: dimensione dell'elemento da cercare, utile per cercare elementi di tipo generic
- pvalue_found: indirizzo in cui scrivere il valore dell'elemento trovato
- psize_found: indirizzo in cui scrivere la dimensione dell'elemento trovato (nel caso di liste con tipo di dato generic)
- pinput_compare: funzione del tipo

  ```
  int pcustom_compare(all_type value1, unsi size1,
                      all_type value2, unsi size2);
  ```

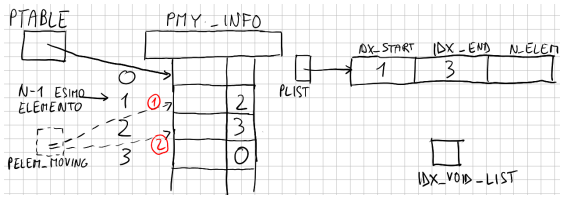
  con cui ciascun elemento della lista viene confrontato (come value1) con l'elemento da cercare (come value2). La funzione deve tornare 0 quando i due elementi sono uguali, e in quel caso l'elemento trovato viene scritto in pvalue_found e psize_found.
- return: 1 se lo ha trovato, 0 altrimenti

```
int print_list_table_BASETYPE(pvoid plist,
                              pcustom_print pinput_print);
```

- plist: lista da stampare
- pinput_print: funzione del tipo

  ```
  int pcustom_print(all_type value, unsi size);
  ```

  con cui stampare un elemento della lista. Deve tornare 1 se lo stampa correttamente, 0 altrimenti.
- return: 1 se tutto va bene, 0 altrimenti



Le due funzioni hanno funzionamento analogo:

1 salvo indirizzo del primo elemento della lista pelem_moving e confronto con pinput_compare o stampo con pinput_print

2 scorro sulla lista finché pelem_moving→idx_next e confronto con pinput_compare o stampo con pinput_print