Strutture dati

Le liste

Introduzione

- Una lista è una successione finita di valori di un tipo (insieme di valori e ordine).
- Come tipo di dato è qualificata dalle operazioni che ci si possono svolgere:
  - inserimento
  - cancellazione
  - visita
  - ricerca
  - inizializzazione

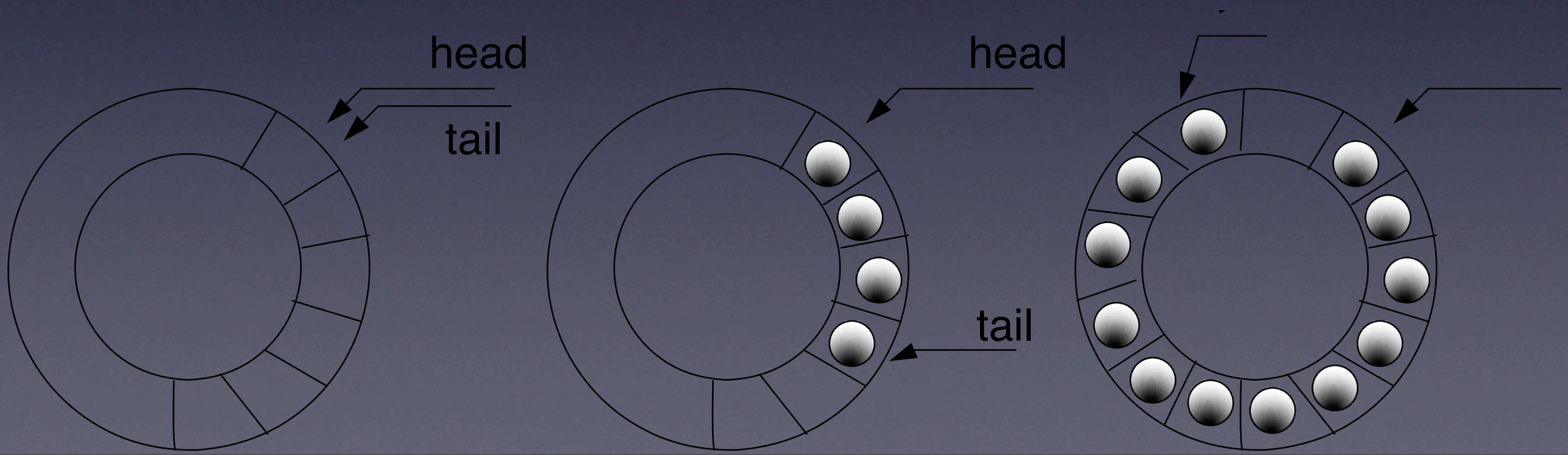
Rappresentazione

- Sequenziale
  ordine codificato dalla posizione dei dati in un vettore
- Collegata
  ordine codificato in modo esplicito, usando indici di un vettore o puntatori

Rappresentazione sequenziale

- Una possibile soluzione è l'implementazione come stack
  - ma è poco pratico per lavorare sulla testa dello stack
- Una soluzione migliore è un buffer circolare
  - si usano due indici: head / tail
  - invariante:
    se head == tail lista ⇒ lista vuota

- cancellazione/inserimento in testa: spostiamo head
- cancella/inserimento in coda: spostiamo tail



```
struct list {
  float *buffer;
  int size;
  int head;
  int tail;
}
...
Boolean pre_insert(struct list *ptr, float value)
{
  if (((ptr->tail+1)%ptr->size)!=ptr->head)
  {
    ptr->head=(ptr->head+ptr->size-1)%ptr->size;
    ptr->buffer[ptr->head]=value;
    return TRUE;
  }
  else
    return FALSE;
}
```

- ptr->head+ptr->size-1
  decrementa head di 1 anche per head == 0

- le operazioni di visita e ricerca possono essere fatte passando un puntatore alla lista per velocità: non si effettua la copia della lista

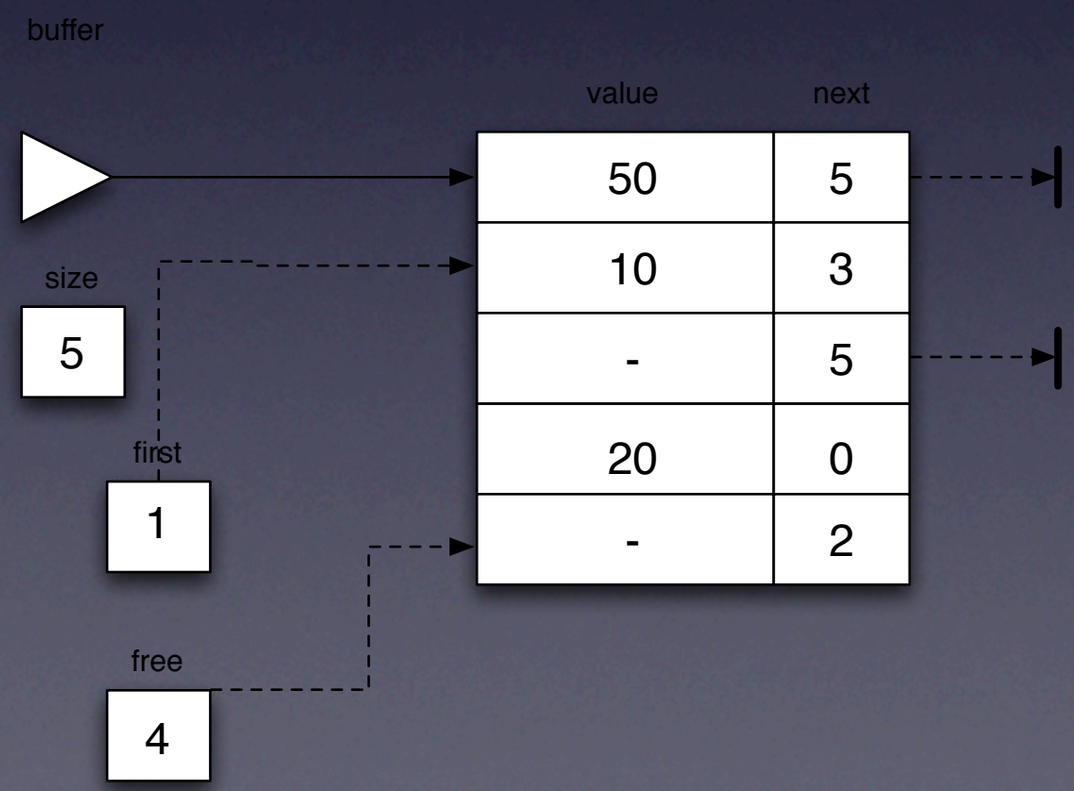
- le operazioni su posizione intermedia richiedono operazioni di copia (di costo lineare)

- dimensione statica del buffer, ma accesso diretto e visita in ordine inverso

Rappresentazione collegata con array e indici

- Si usa un array di record contenenti un valore e l'indice del record successivo
  - l'elemento di coda ha un valore di indice illegale
  - i record non usati sono concatenati in un'altra lista

```
struct record {
  float value;
  int next;
};

struct list {
  int first;
  int free;
  int size;
  struct record *buffer;
};
```



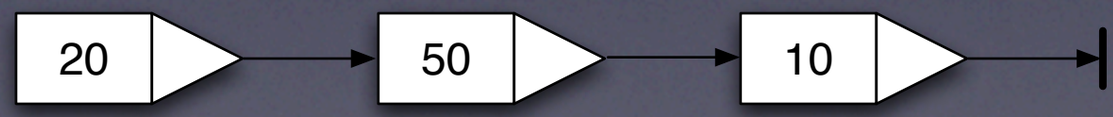
```
Boolean suf_insert(struct list *ptr, float value)
{
  int moved;
  int *positionPtr;

  if(ptr->free!=ptr->size)
  {
    moved=ptr->free;
    ptr->free=((ptr->buffer)[ptr->free]).next;
    positionPtr=&ptr->first;
    while(*positionPtr!=ptr->size)
      positionPtr=&(ptr->buffer[*positionPtr].next);
    *positionPtr=moved
    ptr->buffer[moved].value=value;
    ptr->buffer[moved].next=ptr->size;
    return TRUE;
  } else
    return FALSE;
}
```

- l'inserimento ordinato è come l'inserimento in testa, con una condizione aggiuntiva sul while()

- dimensionamento statico e mancanza di accesso diretto presente nell'altra implementazione

Rappresentazione con puntatori

- i valori sono memorizzati su elementi allocati in posizioni indipendenti
- ciascun valore è collegato al successivo mediante un indirizzo
  - l'indirizzo nullo indica la fine lista



```
struct list{
  float value;
  struct list *nextPtr;
};

void pre_insert(struct list **ppList, float value) {
  struct list *tmpPtr;

  tmpPtr = *ppList;
  *ppList=(struct list *)malloc(sizeof(struct list));
  (*ppList)->value=value;
  (*ppList)->nextPtr=tmpPtr;
}

void suf_insert(struct list **ppList, float value) {

  while(*ppList != NULL)
    ppList = &((*ppList)->nextPtr);

  pre_insert( ppList, value );
}
```

- pre_insert() deve modificare l'indirizzo del primo elemento della lista, quindi deve conoscere l'indirizzo del puntatore che contiene l'indirizzo del primo elemento della lista... serve il doppio puntatore.

- un inserimento intermedio funziona come quello in coda: il ciclo while() ha una condizione di controllo in più

Esempio di errore

```
void pre_insert_mem_leak{struct list *pList, float
value) {
  struct list *tmpPtr;

  tmpPtr = pList;
  pList=(struct list *)malloc(sizeof(struct list));
  pList->value=value;
  pList->nextPtr=tmpPtr;
}



void client(void) {
  struct list *pList;

  pre_insert_mem_leak( pList, value );
}
```

- Le funzioni di visita e ricerca non devono modificare gli elementi della lista: possono ricevere un puntatore, non hanno bisogno di puntatore a puntatore

Ricorsione nei dati

- La definizione di struct list è ricorsiva
  - ma si usa un puntatore: il compilatore sa allocare la memoria necessaria, invece:

```
struct impossibleList {
   float value;
   struct impossibleList;
}
```

è davvero impossibile !

Ricorsione nelle funzioni

- f() è ricorsiva quando delega parte della sua operazione ad una ulteriore istanza di f()
- sia direttamente che indirettamente tramite un'altra funzione

Esempio

```
void visit_r(struct list *pList) {
  if (pList != NULL) {
    printf(“%f\n”,pList->value);
    visit_r(pList->nextPtr);
  }
}

void suf_insert_r(struct list **ppList, value) {
  if (*ppList!=NULL)
    suf_insert_r(ppList=&((*ppList)->nextPtr,value);
  else
    pre_insert(ppList, value);
}
```

- La funzione visit_r è costosa in termini di memoria e tempo di esecuzione: maggiore carico sullo stack
- la soluzione ricorsiva si può adattare meglio alla struttura dati:

```
void visit_r_back(struct list *pList)
{
   if (pList!=NULL) {
      visit_r_back(pList->nextPtr);
      printf("%f\n",pList->value);
   }
}
```