LaB 1: Liste doppiamente collegate, circolari, con sentinella

```
struct list::node {
  Elem info;
  node *prev;
  node *next;
};
```

createEmpty

```
void list::createEmpty(List& li) {
  // visto che la lista è vuota, creo solo la sentinella
  Node *aux = new node;       // nuovo nodo dinamicamente, allocando memoria per un oggetto di tipo
                                 node usando new. il puntatore aux punta al nuovo nodo creato.
  aux -> next = aux;          // essendo una lista circolare vuota, il nodo sentinella (aux) non ha nessun
                                 altro nodo a cui puntare, quindi il suo puntatore next (che di solito
                                 punterebbe al primo nodo della lista) viene impostato su di sé.
  aux -> prev = aux;          // il puntatore prev (che di solito punterebbe all'ultimo nodo della lista)
                                 viene impostato su di sé.
  li = aux;                   // aux primo elemento di li
}
```

clear

```
void list::clear(List& li) {
  node* cur = li -> next;       // salta il nodo sentinella, che non va cancellato, per iniziare a liberare
                                   memoria a partire dal primo vero nodo della lista
// poiché la lista è circolare, il ciclo si ferma quando tutti i nodi sono stati visitati e il puntatore cur ritorna
al nodo sentinella
  while (cur != li) {
    node* temp = cur -> next; // prima di cancellare il nodo corrente, memorizza il puntatore al nodo
                                 successivo in temp
    delete cur;               // dealloca la memoria del nodo puntato da cur
    cur = temp;               // sposta il puntatore cur al nodo successivo (quello precedentemente
                                 memorizzato in temp).
  }
  li -> next = li;            // una volta terminato il ciclo e cancellati tutti i nodi, il puntatore next
                                 del nodo sentinella li viene impostato per puntare di nuovo a sé stesso.
  li -> prev = li;            // anche il puntatore prev viene riportato alla configurazione originale di
                                 una lista vuota
}
```

isEmpty

```
bool list::isEmpty(const List& li) {
  retrurn (li -> next == li);
}
```

size

```
unsigned int list::size(const List& li) {
  unsigned int count = 0;      // Inizializza una variabile count a zero, utilizzata per contare il numero
                                  di nodi presenti nella lista. Si parte da zero poiché, all'inizio, non ci sono
                                  nodi conteggiati.
  node* cur = li -> next;      // Inizializza cur al primo nodo della lista, saltando il nodo sentinella
// Inizia un ciclo while che continua finché cur non punta di nuovo al nodo sentinella.
  while (cur != li) {
    ++count;                   // Ogni volta che il ciclo visita un nodo il contatore aumenta di uno,
                                  registrando la presenza di un nodo nella lista.
    cur = cur -> next;         // Dopo aver contato il nodo attuale, il puntatore cur viene spostato al
                                  nodo successivo nella lista, permettendo al ciclo di continuare a
                                  contare i nodi rimanenti.
  }
  return count;               // Restituisce il conteggio di tutti i nodi tranne la sentinella
}
```

get

```
Elem list::get(unsigned int pos, const List& li){
// Se pos è fuori dai limiti della lista (cioè se ci sono meno nodi di quelli richiesti), genera un'eccezione.
  if (pos >= size(li)) {
    string exc = "errore";
    throw exc;
  }
  unsigned int count = 0;      // Inizializza una variabile count a zero, che verrà utilizzata per contare i
                                  nodi visitati mentre si attraversa la lista per trovare il nodo alla
                                  posizione pos.
  node* cur = li -> next;      // Inizializza un puntatore cur al primo nodo della lista, saltando il nodo
                                  sentinella.
// Questo ciclo scorre i nodi della lista fino a raggiungere la posizione desiderata. Se cur punta di nuovo al
  nodo sentinella o se si raggiunge la posizione pos, il ciclo si ferma.
  while (cur != li && count < pos) {
    ++count;                   // Ogni volta che viene visitato un nodo, il contatore aumenta per tenere
                                  traccia della posizione corrente.
    cur = cur -> next;         // Dopo aver visitato il nodo attuale, cur viene spostato al nodo successivo
                                  nella lista, continuando il ciclo.
  }
  return cur -> info;         // Una volta raggiunta la posizione desiderata, il nodo corrispondente viene
                                  restituito
}
```

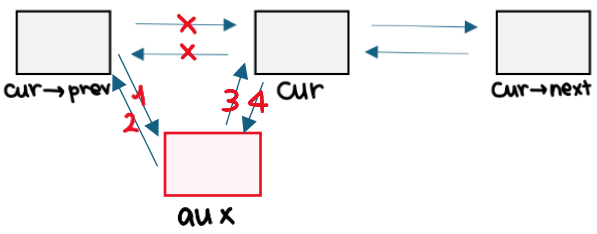
set

```
void list::set(unsigned int pos, Elem el, const List& li){
// Se pos è fuori dai limiti della lista (cioè se ci sono meno nodi di quelli richiesti), genera un'eccezione.
  if (pos >= size(li)) {
    string exc = "errore";
    throw exc;
  }
```

```
  unsigned int count = 0;       // Inizializza una variabile count a zero, che verrà utilizzata per contare i
                                   nodi visitati mentre si attraversa la lista per trovare il nodo alla
                                   posizione pos.
  node* cur = li -> next;       // Inizializza un puntatore cur al primo nodo della lista, saltando il nodo
                                   sentinella.
// Questo ciclo scorre i nodi della lista fino a raggiungere la posizione desiderata. Se cur punta di nuovo al
   nodo sentinella o se si raggiunge la posizione pos, il ciclo si ferma.
  while (cur != li && count < pos) {
    ++count;                    // Ogni volta che viene visitato un nodo, il contatore aumenta per tenere
                                   traccia della posizione corrente.
    cur = cur -> next;          // Dopo aver visitato il nodo attuale, cur viene spostato al nodo successivo
                                   nella lista, continuando il ciclo.
  }
  cur -> info = el;             // Imposta l'informazione del nodo attualmente puntato da cur al nuovo
                                   valore el. Dopo aver raggiunto la posizione desiderata, il valore del nodo
                                   viene aggiornato con il nuovo valore fornito.
}
```

add

```
void list::add(unsigned int pos, Elem el, const List& li){
// Se pos è fuori dai limiti della lista (cioè se ci sono meno nodi di quelli richiesti), genera un'eccezione.
  if (pos > size(li)) {
    string exc = "errore";
    throw exc;
  }
  node* aux = new node;                 // Viene allocato dinamicamente un nuovo nodo della lista
                                           utilizzando l'operatore new. Questo nodo sarà utilizzato per
                                           contenere il nuovo elemento da aggiungere.
  aux -> info = el;                     // Si assegna il valore dell'elemento el (da aggiungere) al
                                           info del nuovo nodo aux.
  unsigned int count = 0;               // Inizializza una variabile count a zero, che verrà
                                           utilizzata per contare i nodi visitati mentre si attraversa la
                                           lista per trovare il nodo alla posizione pos.
  node* cur = li -> next;               // Inizializza un puntatore cur al primo nodo della lista,
                                           saltando il nodo sentinella.
// Questo ciclo scorre i nodi della lista fino a raggiungere la posizione desiderata. Se cur punta di nuovo al
   nodo sentinella o se si raggiunge la posizione pos, il ciclo si ferma.
  while (cur != li && count < pos) {
    ++count;                            // Ogni volta che viene visitato un nodo, il contatore aumenta
                                           per tenere traccia della posizione corrente.
    cur = cur -> next;                  // Cur viene spostato al nodo successivo, continuando il ciclo.
  }
1 cur -> prev -> next = aux;
2 aux -> prev = cur -> prev;
3 aux -> next = cur;
4 cur -> prev = aux;
}
```



addRear

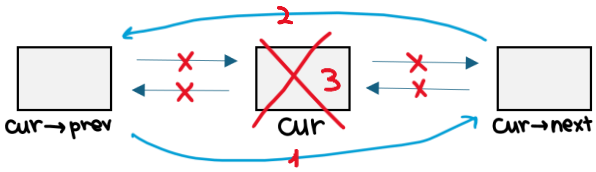
```
void list::addRear(Elem el, const List& li){
  node* aux = new node;          // Viene allocato dinamicamente un nuovo nodo della lista utilizzando
                                    l'operatore new. Questo nodo sarà utilizzato per contenere il nuovo
                                    elemento da aggiungere.
  aux -> info = el;              // Si assegna il valore dell'elemento el (da aggiungere) al info del
                                    nuovo nodo aux.
  node* cur = li -> prev;        // Si crea un puntatore cur che punta all'ultimo nodo attualmente
                                    presente nella lista.
  cur -> next = aux;             // Si aggiorna il puntatore next dell'ultimo nodo attuale (cur),
                                    facendolo puntare al nuovo nodo aux.
  aux -> prev = cur;             // Viene aggiornato il puntatore prev del nuovo nodo aux per farlo
                                    puntare all'ultimo nodo precedente (cur)
  aux -> next = li;              // Il puntatore next del nuovo nodo aux viene impostato per puntare
                                    al nodo sentinella li. Questo è necessario perché li (il nodo
                                    sentinella) segna sia l'inizio che la fine della lista circolare.
  li -> prev = aux;              // Infine, si aggiorna il puntatore prev del nodo sentinella li per farlo
                                    puntare al nuovo nodo aux. In questo modo, il nuovo nodo diventa
                                    l'ultimo nodo della lista.
}
```

addFront

```
void list::addFront(Elem el, const List& li) {
    node* aux = new node;        // Viene allocato dinamicamente un nuovo nodo della lista utilizzando
                                    l'operatore new. Questo nodo sarà utilizzato per contenere il nuovo
                                    elemento da aggiungere.
  aux -> info = el;              // Si assegna il valore dell'elemento el (da aggiungere) al info del
                                    nuovo nodo aux.
  node* cur = li -> next;        // Inizializza un puntatore cur al primo nodo della lista, saltando il
                                    nodo sentinella.
  aux -> next = cur;             // Si collega il nuovo nodo aux al primo nodo attuale della lista,
                                    impostando il puntatore next di aux per puntare a cur.
  cur -> prev = aux;             // Si collega il primo nodo attuale (cur) al nuovo nodo aux, impostando
                                    il puntatore prev del nodo cur per puntare ad aux.
  aux -> prev = li;              // Il puntatore prev del nuovo nodo aux viene impostato per puntare
                                    al nodo sentinella li. Questo è necessario poiché aux diventa il
                                    primo nodo della lista e il nodo sentinella mantiene il collegamento
                                    all'inizio della lista.
  li -> next = aux;              // Si aggiorna il puntatore next del nodo sentinella li per farlo
                                    puntare al nuovo nodo aux, collegando così il nodo sentinella al
                                    nuovo primo nodo della lista.
}
```

removePos

```
void list::removePos(unsigned int pos,const List& li){
// Se pos è fuori dai limiti della lista (cioè se ci sono meno nodi di quelli richiesti), genera un'eccezione.
  if (pos >= size(li)) {
    string exc = "errore";
    throw exc;
  }
  unsigned int count = 0;                  // Inizializza una variabile count a zero, che verrà
                                              utilizzata per contare i nodi visitati mentre si attraversa la
                                              lista per trovare il nodo alla posizione pos.
  node* cur = li -> next;                  // Inizializza un puntatore cur al primo nodo della lista,
                                              saltando il nodo sentinella.
// Questo ciclo scorre i nodi della lista fino a raggiungere la posizione desiderata. Se cur punta di nuovo al
   nodo sentinella o se si raggiunge la posizione pos, il ciclo si ferma.
  while (cur != li && count < pos) {
    ++count;                               // Ogni volta che viene visitato un nodo, il contatore aumenta
                                              per tenere traccia della posizione corrente.
    cur = cur -> next;                     // cur viene spostato al nodo successivo, continuando il ciclo.
  }
1cur -> prev -> next = cur -> next;
2cur -> next -> prev = cur -> prev;
3delete cur;
}
```



removeEl

```
void list::removeEl(Elem el,const List& li){
  node* cur = li -> next;                  // Inizializza un puntatore cur al primo nodo della lista, saltando il
                                              nodo sentinella.
// Questo ciclo while scorre la lista finché cur non è uguale al nodo sentinella li, il che significa che si è
raggiunta la fine della lista.
  while (cur != li) {
    node* temp = cur -> next;              // Si salva un puntatore temporaneo temp che punta al nodo
                                              successivo a cur. Questo è importante perché se si decide di
                                              rimuovere il nodo cur, si perderà il riferimento al nodo successivo
                                              durante la rimozione. temp permette di continuare a scorrere.
// Viene controllato se il campo info del nodo corrente cur contiene l'elemento el che si desidera rimuovere. Se
sì, si procede alla rimozione del nodo.
  if (cur -> info == el) {
    1cur -> prev -> next = cur -> next;
    2cur -> next -> prev = cur -> prev;
    3delete cur;
    }
    cur = temp;                            // Dopo aver potenzialmente rimosso il nodo cur, si passa al nodo
                                              successivo (che era stato salvato in temp) per continuare
                                              l'iterazione sulla lista.
  }
}
```