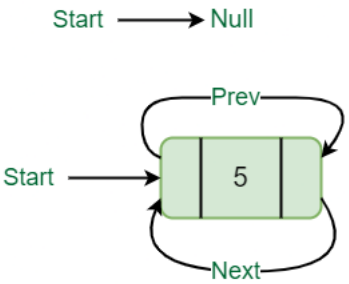
LAB1

N.B. Prima di spiegare ogni singola funzione tengo a precisare di osservare bene la struct di questa lista e di leggere con attenzione le richieste e soprattutto la mini spiegazione che si trova all'inizio del pdf di questo lab all'interno nella stessa cartella.

void list::createEmpty: Questa funzione mi permette di creare una lista vuota, essendo doppiamente collegata circolare e con sentinella non ci basta creare una nuova cella e impostarla a nullptr, ma dobbiamo creare una cella nuova e impostiamo i suoi campi next e prev che puntano alla cella stessa.

Codice:

```
node* aux = new node;
 aux -> next = aux;
 aux -> prev = aux;
 li = aux;
```



Nella figura non è raffigurata la sentinella, ma si troverebbe vicino a start

void list::clear: Questa funzione "smantella" la lista, rimuovendo tutti i nodi eccetto la sentinella.
Inizializzo cur a l -> next in quanto devo saltare la sentinella, creo un ciclo while che mi scorre tutta la lista senza tornare all'inizio (evito la sentinella).
Per eliminare ogni singolo elemento uso una variabile temp che viene aggiornata ad ogni ciclo, mi serve per poter eliminare ogni singolo elemento.
Alla fine di tutto (esco dal ciclo) imposto la lista che punta a se stessa, la lista viene considerata come vuota (ricordiamoci che c'è sempre la sentinella, quindi la situazione è come nella createEmpty)

Codice:

```
node* cur = li -> next;

while(cur != li){
node* temp = cur -> next;
delete cur;
cur = temp;
}

li -> next = li;
li -> prev = li;
```

bool list::isEmpty: Semplicemente quetsa funzione controlla se la lista è vuota, per farlo mi basta una if che controlla se il campo next della lista punta a se stessa

unsigned int list::size: Questa funzione si occupa di restituire la dimensione della mia lista, per farlo uso una variabile count.
Scorro l'intera lista e incremento il mio count, ragionamento analogo con le liste semplici... niente di complicato.
Codice:

```
node* cur = li -> next;

unsigned int count = 0;

 while (cur != li){
  cur = cur -> next;
 ++count;

 return count;
```

Elem list::get: Restituisce l'elemento in posizione pos, se pos non è corretta allora solleviamo una eccezione di tipo string.
Semplicemente impostiamo una variabile count, scorriamo la lista finche non raggiunge la posizione e restituiamo l'elemento.

Codice:

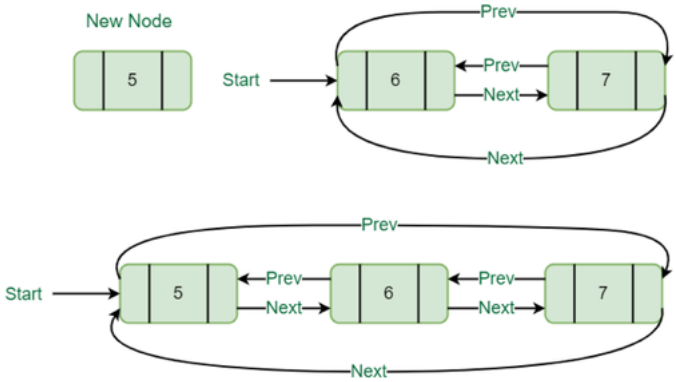
```
if (pos >= size(li)) {
 string err = "get: indice invalido";
 throw err;
}

 node* cur = li -> next;
  unsigned int count = 0;
  while (cur != li && count < pos) {
```

```
 cur = cur -> next;
 ++count;
}
return cur -> info;
```

void list::set: Modifica l'elemento in posizione pos.Il funzionamento è simile alla get ma come unica differenza Andiamo a sovrascrivvere l'elemento i-esimo con cur -> info = el;

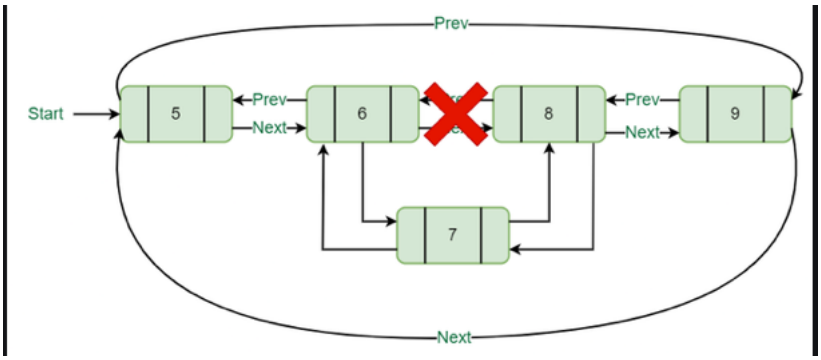
void list::add: Inserisco l'elemento in posizione pos, la differenza della set è che devo shiftare tutti gli altri elementi a destra.
Si verificano diversi casi, partiamo da quello in **testa.**
Quello che succede è la seguente situazione:



Inserimento in testa

Per fare l'inserimento in testa creiamo il nuovo nodo, io l'ho chiamato aux.
Aux -> next deve puntare a li -> next e non a li (ricordiamoci che li sarebbe la sentinella).
Ora essendo la nuova testa ovviamente l'elemento next (quindi li -> next) si deve collegare "indietro" al nuovo nodo, per farlo scriviamo semplicemente li -> next -> prev = aux.
colleghiamo il prev di aux a li essendo la sentinella e ora li -> next deve collegarsi ad aux in quanto essendo circolare ora li -> next deve puntare alla nuova testa che ora è aux.

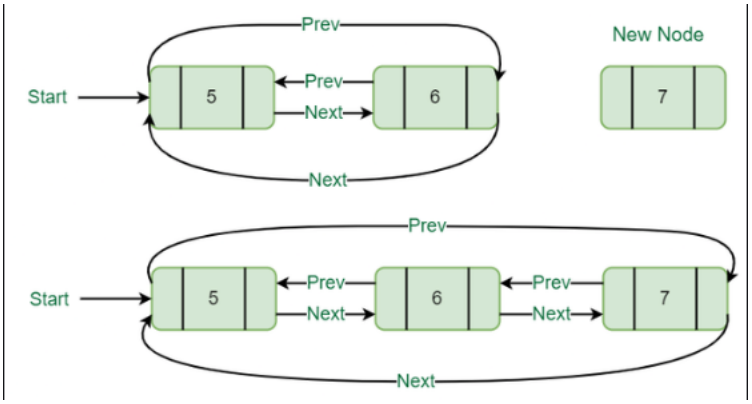
Ora tutti gli altri casi (in realtà uno solo):



Inserimento in mezzo a due nodi

Come si vede da immagine dobbiamo aggiornare i campi prev e next di cur.
(Cur -> prev) -> next = aux, aux -> prev = cur -> prev, aux -> next = cur, cur -> prev = aux.
Sembra complicato ma il segreto è farsi I disegni, se aux devo inserirlo tra due nodi è abbastanza logico che devo collegare il nodo precedente (cur -> prev) e nodo successivo ad aux.

void list::addRear: questo rappresenta il caso di inserimento in coda.



tailInsert

Come sempre creiamo il nostro nuovo nodo aux, usiamo inoltre un nodo cur inizializzato a cur -> prev. Questo perche ricordiamoci che la nostra sentinella sarebbe li, quindi li -> prev indica il suo elemento precedente (ultimo elemento nella lista, nella figura sarebbe il 6).
Come prima fate attenzione ai disegni, la implementazione è la seguente:

```
node* aux = new node;
aux -> info = el;
```

```
node* cur = li -> prev;

cur -> next = aux;
aux -> prev = cur;
aux -> next = li;
li -> prev = aux;
```

void list::addFront: Inserimento in testa. Questa funzione fa semplicemente il caso in testa che abbiamo visto nella funzione void::list add, codice identico.

void list::removePos: cancello l'elemento in posizione pos.
L'unica difficoltà in questa funzione è quella di coprire ogni singolo caso, partiamo da quello base.
L'elemento da eliminare è la nostra testa.
Rimuovo il primo nodo dopo la sentinella, aggiorno i puntatori e dealloco il nodo rimosso.
Uso un nodo cur che punta a li -> next, aggiorno i cambi prev e next di cur e procedo con l'eliminazione:

```
node* cur = li -> next;
cur -> prev -> next = cur -> next;
cur -> next -> prev = cur -> prev;
delete cur;
return;
```

Per gli altri casi aggiorno sempre i campi prev e next di cur.
Nella mia implementazione ho considerato solo il caso in mezzo in quanto abbiamo una funzione già apposita per la rimozione in coda, ma potete comunque benissimo aggiungere una if impostandola a :

```
if (cur == li -> prev) .
```

void list::removeEl: cancella tutte le occorenze dell'elemento elem.
Impostiamo cur a li -> next ed entriamo nel ciclo.

Prima di fare qualsiasi operazione con cur, salviamo il nodo successivo in un puntatore temporaneo temp. Questo è importante perché potremmo deallocare cur durante il ciclo, e quindi abbiamo bisogno di sapere quale nodo considerare successivamente.

Usiamo una if per controllare se cur -> info == el :

```
if (cur -> info == el) {
  cur -> next -> prev = cur -> prev;
  cur -> prev -> next = cur -> next;
  delete cur;
}
```

Se l'elemento del nodo corrente (`cur -> info`) è uguale all'elemento che vogliamo rimuovere (`el`), facciamo quanto segue:

1. Aggiornamento dei Puntatori:
   - `cur -> next -> prev = cur -> prev;`: Questo aggiorna il puntatore `prev` del nodo successivo a `cur` per puntare al nodo precedente a `cur`.
   - `cur -> prev -> next = cur -> next;`: Questo aggiorna il puntatore `next` del nodo precedente a `cur` per puntare al nodo successivo a `cur`.
2. Deallocazione del Nodo:
   - `delete cur;`: Questo dealloca il nodo corrente, rimuovendolo dalla memoria.

Alla fine del ciclo, avanziamo `cur` al nodo successivo (`temp`), che abbiamo salvato prima nel ciclo.