Listas

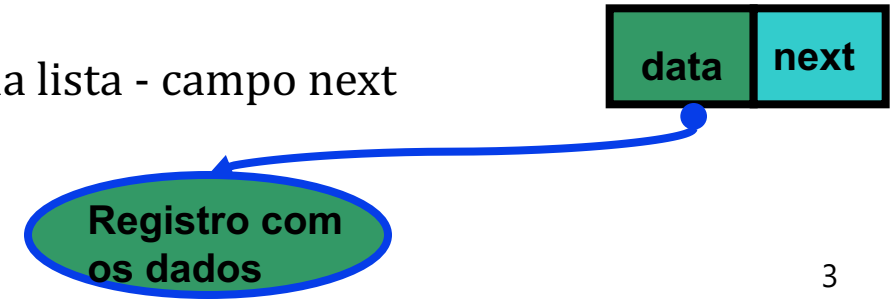
Estrutura de Dados
Prof. Anselmo C. de Paiva
Departamento de Informática – Núcleo de Computação Aplicada NCA-UFMA

Limitações dos vetores

- Vetores
  - Simples,
  - Rápidos
- Mas,
  - é necessário especificar o tamanho no momento da construção
  - Lei de Murphy:
    - Construa um vetor com espaço para n elementos e você sempre chegará à situação em que necessita de n+1 elementos

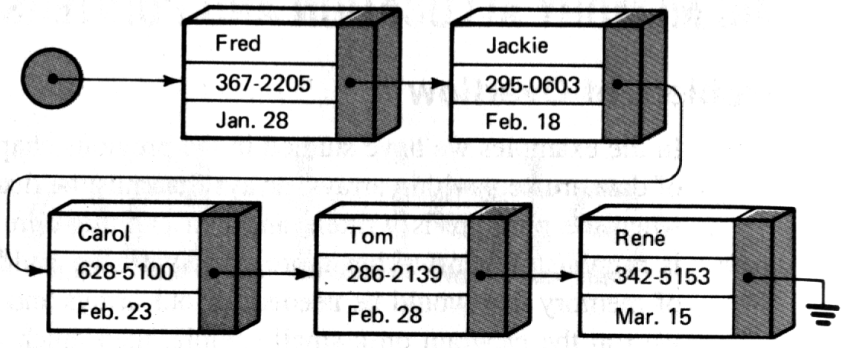
Listas Encadeadas

- Utilização flexível da memória
  - memória é alocada dinamicamente para cada elemento a medida que for necessária
  - cada elemento da lista inclui um ponteiro para o próximo
- Lista Encadeada
  - Cada item(tipo node) da lista contém:
    - o item de dado (ponteiro para a estrutura que realmente guarda os dados) - campo data
    - um ponteiro pra o próximo item da lista - campo next



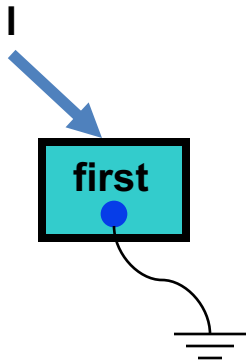
Listas Encadeadas

- Vantagens e desvantagens
  - ↑ Alocação de memória sob demanda
  - ↑ Facilidade de manutenção
  - ↓ Memória extra
  - ↓ Acesso seqüencial
- Propriedades
  - a lista pode ter 0 ou infinitos itens
  - um novo item pode ser incluído em qualquer ponto
  - qq item pode ser removido
  - qq item pode ser acessado
  - permite muitas operações



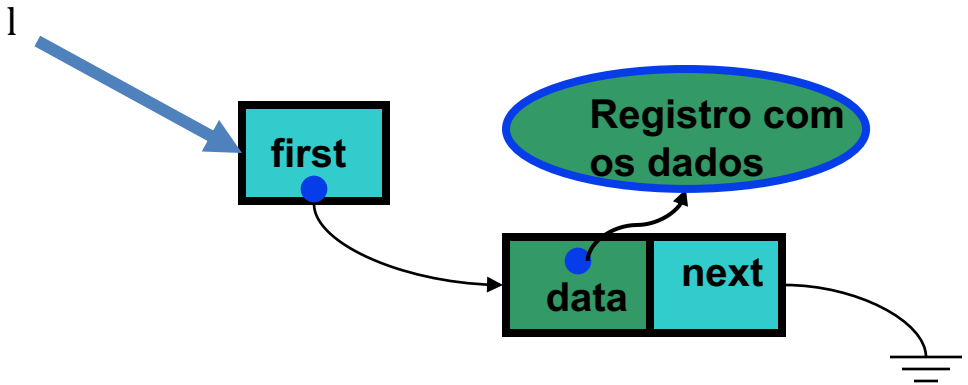
Listas Encadeadas

- A estrutura Coleção possui em substituição ao vetor de ponteiros para dados um ponteiro para a lista head
  - Inicializado com NULL



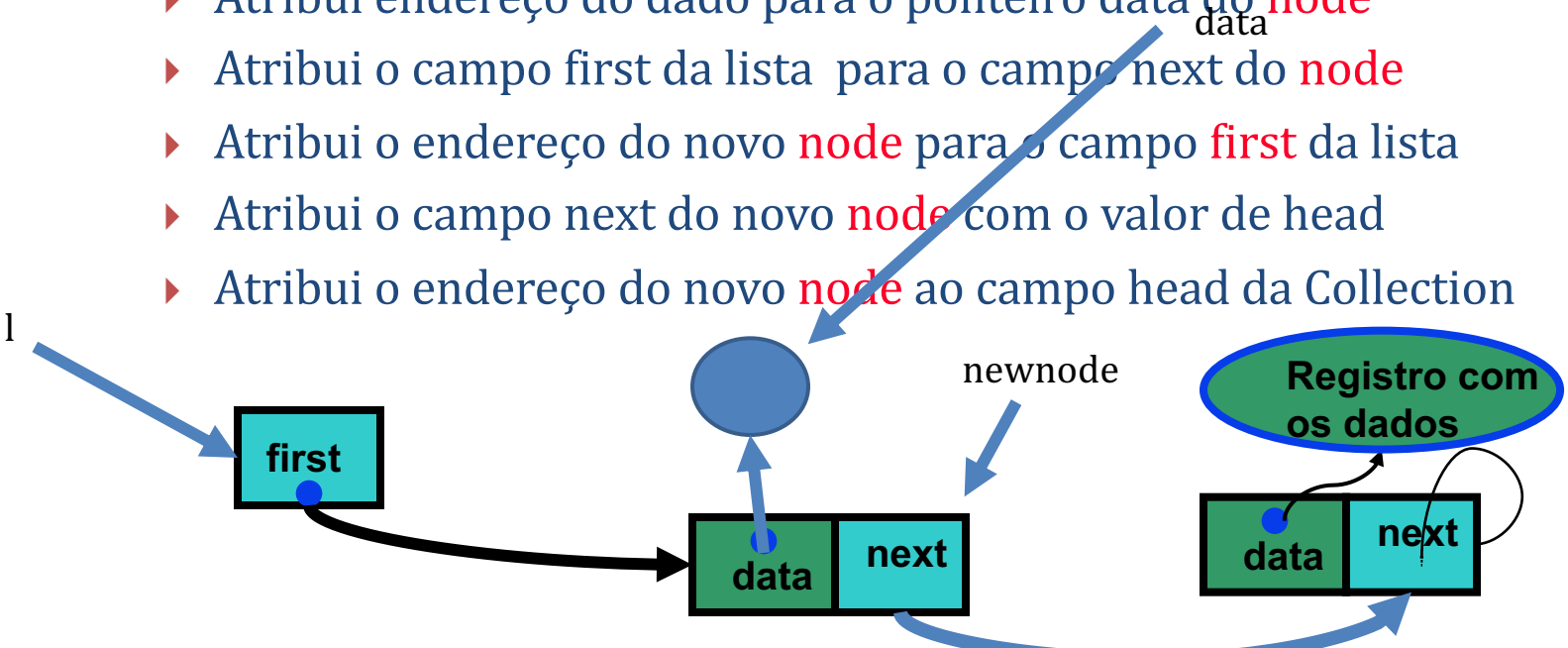
Listas Encadeadas

- Insere o primeiro elemento na lista
  - Aloca espaço para um *node*
  - Atribui endereço do dado para o ponteiro data do node
  - Atribui NULL para o campo next do node
  - Atribui o endereço do novo node para o campo FIRST da Coleção

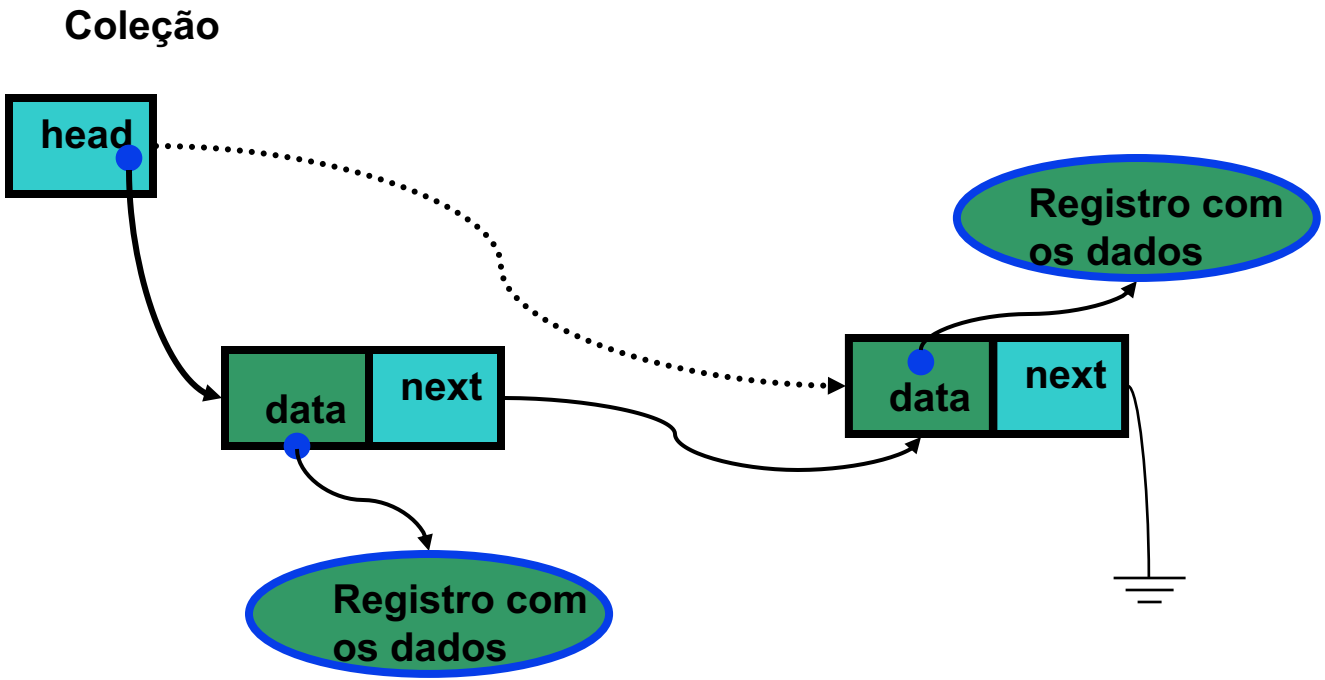


Listas Encadeadas

- Insere o segundo item
  - Aloca espaço para um node
  - Atribui endereço do dado para o ponteiro data do node
  - Atribui o campo first da lista para o campo next do node
  - Atribui o endereço do novo node para o campo first da lista
  - Atribui o campo next do novo node com o valor de head
  - Atribui o endereço do novo node ao campo head da Collection



Listas Encadeadas



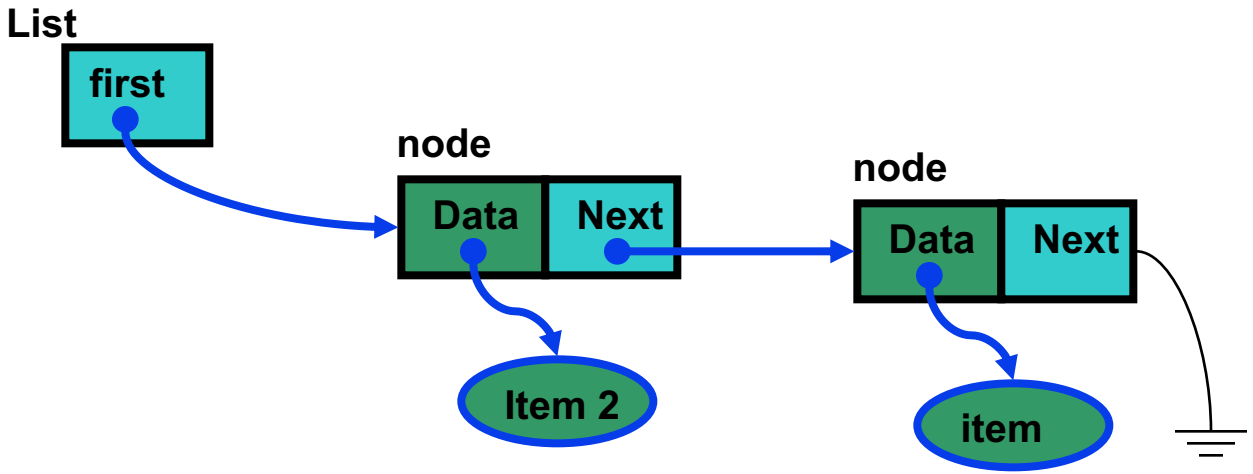
Listas - Implementação operação ADD

```
struct _slnode_ {
  void *data;
  struct _slnode_ *next;
} SLNode;
typedef struct _collection_ {
 SLNode *first;
 SLNode *cur;
}SLList;

SLList *sllCreate ( ) {
   SLList *l;
   l= (SLList *) malloc ( sizeof(SLList));
   if ( l != NULL ) {
      l->first = NULL;
      return l;
   }
   return NULL;
```

```
int sllDestroy (SLList *l) {
    if (l != NULL ) {
      if ( l->first == NULL ) {
          free (l);
          return TRUE;
      }
    }
    return FALSE;
}
```

Listas Ligadas

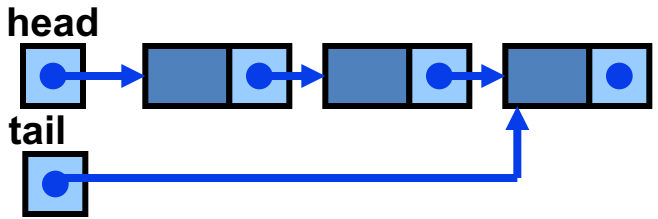
- Tempo para inserção
  - Constante - independente do número de elementos na lista n
- Tempo de Consulta
  - Pior caso - n



Pilhas e Filas como Listas Ligadas

- Pilha - Implementação mais simples
  - Adiciona e retira somente no início da lista
  - Last-In-First-Out (LIFO) semantics
- Fila
  - First-In-First-Out (FIFO)
  - Mantém um ponteiro para o final da lista

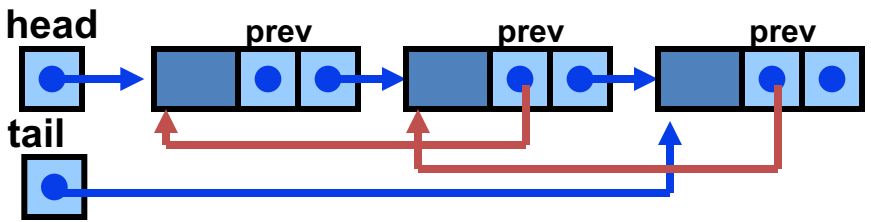
```
struct _slnode_ {
    void *data;
    struct _slNode_ *next;
} SLNode;
struct _lqueue_ {
 SLNode *head, *tail;
}LQueue;
```



Listas - Duplamente encadeadas

- Podem ser percorridas nas duas direções

```
struct _dlnode_ {
  void *item;
  struct _dlnode_ *prev,
*next;
}Node;
struct _DLList_ {
    DLNode *first;
}DLList;
```



Pilha

- sllCreate. stkCreate
- sllDestroy stkDestroy
- sllInsertFirst. stkPush
- sllRemoveFirst. stkPop

Fila

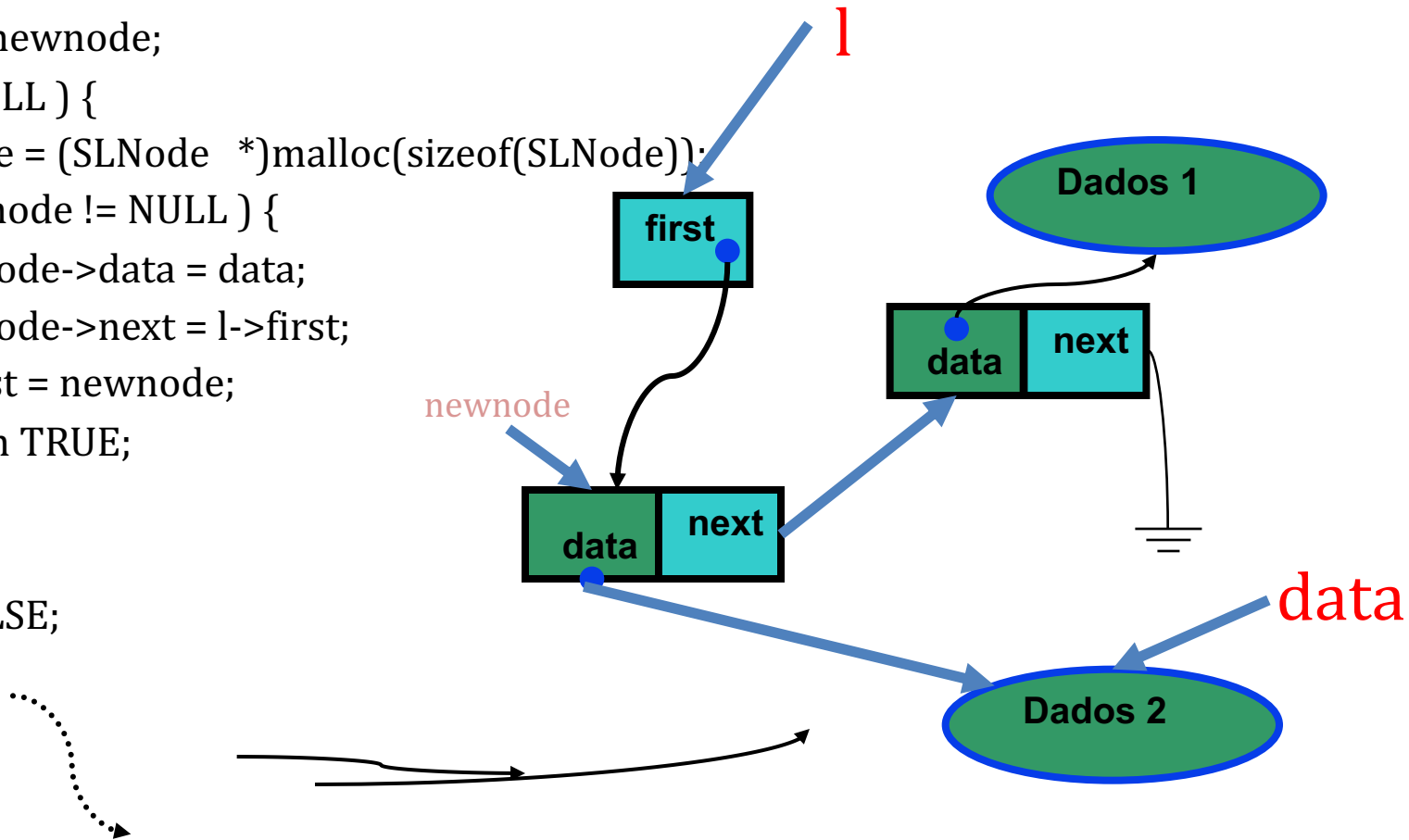
- sllCreate. qCreate
- sllDestroy qDestroy
- sllInsertLast qEnqueue
- sllRemoveFirst. qDequeue

```
int sllInsertFirst ( Sllist *l, void *data)
 {
    SLNode *newnode;
    if ( l != NULL ) {
      newnode = (SLNode   *)malloc(sizeof(SLNode));
      if ( newnode != NULL ) {
          newnode->data = data;
          newnode->next = l->first;
          l->first = newnode;
          return TRUE;
      }
   }
   return FALSE;
}
```

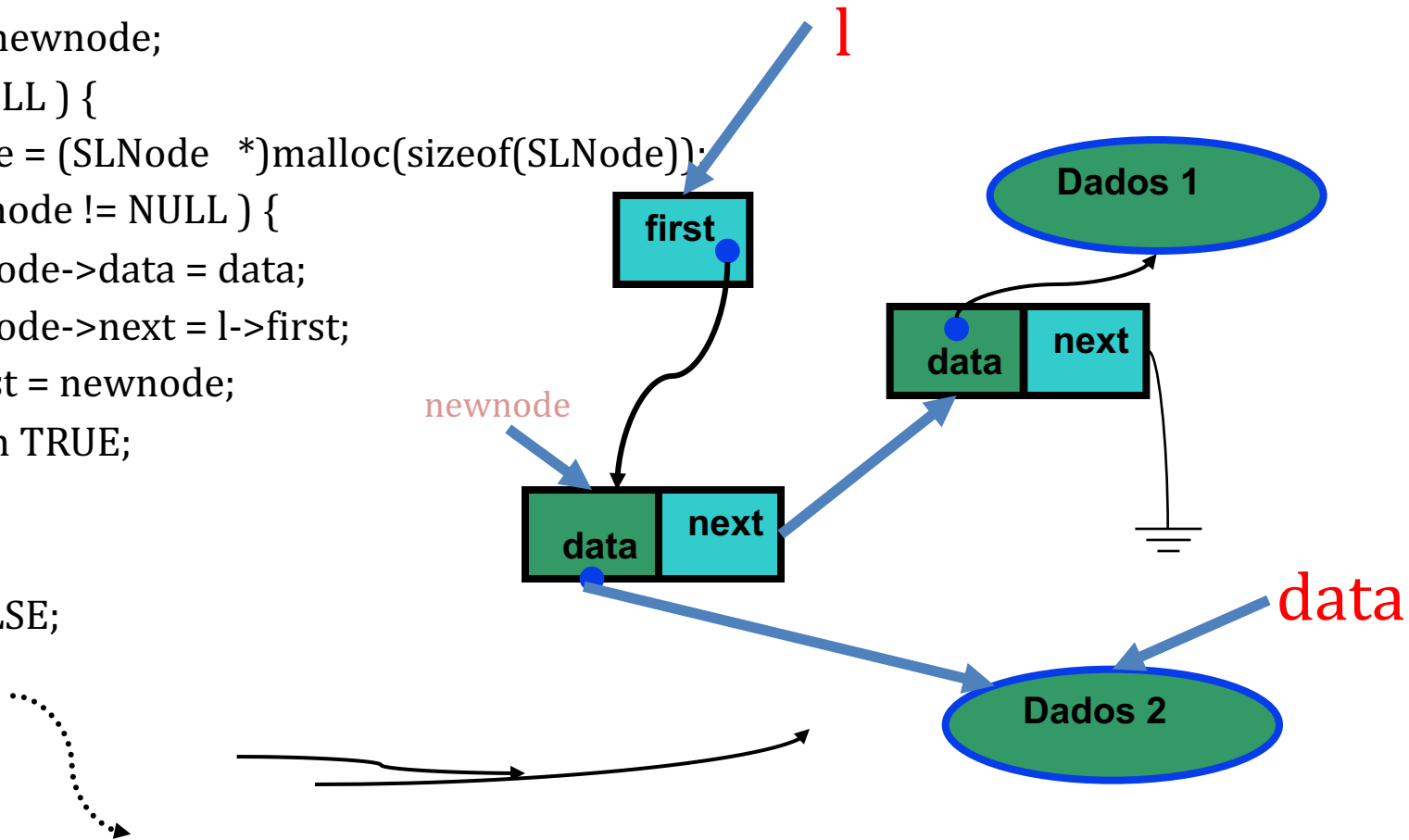
Insere o primeiro elemento na lista

- Aloca espaço para um *node*
- Atribui endereço do dado para o ponteiro data do node
- Atribui o campo first para o campo next do node
- Atribui o endereço do novo node para o campo FIRST da Coleção

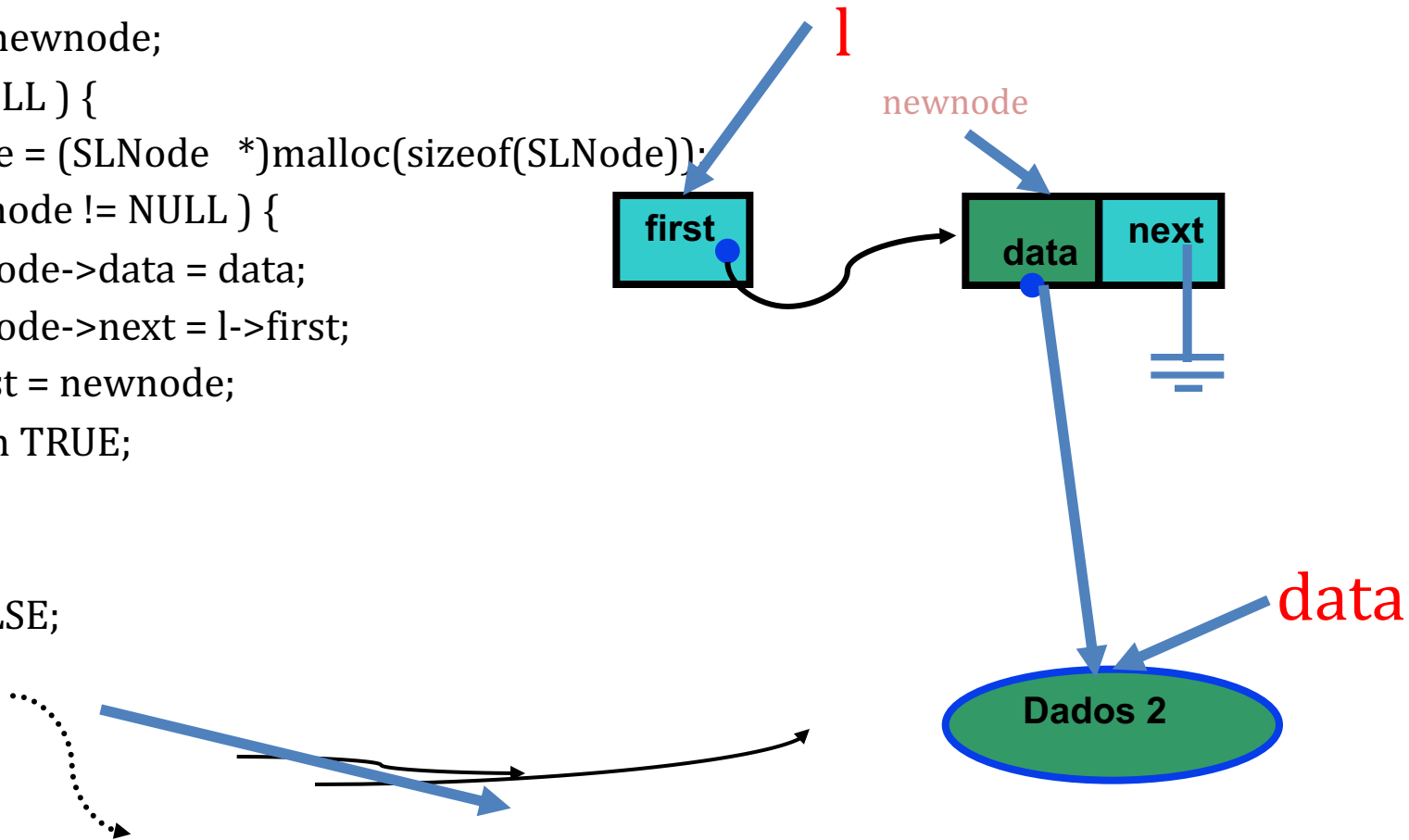
```
int sllInsertFirst ( Sllist *l, void *data)
 {
    SLNode *newnode;
    if ( l != NULL ) {
      newnode = (SLNode   *)malloc(sizeof(SLNode));
      if ( newnode != NULL ) {
          newnode->data = data;
          newnode->next = l->first;
          l->first = newnode;
          return TRUE;
      }
   }
   return FALSE;
}
```



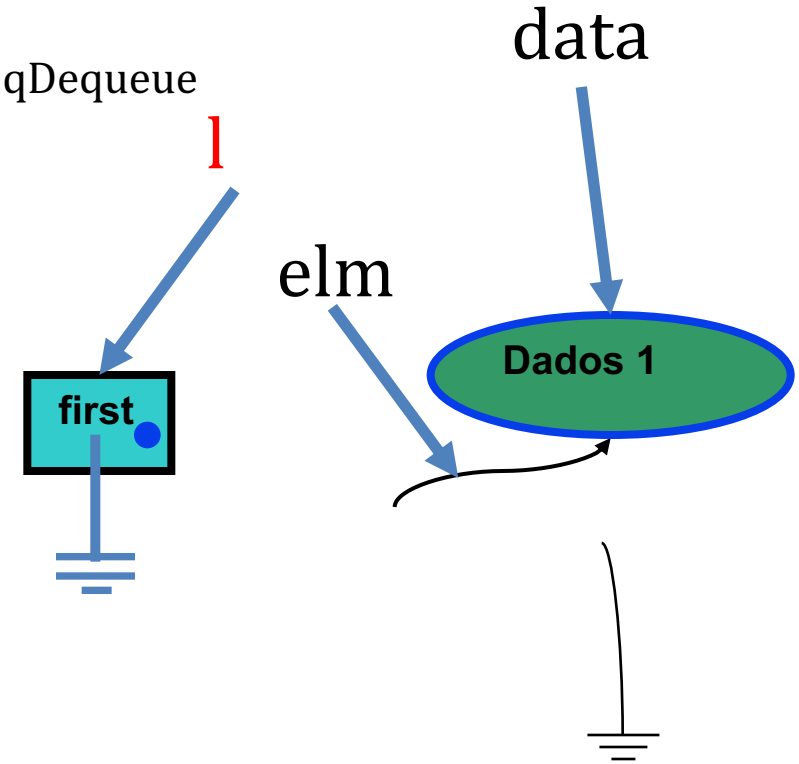
```
int sllInsertFirst ( Sllist *l, void *data)
 {
    SLNode *newnode;
    if ( l != NULL ) {
      newnode = (SLNode   *)malloc(sizeof(SLNode));
      if ( newnode != NULL ) {
          newnode->data = data;
          newnode->next = l->first;
          l->first = newnode;
          return TRUE;
      }
   }
   return FALSE;
}
```



```
int sllInsertFirst ( Sllist *l, void *data) // stkPush
 {
    SLNode *newnode;
    if ( l != NULL ) {
      newnode = (SLNode   *)malloc(sizeof(SLNode));
      if ( newnode != NULL ) {
          newnode->data = data;
          newnode->next = l->first;
          l->first = newnode;
          return TRUE;
      }
   }
   return FALSE;
}
```

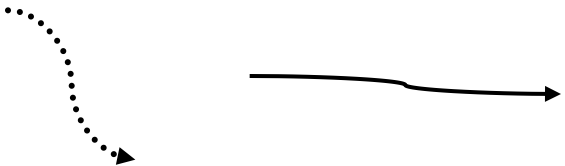


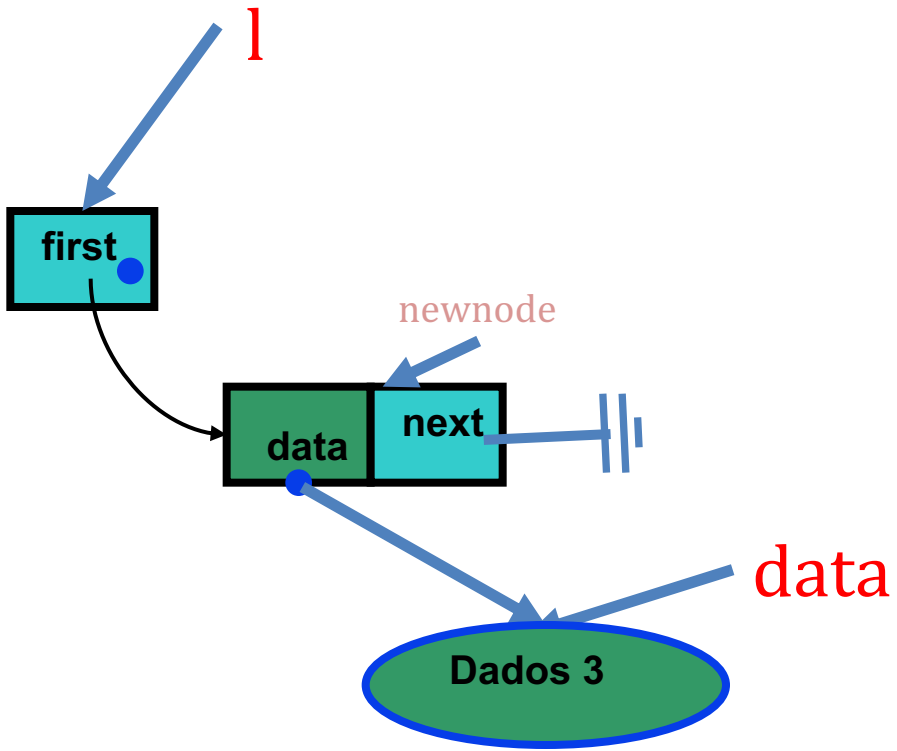
```
void *sllRemoveFirst ( Sllist *l)  // stk Pop. Ou qDequeue
 {
    void * data;
     SLNode elm;
    if ( l != NULL ) {
        if ( l->first != NULL ) {
            elm = l->first;
            data = elm->data;
            l->first = elm->next;
            free(elm);
            return data;
        }
    }
    return NULL;
}
```



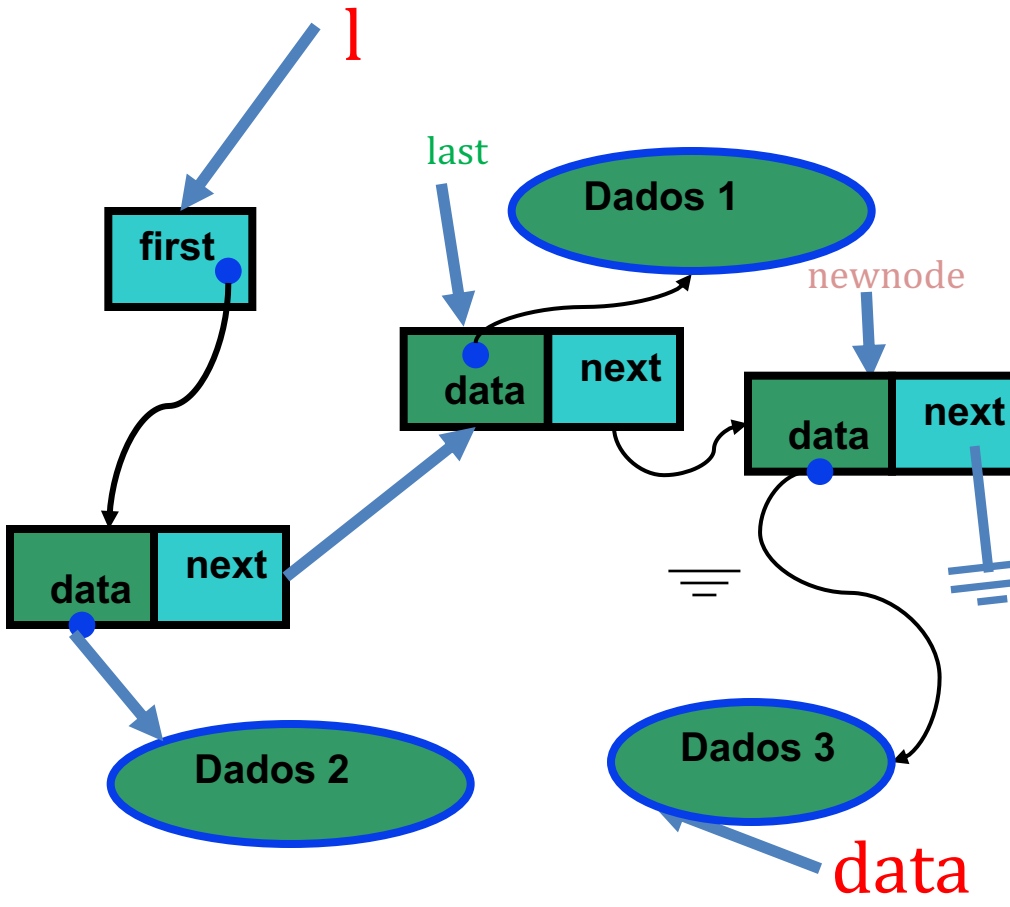
```
int sllInsertLast ( Sllist *l, void *data)
 {
    SLNode *newnode;
    if ( l != NULL ) {
      if ( l->first == NULL ) {
         if (l->first == NULL ) {
            newnode = (SLNode   *)malloc(sizeof(SLNode));
             if ( newnode != NULL ) {
                  newnode->data = data;
                  newnode->next = NULL;
                 l->first = newnode
              }
          } else{

         }
```

```
}
```





```
int sllInsertLast ( Sllist *l, void *data)
{
    SLNode *newnode;  SLNode * last;
    if ( l != NULL ) {
        newnode = (SLNode   *)malloc(sizeof(SLNode));
        if ( newnode != NULL ) {
            newnode->data = data;
            newnode->next = NULL;
            if (l->first == NULL ) {
                l->first = newnode;
            } else{
                last = l->first;
                while (last->next != NULL) {
                    last = last->next;
                }
                last->next = newnode;
            }
            return TRUE
        }
    }
  }
  return FALSE
}
```

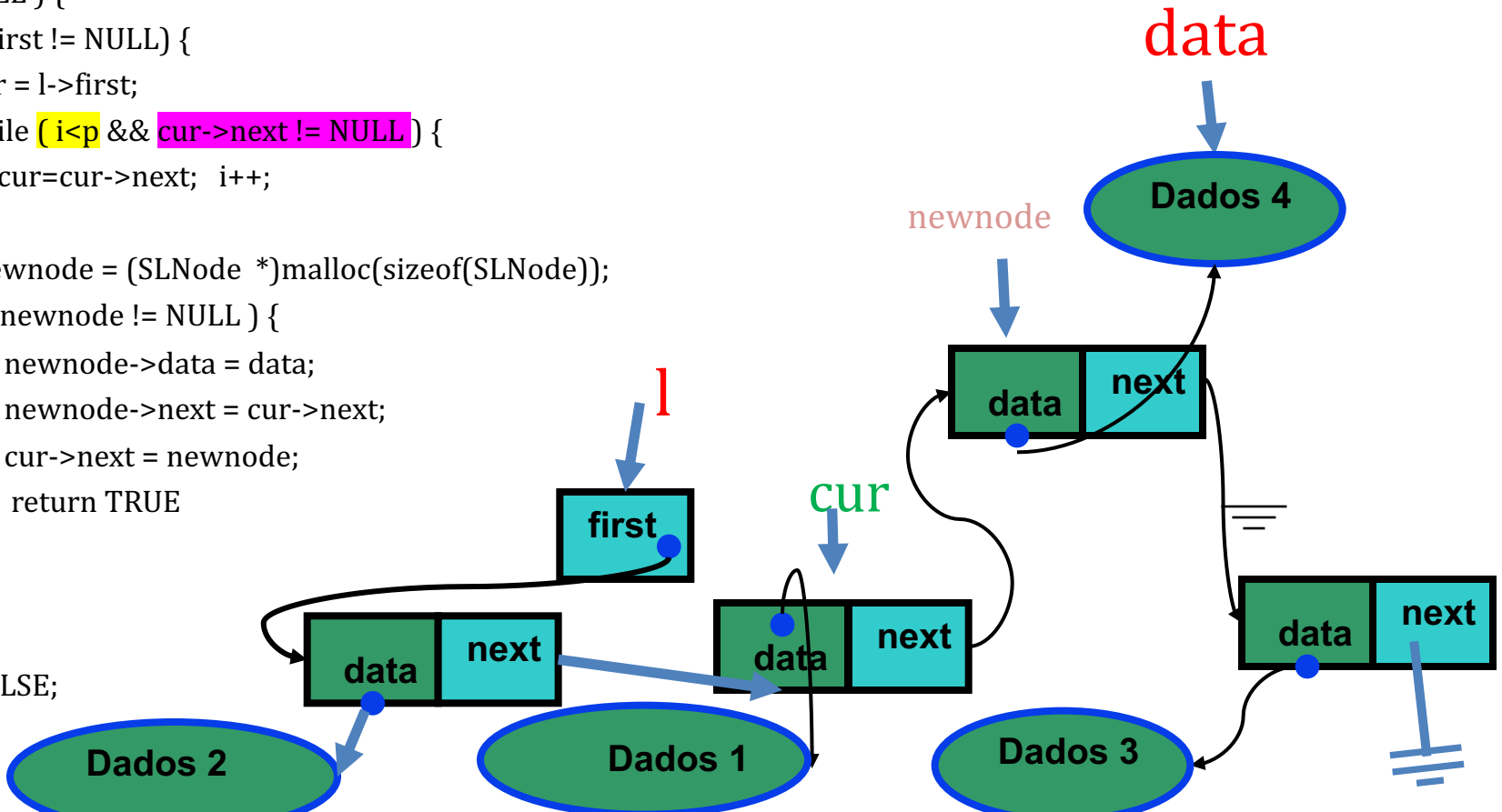


```
int sllNumNodes ( SLList *l)
{
      int n=0;
     if ( l != NULL ) {
        if ( l->first != NULL) {
            node = l->first;
             n++;
             while ( node->next != NULL){
                  node = node->next;
                  n++;
              }
               return n;
        }
         return 0;
     }
     return -1;
}
```

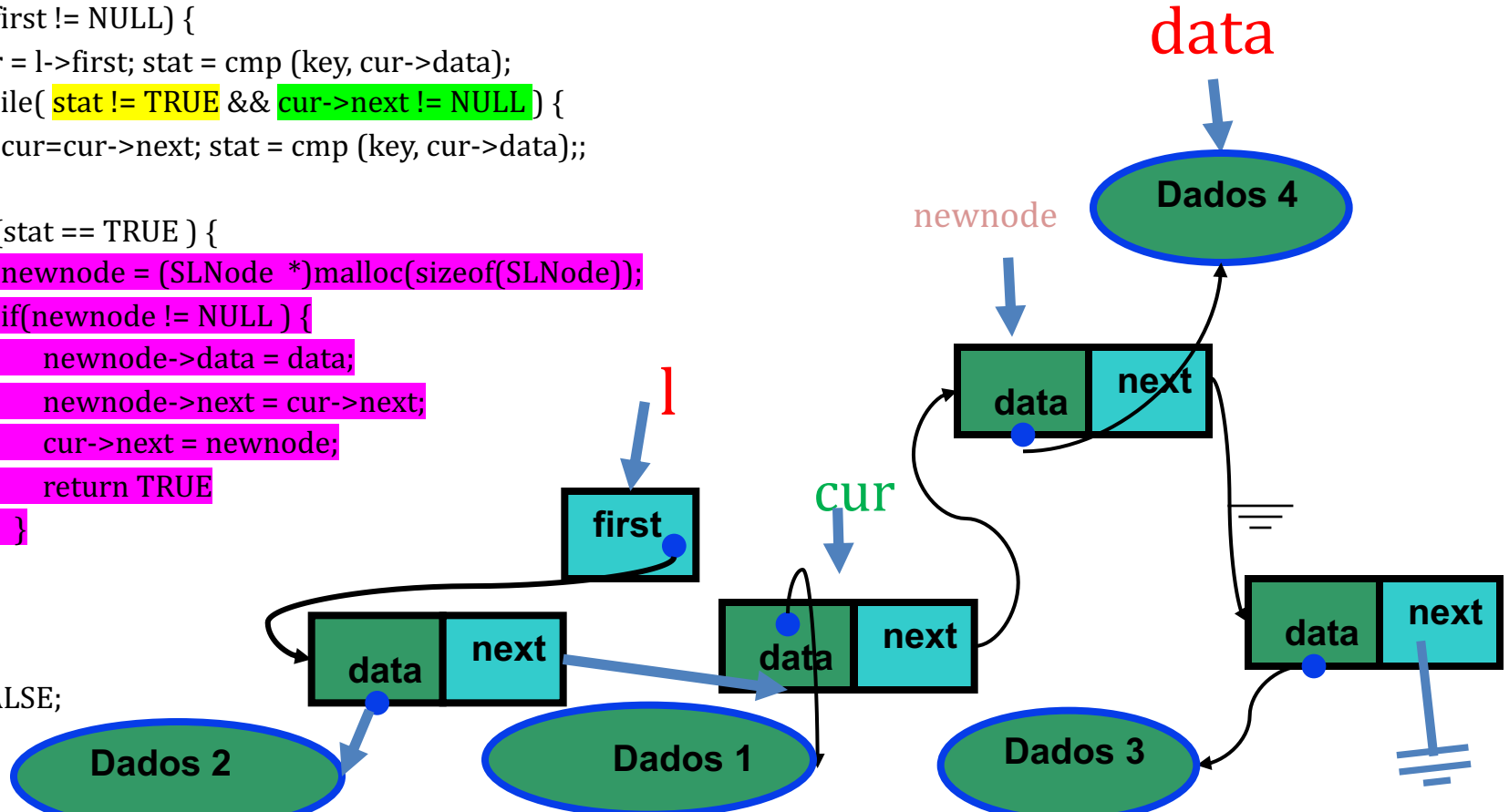
Operações sobre Listas Ligadas

- Criar lista vazia
- Inserir primeiro elemento
- Inserir no início de uma lista
- Remover primeiro elemento
- Inserir no final da lista
- Tamanho da lista
- Inserir um novo elemento na posição p
- Inserir após um elemento identificado pela chave
- Inserir um novo elemento antes da posição p
- Inserir antes um elemento identificado pela chave
- Remover o p-esimo elemento
- Remover um elemento identificado pela chave
- Remover após um elemento identificado pela chave
- Remover antes um elemento identificado pela chave

```
int sllInsertAfterPesimo( Sllist *l, void *data, int p)
 {
    SLNode *newnode, *cur;  SLNode * last; int i=0;
    if ( l != NULL ) {
        if ( l->first != NULL) {
             cur = l->first;
            while ( i<p && cur->next != NULL ) {
                cur=cur->next;   i++;
            }
             newnode = (SLNode  *)malloc(sizeof(SLNode));
             if(newnode != NULL ) {
                 newnode->data = data;
                 newnode->next = cur->next;
                 cur->next = newnode;
                  return TRUE
              }
        }
    }
     return FALSE;
}
```

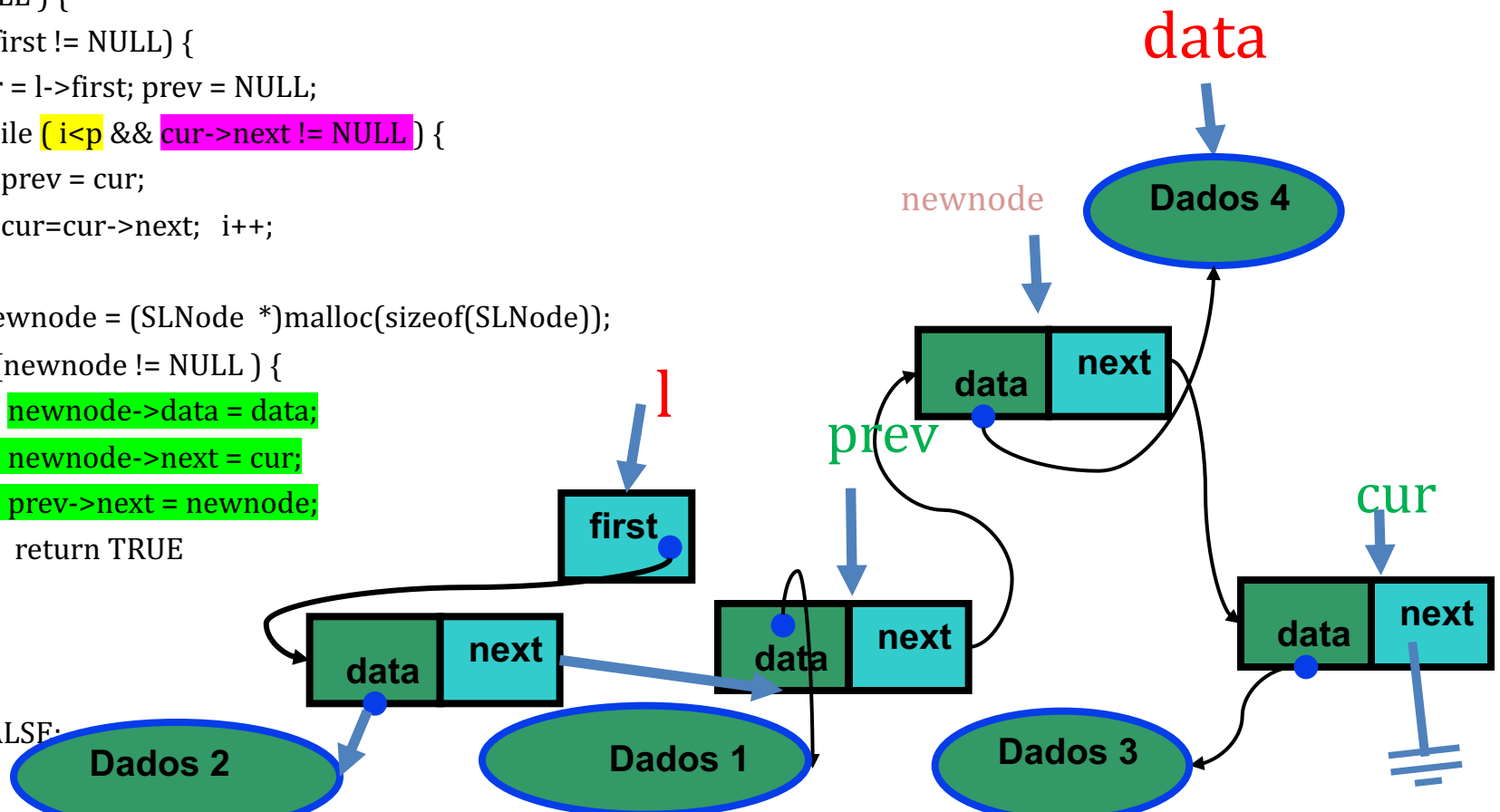


```
int sllInsertAfterSpec( Sllist *l, void *data, void *key, int (*cmp)(void *, void *))
 {
    SLNode *newnode, *cur; int i=0;
    if ( l != NULL ) {
        if ( l->first != NULL) {
            cur = l->first; stat = cmp (key, cur->data);
            while( stat != TRUE && cur->next != NULL ) {
                cur=cur->next; stat = cmp (key, cur->data);;
             }
             if (stat == TRUE ) {
                newnode = (SLNode  *)malloc(sizeof(SLNode));
                if(newnode != NULL ) {
                    newnode->data = data;
                    newnode->next = cur->next;
                    cur->next = newnode;
                    return TRUE
                }
             }
        }
    }
     return FALSE;
}
```

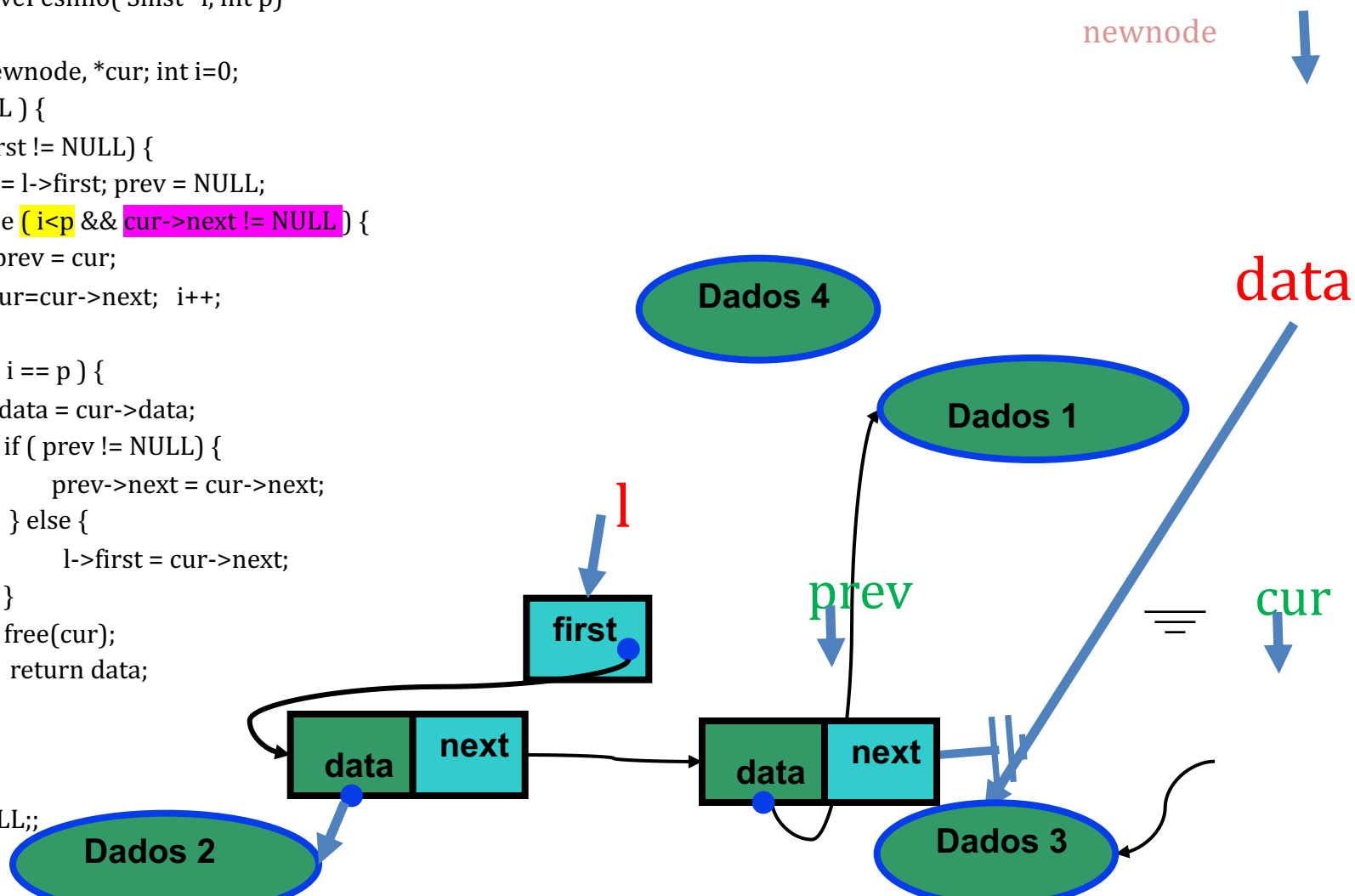


```
int sllInsertBeforePesimo( Sllist *l, void *data, int p)
 {
    SLNode *newnode, *cur, *prev; int i=0;
    if ( l != NULL ) {
        if ( l->first != NULL) {
            cur = l->first; prev = NULL;
            while ( i<p && cur->next != NULL ) {
                prev = cur;
                cur=cur->next;   i++;
             }
             newnode = (SLNode  *)malloc(sizeof(SLNode));
             if(newnode != NULL ) {
                newnode->data = data;
                newnode->next = cur;
                prev->next = newnode;
                 return TRUE
             }
        }
    }
     return FALSE;
}
```

Ta errada ver a correção no slide la do final



```
Void * sllRemovePesimo( Sllist *l, int p)
 {
    SLNode *newnode, *cur; int i=0;
    if ( l != NULL ) {
        if ( l->first != NULL) {
            cur = l->first; prev = NULL;
           while ( i<p && cur->next != NULL ) {
                prev = cur;
               cur=cur->next;   i++;
           }
            if ( i == p ) {
                data = cur->data;
                if ( prev != NULL) {
                      prev->next = cur->next;
                } else {
                       l->first = cur->next;
                }
                free(cur);
                return data;
            }
        }
    }
    return NULL;;
}
```



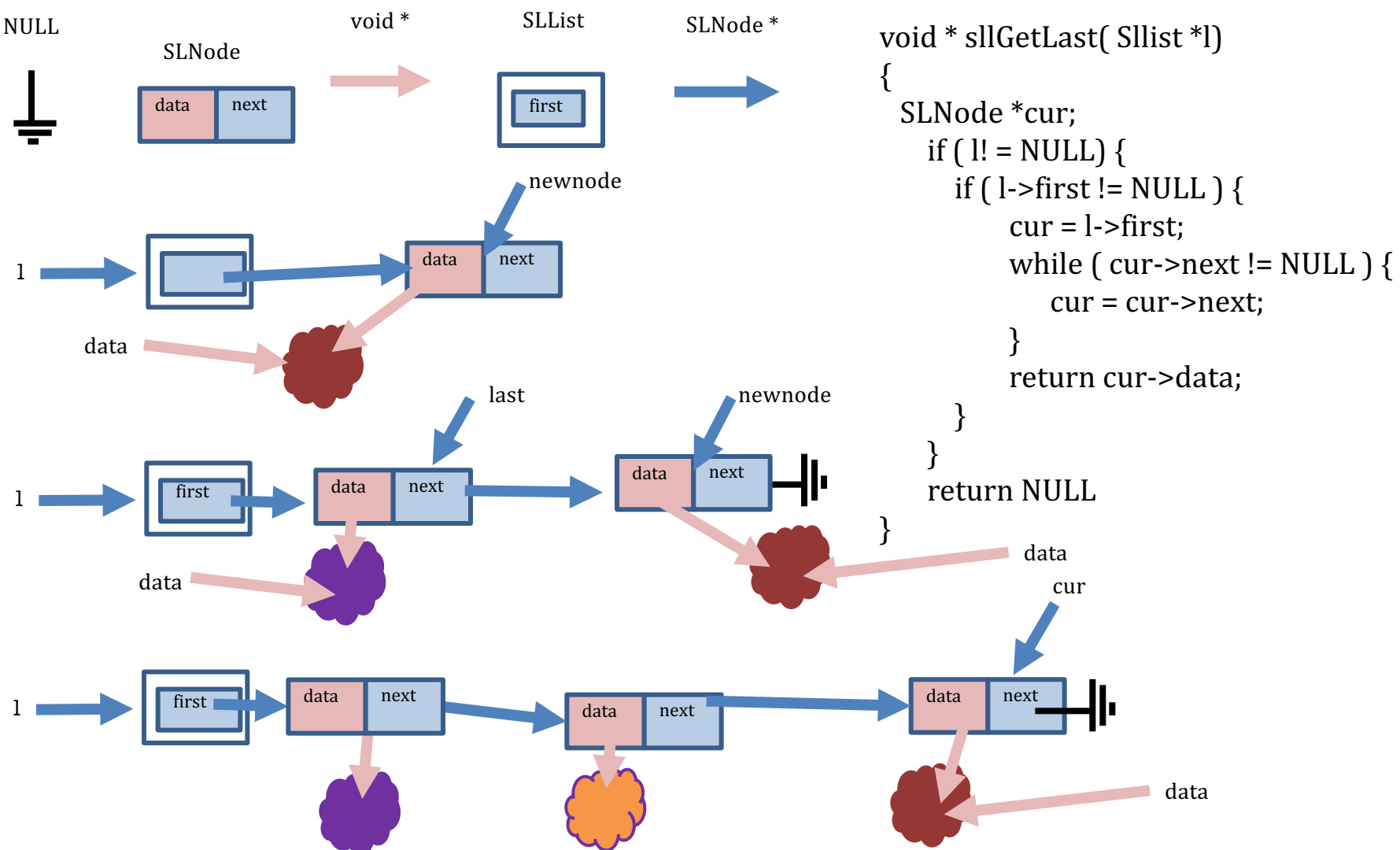
```
// SLList.h
typedef struct _SLNode_ {
        struct _SLNode_ *next;
        void *data;
} SLNode;

Typedef struct _SLList_ {
        SLNode *first;
} SLList;
```

```
void *sllGetFirst ( SLList *l)
{
  SLNode *aux;
  void * data;
  if ( l != NULL ){
     if ( l->first !-= NULL ) {
       aux =  l-> first;
       data = aux->data;
        return data;
     }
  }
   return  NULL;
}
}
```

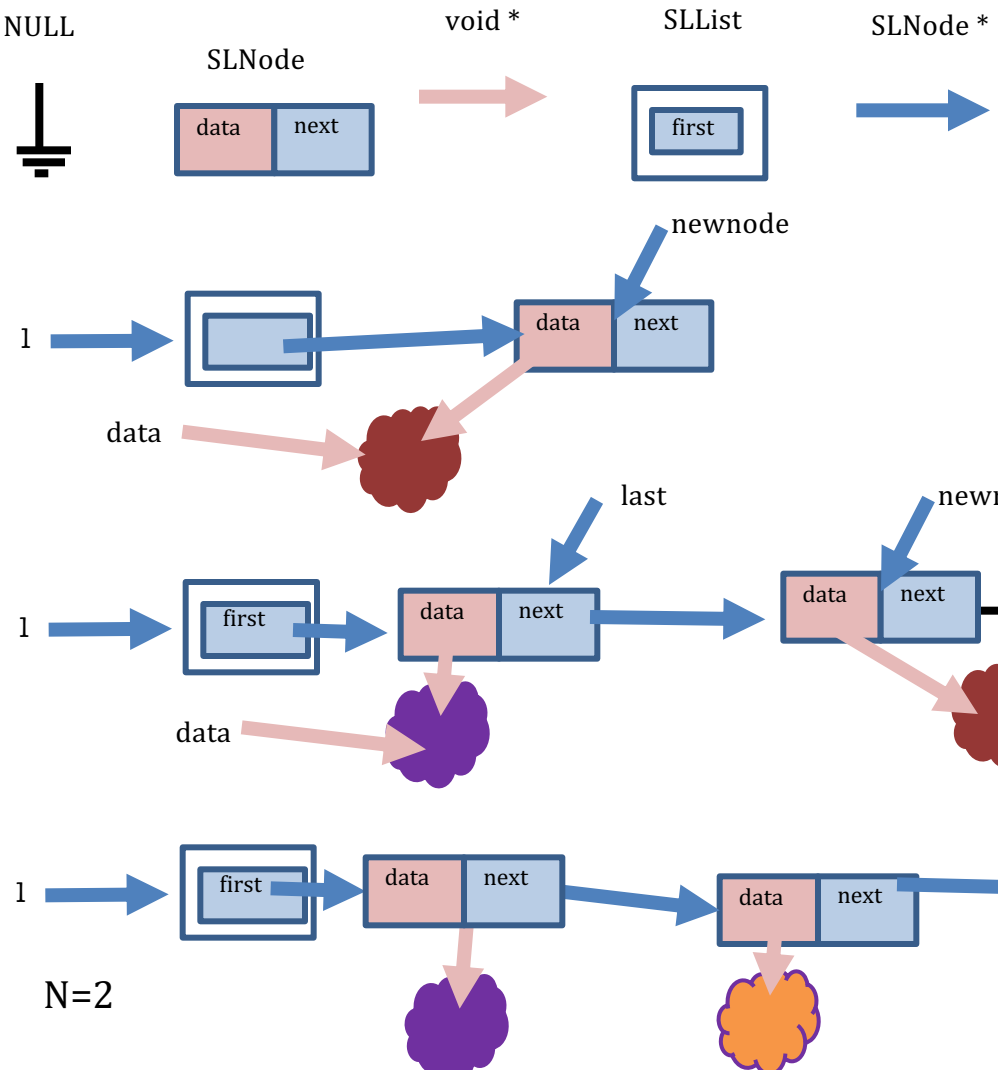
```
//SLLStack.c
SLLIst  *sllCreate(   )
{
     SLList *l;
     l = (SLList     *) malloc ( sizeof (SLList ));
     if ( l != NULL ) {
              l->first  = NULL;
              return l;
     }
     return NULL;
}

int sllInsertFirst ( SLList *l, void *elm). //slstkPush
{
    SLNode *newnode;
    if ( l != NULL ){
         newnode = (SLNode *)malloc(sizeof(SLNode));
        if (newnode != NULL ) {
           newnode->data = elm;
            if ( l->first  == NULL ) {
               newnode->next = NULL;
            } else {
                newnode->next = l->first;
             }
            l->first  = newnode;
            return TRUE;
         }
     }
     return FALSE;
}
```

```
void *sllRemoveFirst ( SLList *l)  // slstkPop
{
  SLNode *aux;
  void * data;
  if ( l != NULL ){
     if ( l->first !-= NULL ) {
       aux =  l-> first;
       data = aux->data;
       l->first = aux->next;
       free(aux);
       return data;
     }
  }
   return  NULL;
}

Int sllDestroy (SLList *l)
{
    if ( l ! =NULL) {
       if ( l->first == NULL) {
         free(l);
         return TRUE;
       }
    }
    return FALSE;
}
```

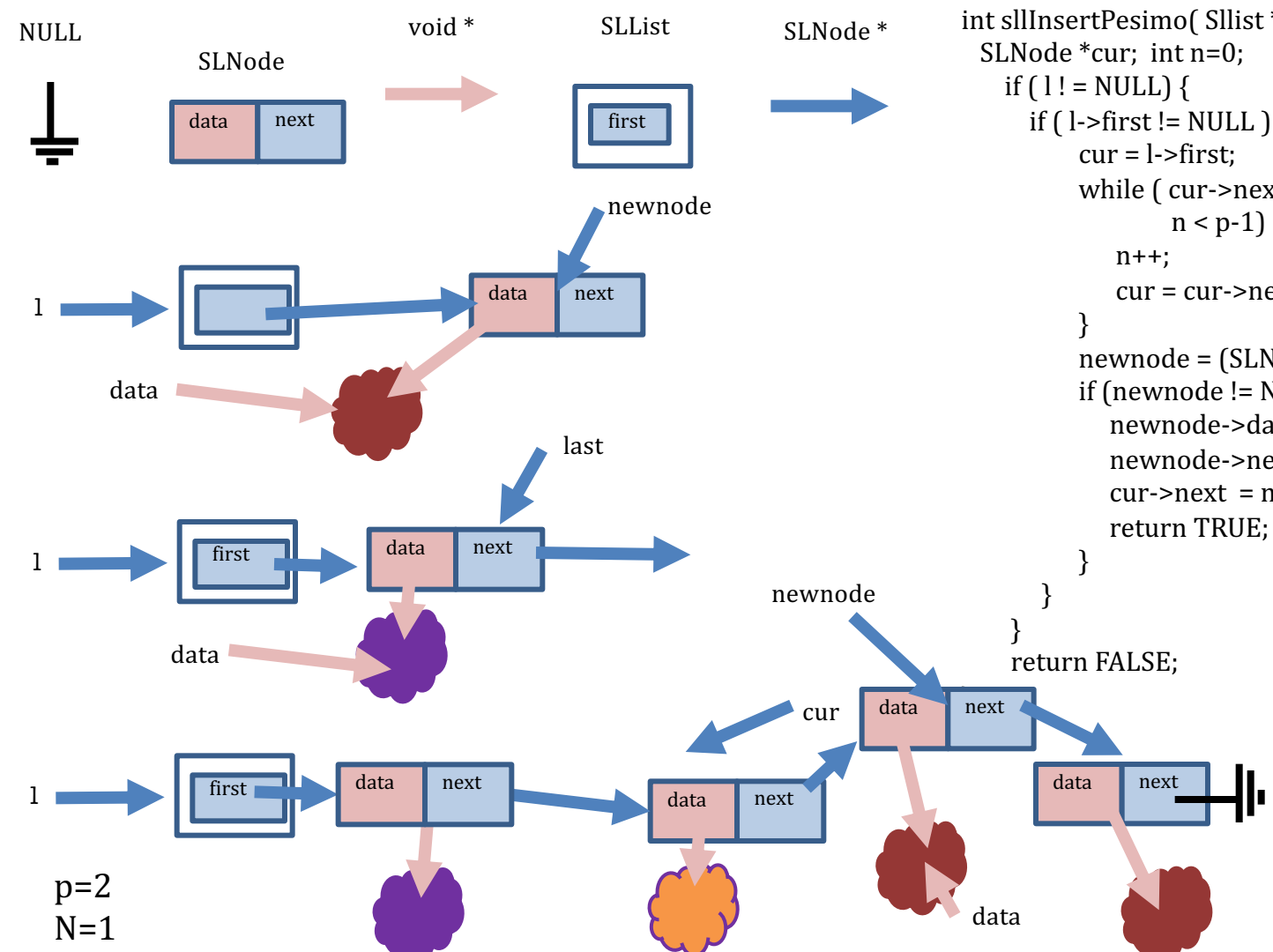
- Acessar último elemento (sllGetLast)
- Tamanho da lista (sllNElms)
- Inserir um element na posição p (sllInsertNesimo)
- Consultar um elemento da lista identificado por uma chave
- Remover um elemento da lista identificado por uma chave
- Inserir após um elemento da lista identificado por uma chave
- Inserire antes de um elemento da lista identificado por uma chave
- Criar lista vazia (SllCreate)
- Inserir no início de uma lista (sllINsertFirst)
- Acessar primeiro elemento (SllGetFirst)
- Remover o ultimo element (SllRemoveFirst)
- Destruir a lista (SllDestroy)
- Inserir um element na ultima posicao (SllInsertLast)



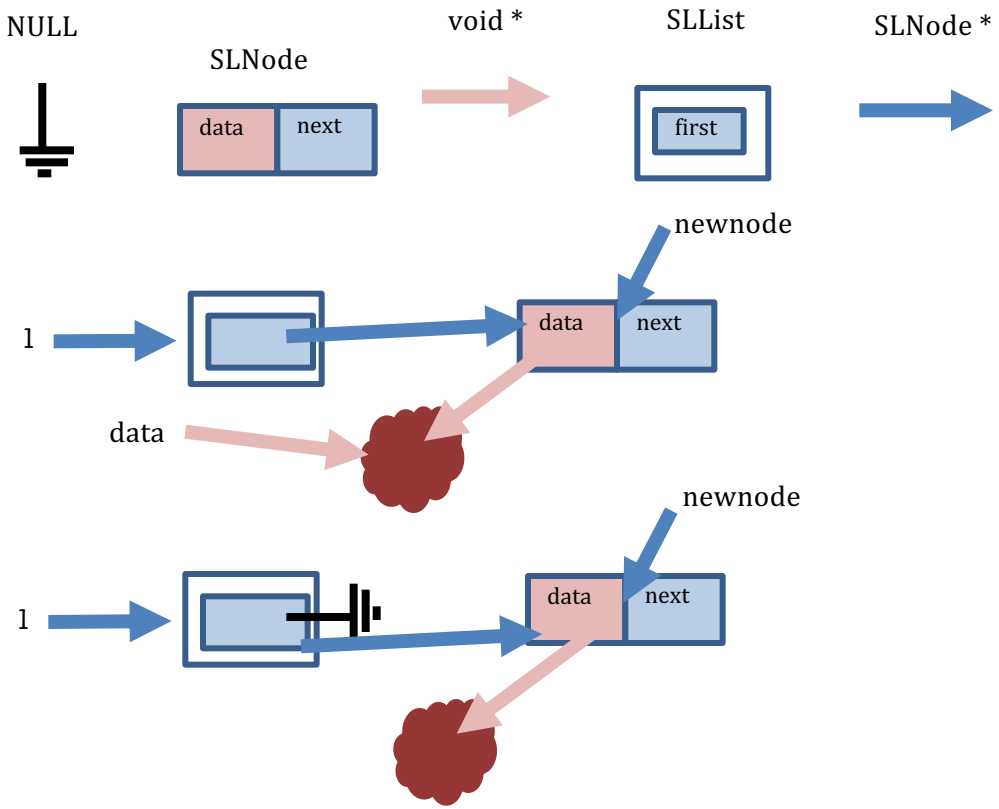

```
void * sllGetLast( Sllist *l)
{
  SLNode *cur;
     if ( l! = NULL) {
        if ( l->first != NULL ) {
              cur = l->first;
              while ( cur->next != NULL ) {
                   cur = cur->next;
              }
              return cur->data;
        }
     }
     return NULL
}
```




```
void * sllGetNumElms( Sllist *l)
{
  SLNode *cur;  int n=0;
     if ( l ! = NULL) {
        if ( l->first != NULL ) {
              cur = l->first;
              while ( cur->next != NULL ) {
                   n++;
                   cur = cur->next;
              }
               n++;
              return n;
        }
        return 0;
     }
     return -1;
}
```




```
int sllInsertPesimo( Sllist *l, int p, void *data) {
  SLNode *cur;  int n=0;
    if ( l ! = NULL) {
      if ( l->first != NULL ) {
            cur = l->first;
            while ( cur->next != NULL
                      n < p-1) {
                n++;
                cur = cur->next;
            }
            newnode = (SLNode *)malloc(sizeof(SLNode));
            if (newnode != NULL ) {
               newnode->data = elm;
               newnode->next = cur->next;
               cur->next  = newnode;
               return TRUE;
            }
        }
    }
    return FALSE;
```




```
int sllInsertPesimo( Sllist *l, int p, void *data) {
  SLNode *cur;  int n=0;
    if ( l ! = NULL) {
     if ( p>0 ) {
      if ( l->first != NULL ) {
            cur = l->first;
            while ( cur->next != NULL
                      n < p-1) {
                n++;
                cur = cur->next;
            }
            newnode = (SLNode *)malloc(sizeof(SLNode));
            if (newnode != NULL ) {
               newnode->data = elm;
               newnode->next = cur->next;
               cur->next  = newnode;
               return TRUE;
            }
       }  else {
           newnode = (SLNode *)malloc(sizeof(SLNode));
            if (newnode != NULL ) {
               newnode->data = elm;
               newnode->next = l->first;
               l->first = newnode;
               return TRUE;
            }
}
     return FALSE;
```

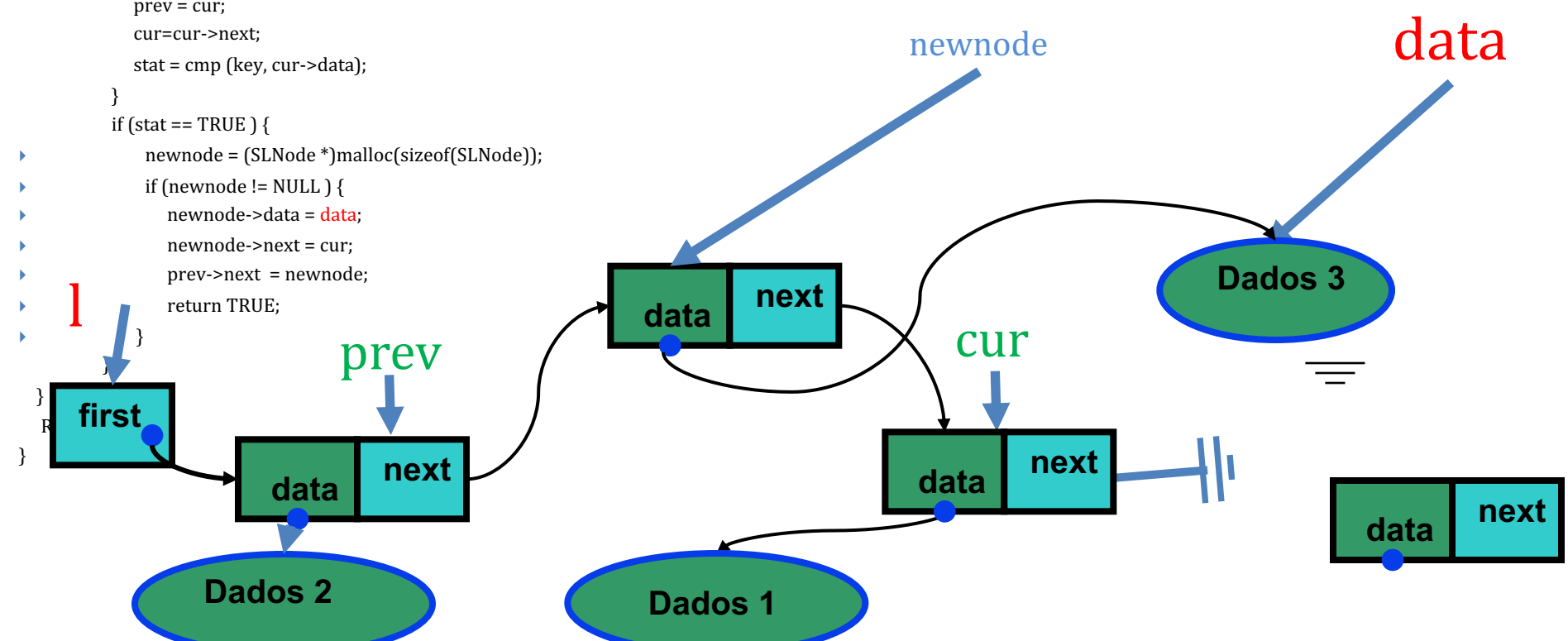
```
Void *sllGetFirst (SLList *l)
{
   if (l != NULL ) {
     if ( l->first != NULL ) {
          l->cur = l->first;
          return l->cur->data;
     }
   }
   return NULL;
}
```

```
Void *sllGetNext (SLList *l)
{
   if (l != NULL ) {
     if ( l->first != NULL ) {
        if ( l->cur->next != NULL ) {
             l->cur= l->cur->next;
            return l->cur->data;
     }
   }
   return NULL;
}
```

1 - Inserir antes um elemento identificado pela chave

```
int sllInsertBeforeSpec( SLList *l, void *data, void *key, int (*cmp) (void *, void *))
{.  SLNode *cur, *prev, *newnode; int stat;
    if (l != NULL ) {
      if ( l->first != NULL ) {
            cur = l->first; prev = NULL;
            stat = cmp (key, cur->data);
            while( stat != TRUE && cur->next != NULL) {
                prev = cur;
                cur=cur->next;
                stat = cmp (key, cur->data);
            }
            if (stat == TRUE ) {
                newnode = (SLNode *)malloc(sizeof(SLNode));
                if (newnode != NULL ) {
                    newnode->data = data;
                    newnode->next = cur;
                    prev->next  = newnode;
                    return TRUE;
                }
    }
  }
  R
}
```

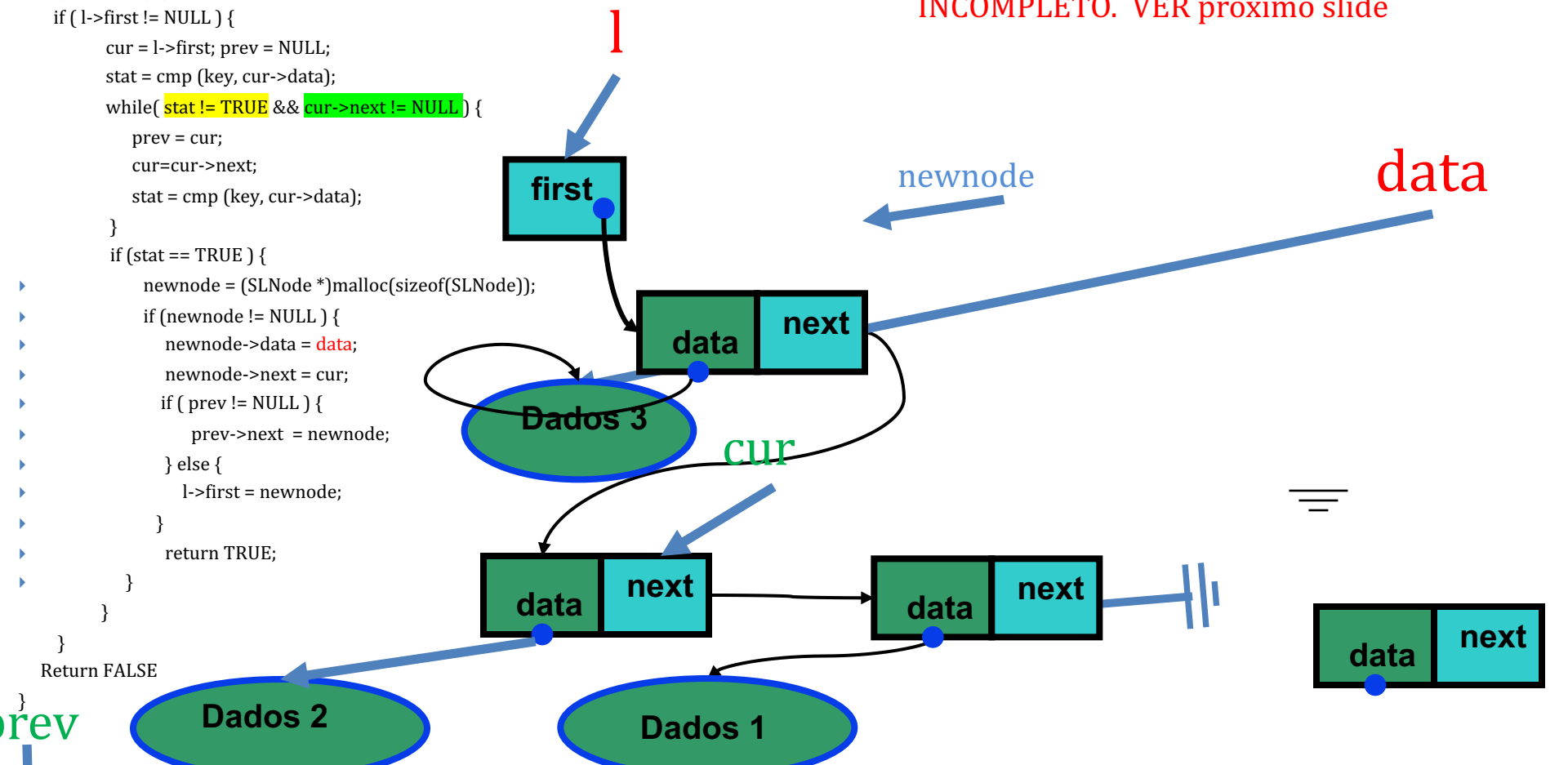
INCOMPLETO. VER próximo slide



1 - Inserir antes um elemento identificado pela chave

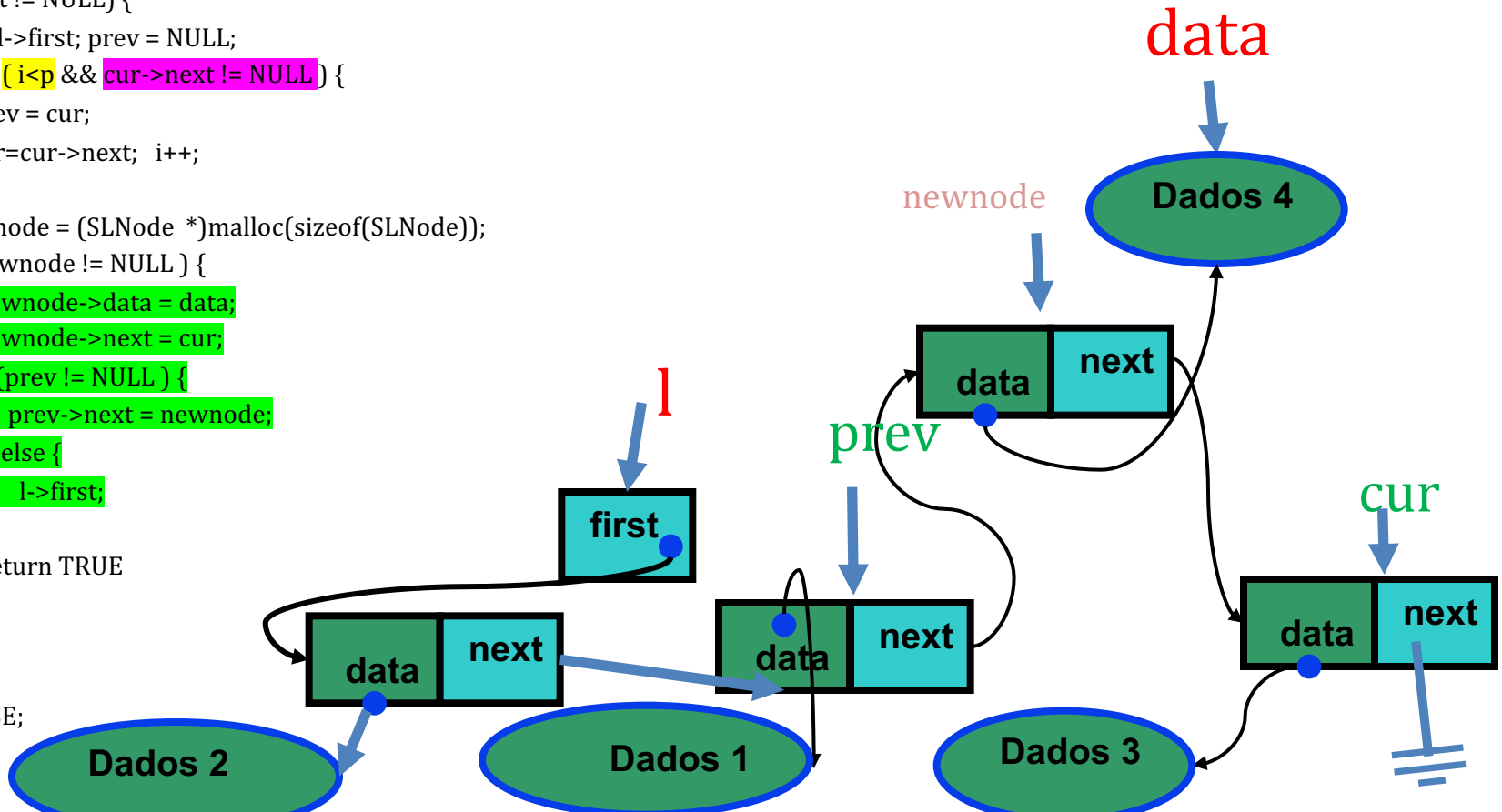
```
int sllInsertBeforeSpec( SLList *l, void *data, void *key, int (*cmp) (void *, void *))
{   SLNode *cur, *prev, *newnode; int stat;
    if (l != NULL ) {
      if ( l->first != NULL ) {
            cur = l->first; prev = NULL;
            stat = cmp (key, cur->data);
            while( stat != TRUE && cur->next != NULL) {
                prev = cur;
                cur=cur->next;
                stat = cmp (key, cur->data);
            }
            if (stat == TRUE ) {
                newnode = (SLNode *)malloc(sizeof(SLNode));
                if (newnode != NULL ) {
                    newnode->data = data;
                    newnode->next = cur;
                   if ( prev != NULL ) {
                        prev->next  = newnode;
                    } else {
                       l->first = newnode;
                   }
                    return TRUE;
                }
            }
      }
    Return FALSE
}
```

INCOMPLETO. VER próximo slide



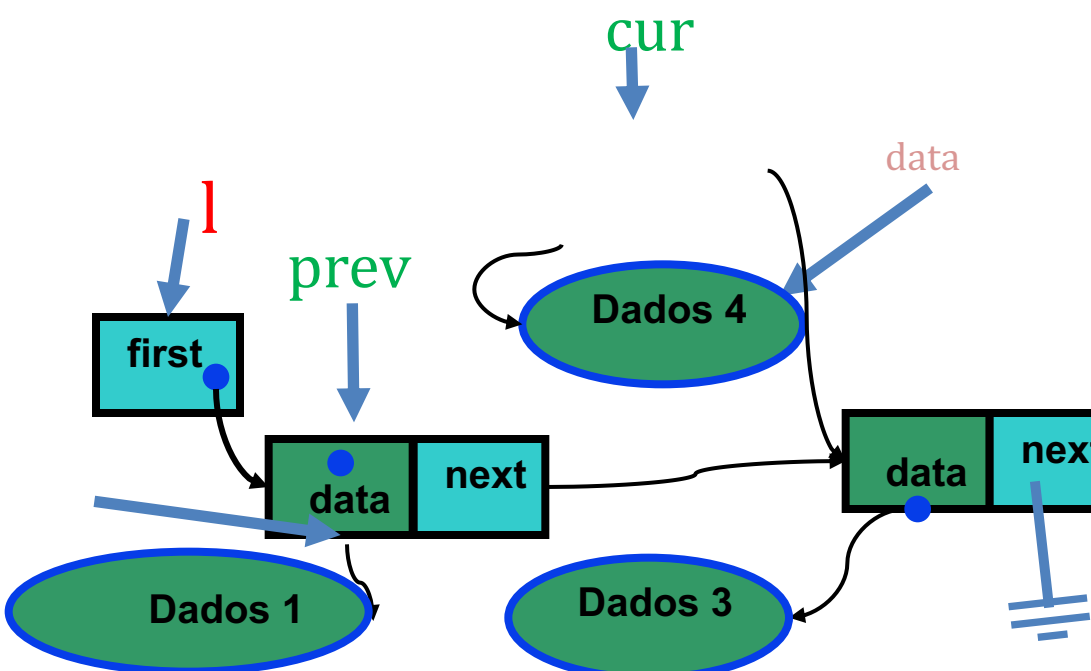
```
int sllInsertBeforePesimo( Sllist *l, void *data, int p)
{
    SLNode *newnode, *cur, *prev; int i=0;
    if ( l != NULL ) {
        if ( l->first != NULL) {
            cur = l->first; prev = NULL;
            while ( i<p && cur->next != NULL ) {
                prev = cur;
                cur=cur->next;   i++;
            }
             newnode = (SLNode  *)malloc(sizeof(SLNode));
             if(newnode != NULL ) {
                newnode->data = data;
                newnode->next = cur;
                if (prev != NULL ) {
                    prev->next = newnode;
                } else {
                    l->first;
                }
                return TRUE
            }
        }
    }
     return FALSE;
}
```

2 - Corrigir a funcao sllInsertBeforePesimo tratando o caso em que p-esimo é o primeiro elemento



- Remover um elemento identificado pela chave”

```
void *sllRemoveSpec( SLList *l, void *key, int (*cmp) (void *, void *))
{
    if (l != NULL ) {
      if ( l->first != NULL ) {
         cur = l->first; prev = NULL;
          stat = cmp (key, cur->data);
          while( stat != TRUE && cur->next != NULL) {
                    prev = cur;
                    cur=cur->next;
                    stat = cmp (key, cur->data);
           }
          if (stat == TRUE ) {
              data = cur->data;
               if (prev != NULL ) {
                  prev->next = cur ->next;
               } else {
                  l->first = cur->next;
               }
               free (cur);
                return data;
          }
       }
    }
     return NULL;
}
```



Escreva um algoritmo que recebe duas listas (L1 e L2) e retorna um valor lógico: Verdadeiro, se L1 = L2 e Falso, se L1 <> L2. É igual se todo valor em l1 tem um no com o memso valor em l2 e viceversa

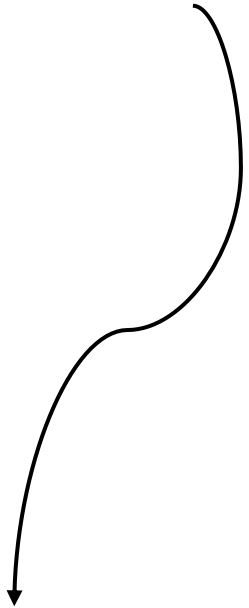
```
int sllÉIgual ( SLList *l1, SLList *l2, int (*cmp) ( void *, void *))
{
   if ( l1 != NULL && l2 != NULL ) {
      if ( l1->first != NULL && l1->first != NULL ) {
         cur1 = l1->first;
         while ( cur1->next != NULL ) {
            cur2 = l2->first; stat = FALSE;
            while (cur2->next != NULL ) {
               if ( cmp (cur1->data, cur2->data ) == TRUE ) {
                  stat = TRUE;
               }
               cur2 = cur2->next
            }
            if ( stat == FALSE ) {
               return FALSE;
            }
            cur1 = cur1->next;
         }
         return TRUE;
      }
   }
   return FALSE;
}
```

1. Escreva um algoritmo diferença (L, L1, L2) para construir a lista L igual à diferença L1-L2;
   SLList *sllDiferença (SLList *l1, SLList *l2, int (*cmp) (void *, void *))
1. Escreva um algoritmo NumComuns (L1,L2) , que deve retornar um valor inteiro igual ao número de valores comuns às duas listas ordenadas L1 e L2;
   int sllNumComuns (SLList *l1, SLList *l2, int (*cmp) (void *, void *))

1. Escreva um algoritmo ÉInversa (L1, L2) que retorna 1 se a lista L1 tem os mesmos elementos de L2 na ordem inversa, -1 se L1 tem menos elementos que L2 e 0 se L1 tem mais elementos que L2 (30 pontos)
2. Receber duas listas circulares simplesmente encadeadas (L1 e L2), incluir todos os nós de L2 em L1, de maneira intercalada. Não pode alocar novos nós. A lista L1 ficara com um nó de L1, sempre seguido de um nó de L2

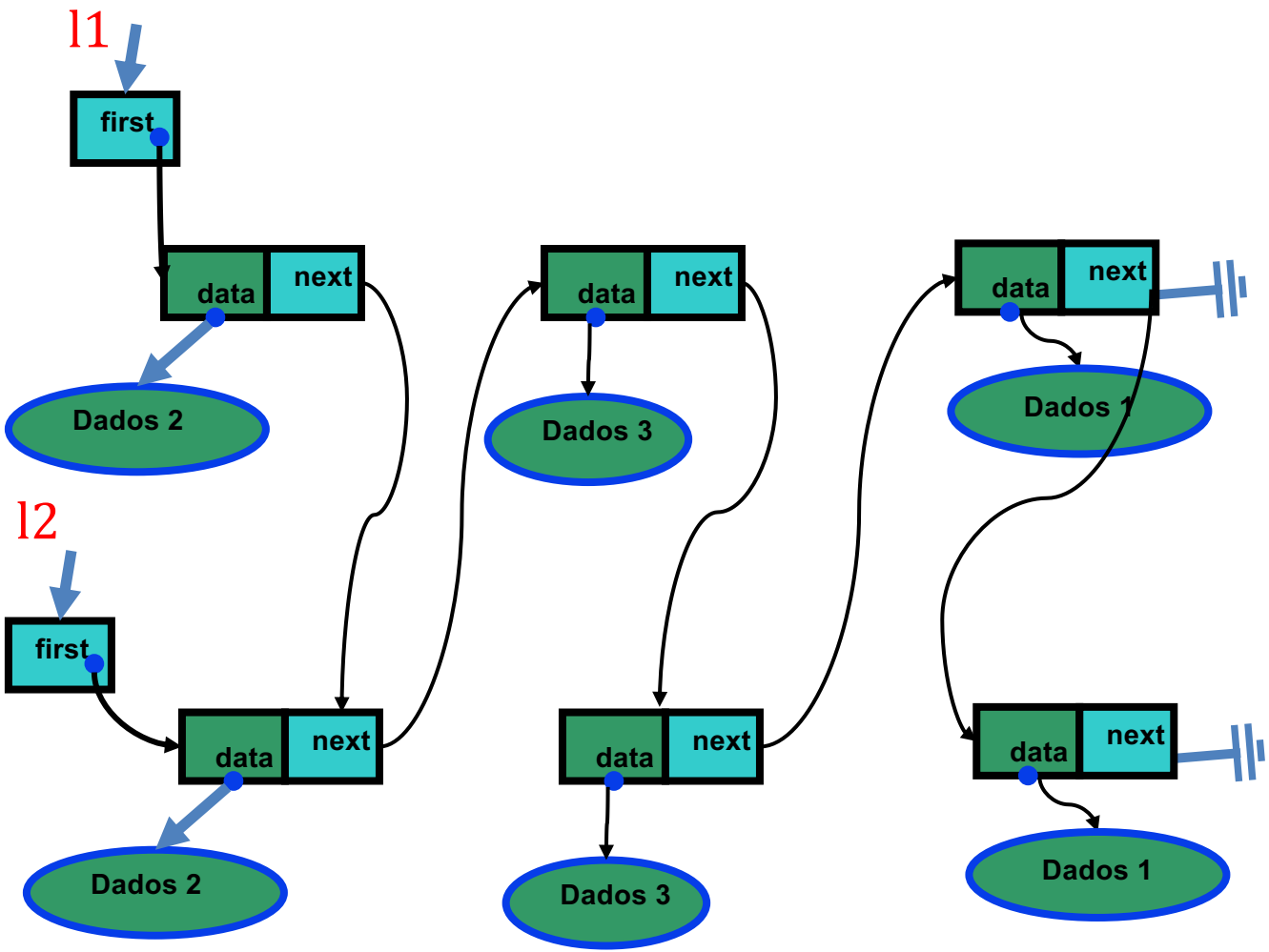
- Receber duas listas circulares simplesmente encadeadas (L1 e L2), incluir todos os nós de L2 em L1, de maneira intercalada. Não pode alocar novos nós. A lista L1 ficara com um nó de L1, sempre seguido de um nó de L2

Int sllIntercala(SLList *l, Sllist *l2



- Receber duas listas circulares simplesmente encadeadas (L1 e L2), incluir todos os nós de L2 em L1, de maneira intercalada. Não pode alocar novos nós. A lista L1 ficara com um nó de L1, sempre seguido de um nó de L2

```
Int sllIntercala( SLList *l, Sllist *l2)
{
  if ( l1 != NULL && l2 != NULL ) {
     if (l1->first != NULL && l2->first != NULL ) {
        Cur1 = l1-> first;
         Cur2 = l2->first;
         Next1 = cur1->next;
         Next2 = cur2->next;
        While (cur1 != NULL &&  cur2 != NULL) {
               Cur1->next = cur2;
               Cur2->next = next1;
               Cur1 = next1;
               Cur2 = next2;
              If ( cur1 != NULL ) {
                 Next1 = cur1->next;
               }
               If ( cur2 != NULL) {
                    Next2 = cur2->next;
               }
          }
          return TRUE;
     }
   }
  return FALSE;
}
```



data

newnode

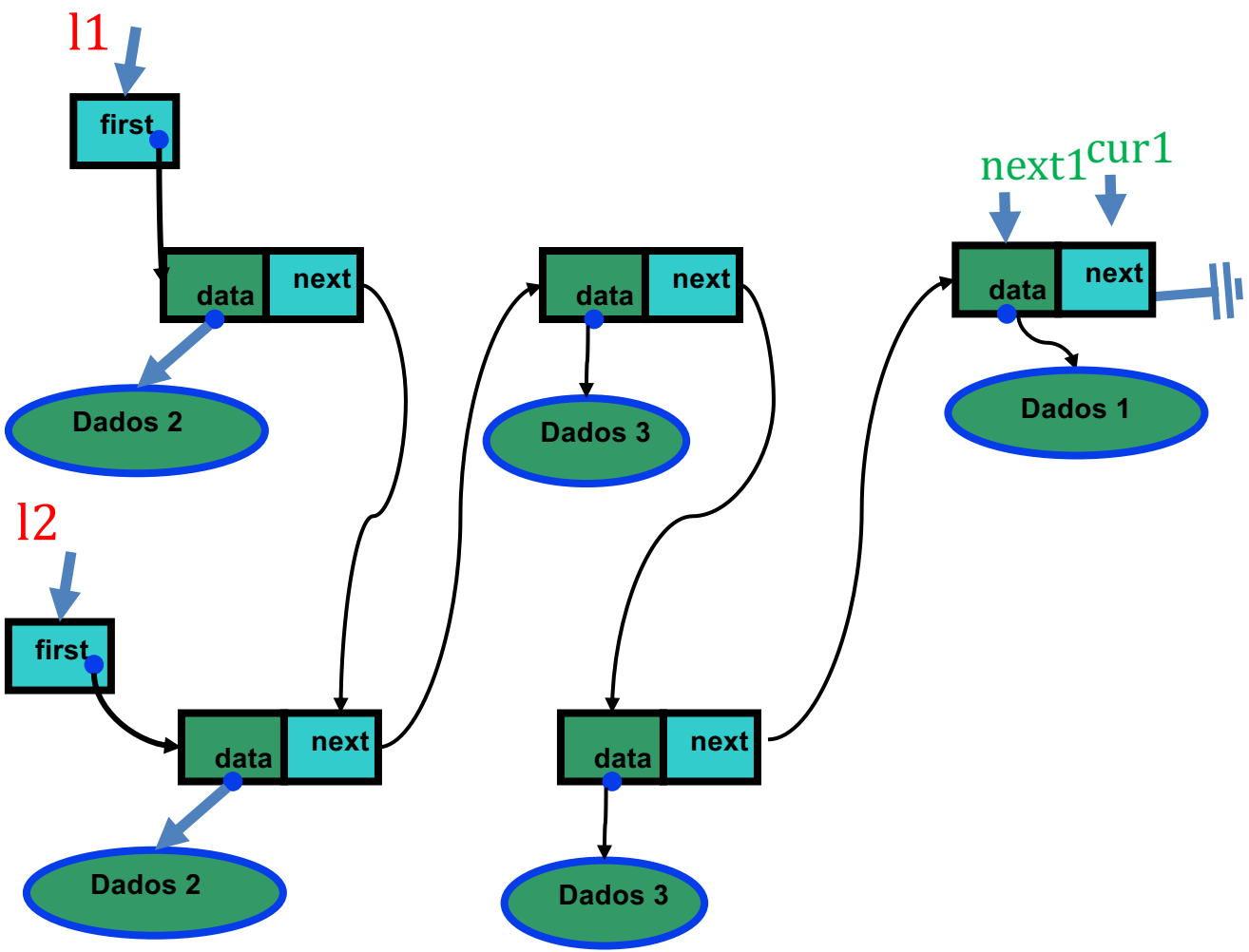
next1

cur1

```
Cur1 = l1-> first;
Cur2 = l2->first;
Next1 = cur1->next;
Next2 = cur2->next;

While (cur1 != NULL &&
        cur2 != NULL) {
Cur1->next = cur2;
Cur2->next = next1;
Cur1 = next1;
Cur2 = next2;
If ( cur1 != NULL ) {
Next1 = cur1->next;
}
If ( cur2 != NULL) {
Next2 = cur2->next;
}
}
```

next2

cur2




```
Cur1 = l1-> first;
Cur2 = l2->first;
Next1 = cur1->next;
Next2 = cur2->next;

While (cur1 != NULL &&
        cur2 != NULL) {
Cur1->next = cur2;
Cur2->next = next1;
Cur1 = next1;
Cur2 = next2;
If ( cur1 != NULL ) {
Next1 = cur1->next;
}
If ( cur2 != NULL) {
Next2 = cur2->next;
}
}
```