Listas

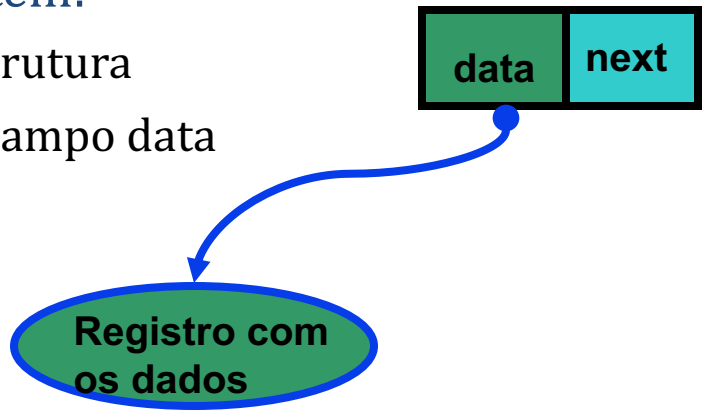
Estrutura de Dados
Prof. Anselmo C. de Paiva
Departamento de Informática – Núcleo de Computação Aplicada NCA-UFMA

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

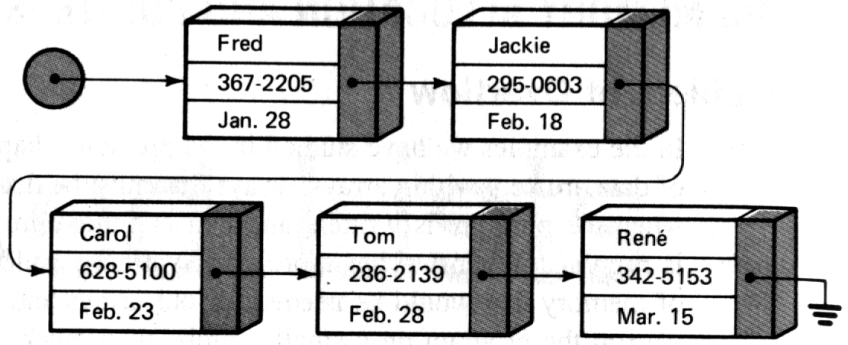
Listas Encadeadas

- 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

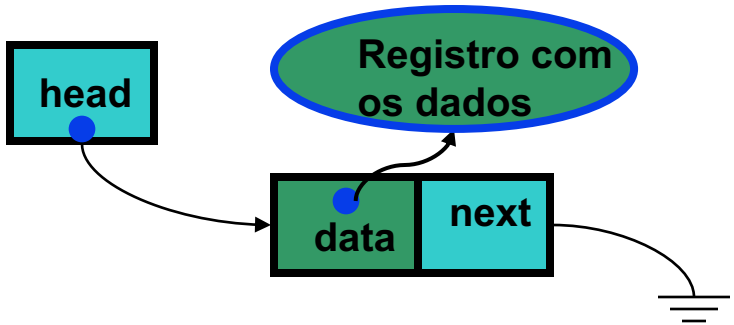


Listas Encadeadas

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

Listas Encadeadas

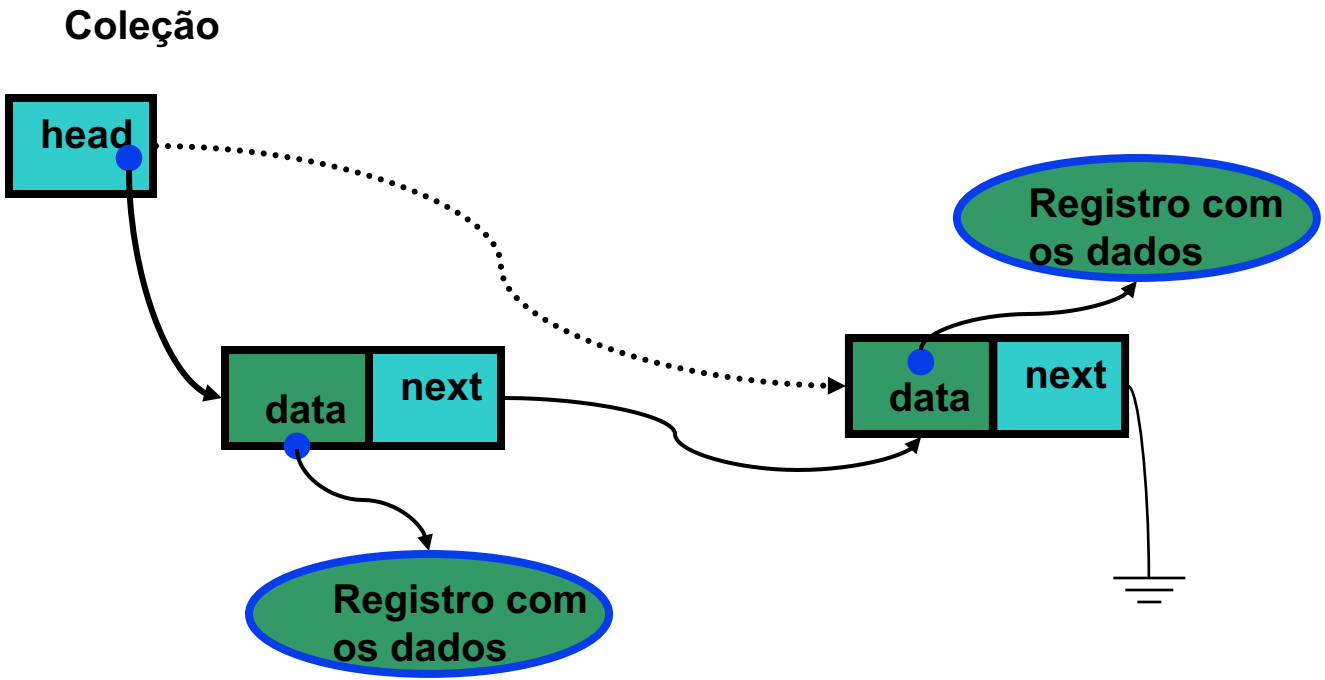
- Insere o primeiro elemento na lista
  - Aloca espaço para um *node*
  - Atribui endereço do dado para o ponteiro data do node
  - Atribui NULL para o campo next do node
  - Atribui o endereço do novo node para o campo head da Coleção



Listas Encadeadas

- Insere o segundo item
  - 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 head da Collection
  - Atribui o campo next do novo nodecom o valor de head
  - Atribui o endereço do novo node ao campo head da Collection

Listas Encadeadas



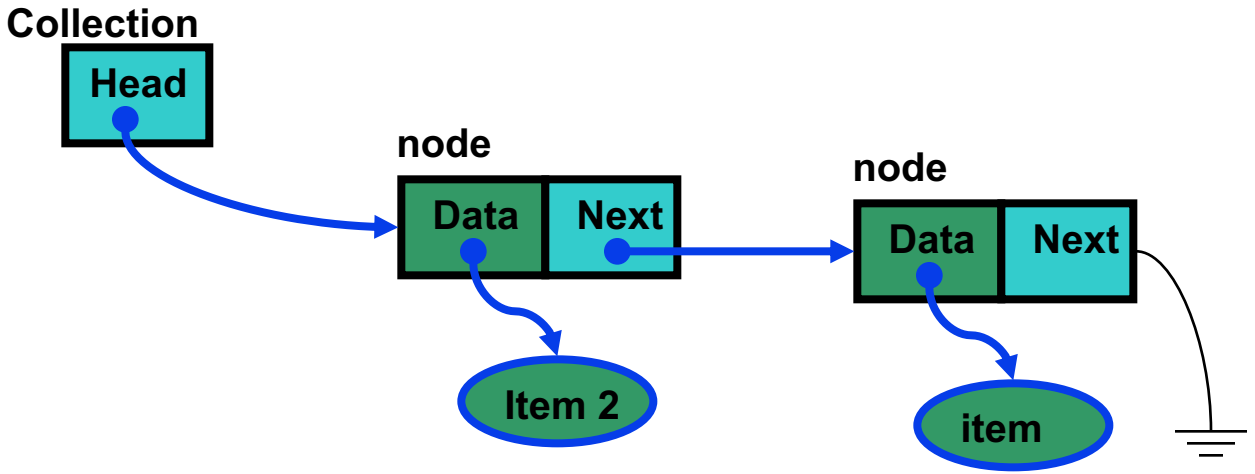
Listas - Implementação operação ADD

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

int AddToCollection( Collection *l, void *item )
{
    SLNode *newnode = malloc( sizeof(SLNode) );
    new->data = item;
    new->next = c->first;
    l->first = new;
    return TRUE;
}
```

Listas Ligadas

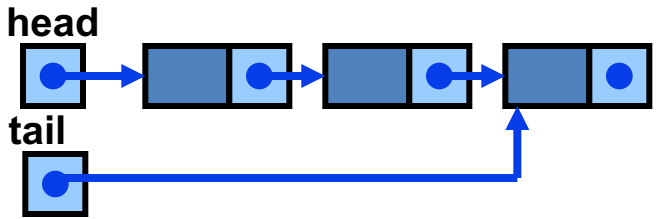
- Tempo para inserção
  - Constante - independente do número de elementos na lista n
- Tempo de Consulta
  - Pior caso - n



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

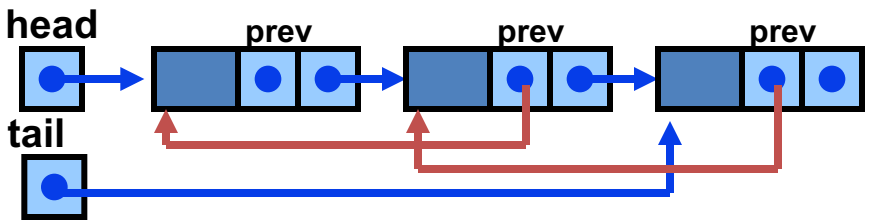
```
struct _slnode_ {
    void *data;
    struct _slNode_ *next;
} SLNode;
struct _lqueue_ {
 SLNode *head, *tail;
}LQueue;
```



Listas - Duplamente encadeadas

- Podem ser percorridas nas duas direções

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



Operações sobre Listas Ligadas

- Criar lista vazia
- Inserir primeiro elemento
- Inserir no início de uma lista
- Acessar primeiro elemento
- Acessar último elemento
- Tamanho da lista
- Inserir valor v na posição p+1

```
// 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;
}
}
```

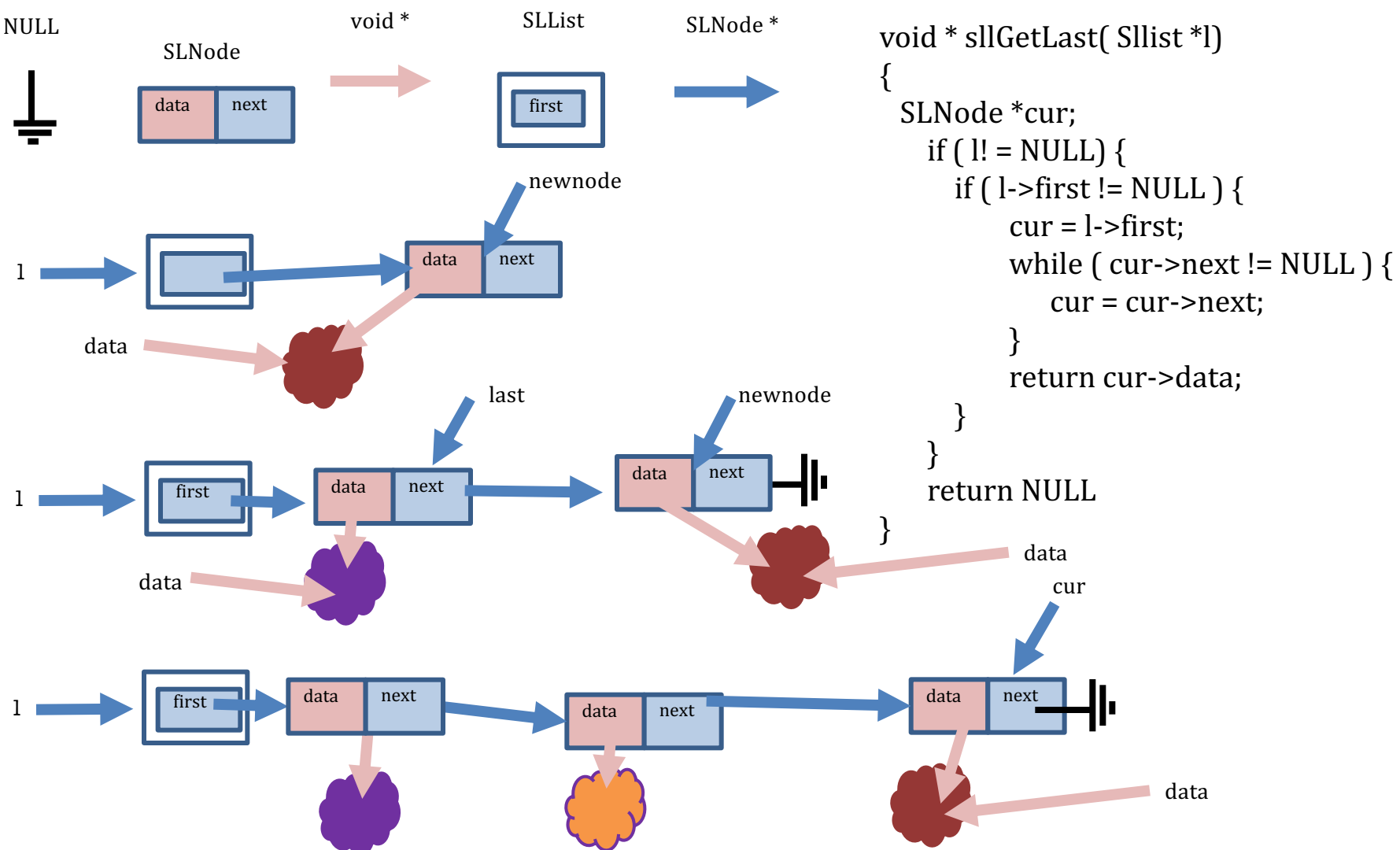
```
//SLLStack.c
SLLIst  *sllCreate(   )
{
     SLList *l;
     l = (SLList     *) malloc ( sizeof (SLList ));
     if ( l != NULL ) {
              l->first  = NULL;
              return l;
     }
     return NULL;
}

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;
}
```

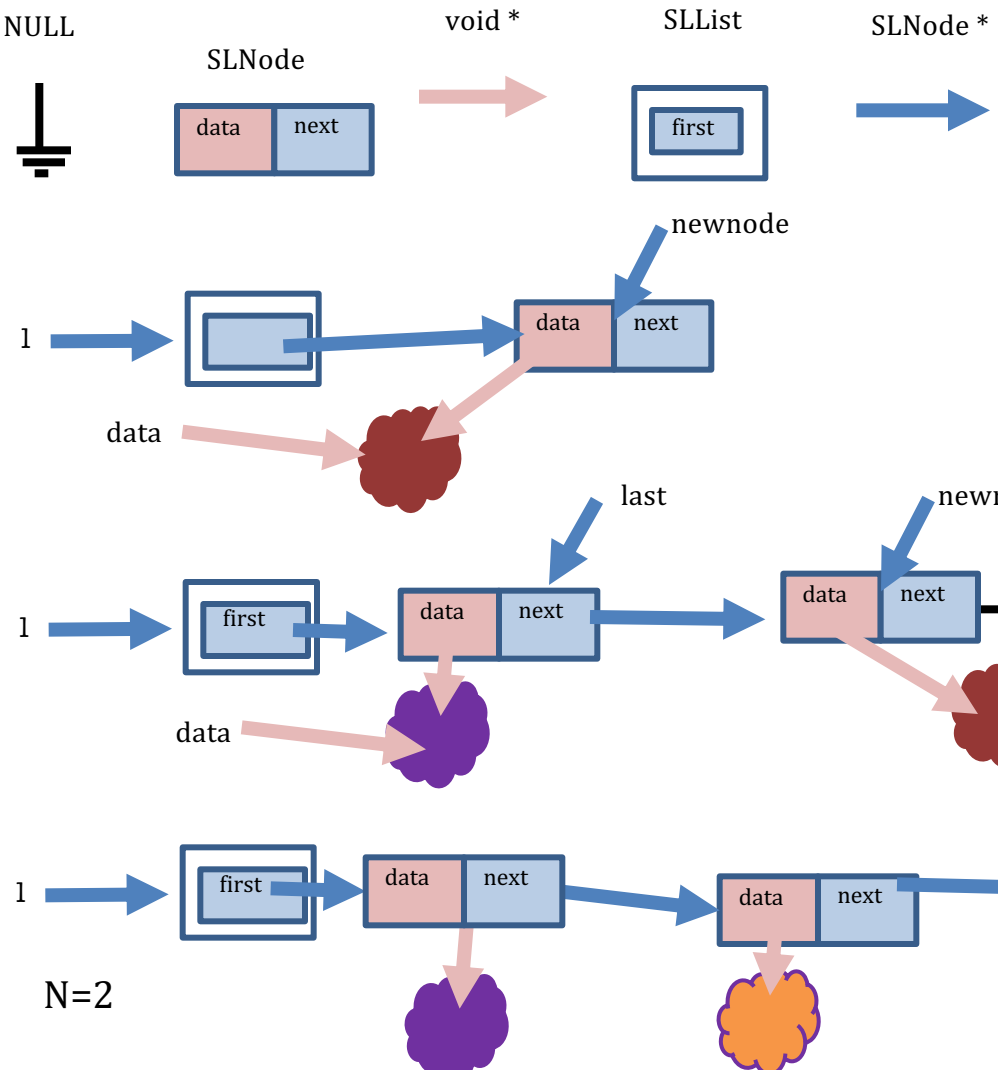
```
void *sllRemoveFirst ( SLList *l)  // slstkPop
{
  SLNode *aux;
  void * data;
  if ( l != NULL ){
     if ( l->first !-= NULL ) {
       aux =  l-> first;
       data = aux->data;
       l->first = aux->next;
       free(aux);
       return data;
     }
  }
   return  NULL;
}

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

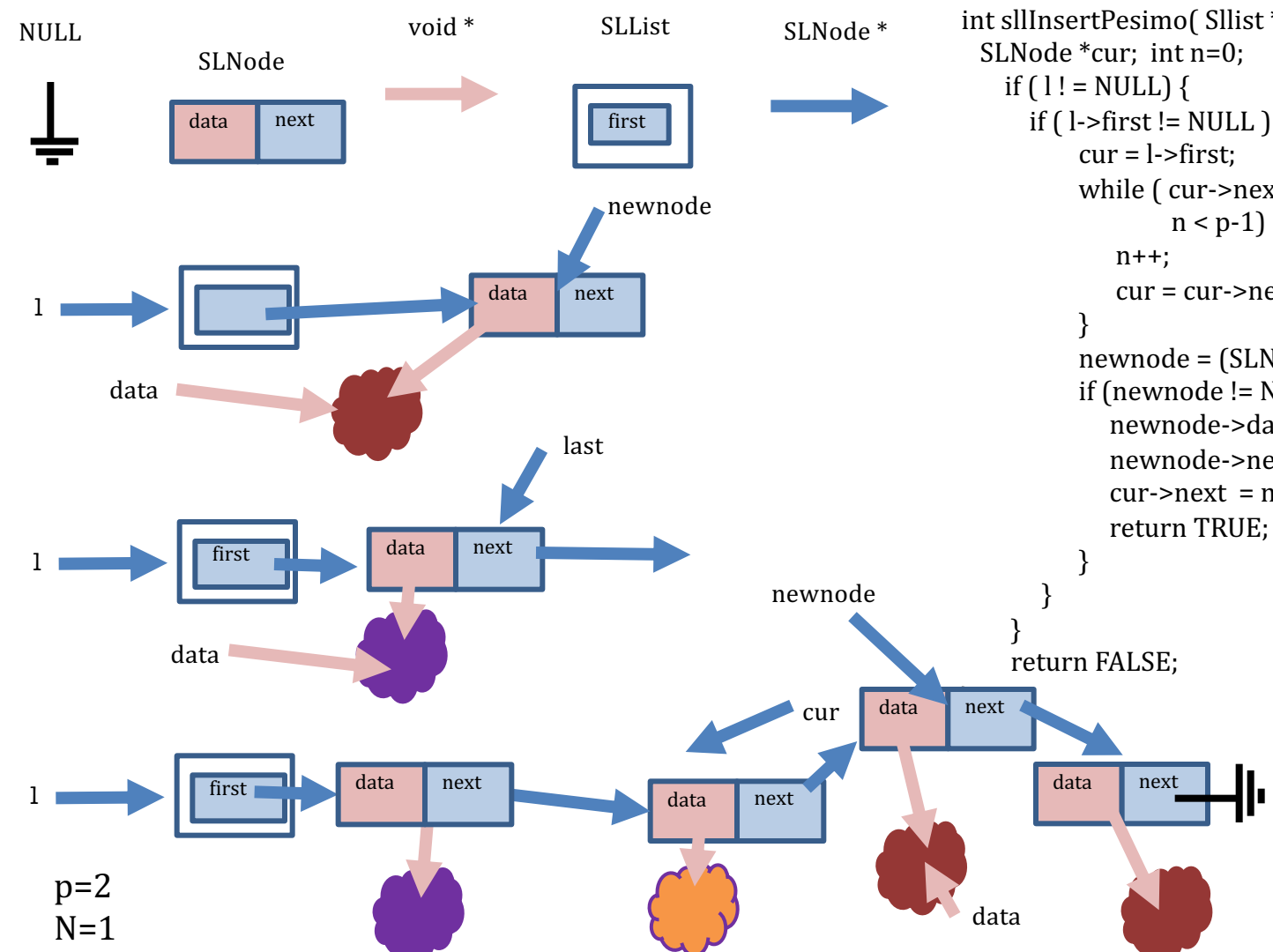
- Acessar último elemento (sllGetLast)
- Tamanho da lista (sllNElms)
- Inserir um element na posição p (sllInsertNesimo)
- Consultar um elemento da lista identificado por uma chave
- Remover um elemento da lista identificado por uma chave
- Inserir após um elemento da lista identificado por uma chave
- 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)



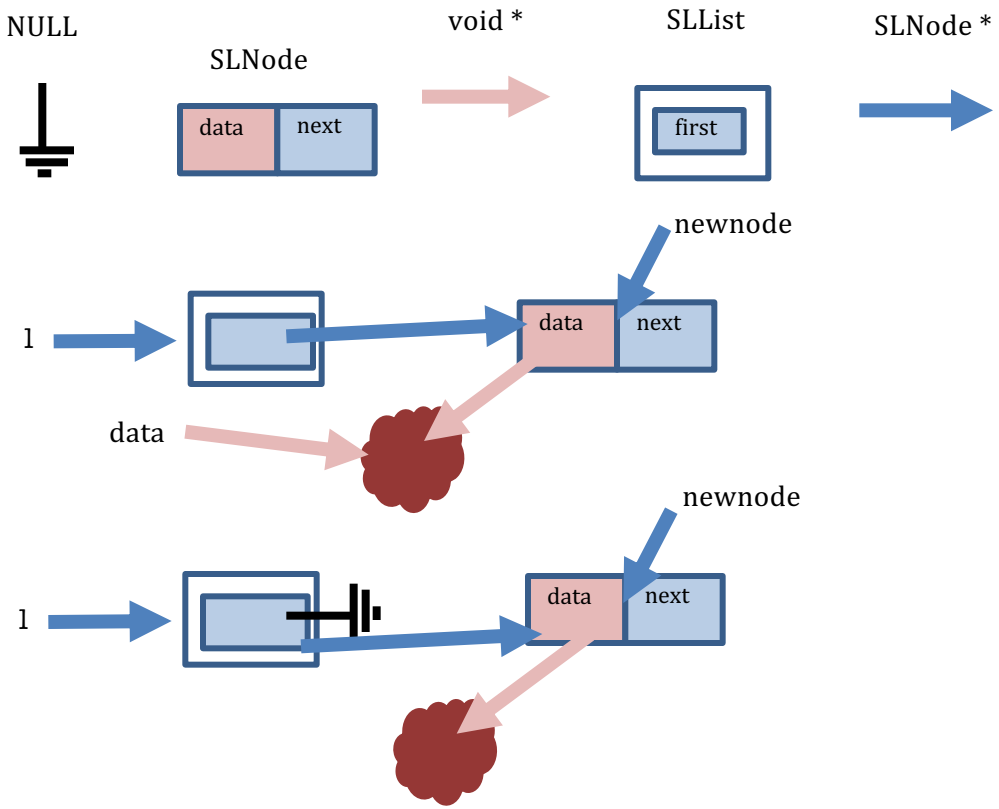

```
void * sllGetLast( Sllist *l)
{
  SLNode *cur;
     if ( l! = NULL) {
        if ( l->first != NULL ) {
              cur = l->first;
              while ( cur->next != NULL ) {
                   cur = cur->next;
              }
              return cur->data;
        }
     }
     return NULL
}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




```
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;
```

Implementação das Operações

1- Criação da lista vazia

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

2) Inserção do primeiro item

```
int  sllAddAsFirst  (SLList *l, void *item)
{
   SLNode *newNode;
    if (  l!= NULL ) {
       newNode= (SLNode *) malloc(sizeof(SLNode));
       if (newNode != NULL ) {
           newNode->data =item;
            newNode->next = l->first;
           l->first = newNode;
           return TRUE;
       }
    }
   return FALSE
}
```

Implementação das Operações

4) Acesso ao primeiro elemento

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

5) Acesso ao último elemento

```
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;
}
```

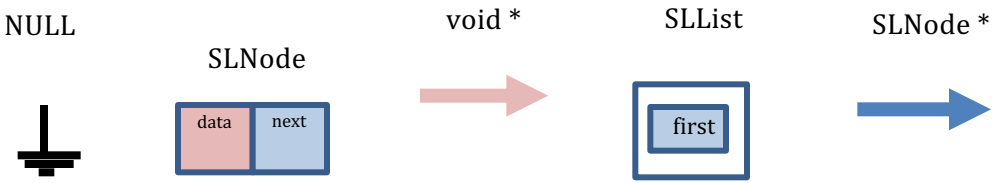
Implementação das Operações

6) Qual o número de elementos ?

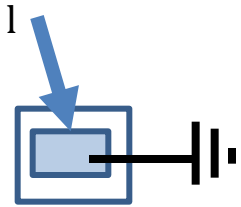
```
int sllGetNumElm( SLList *l )
{
    SLNode *cur;
    int nElm=0;
   if ( l  != NULL) {
      if( l->first  != NULL ) {
          cur = l->first;
          while(current->next != NULL) {
                nElm++;
              cur = cur->next;
           }
           return nElm;
      }
   }
}
```

Outras Operações

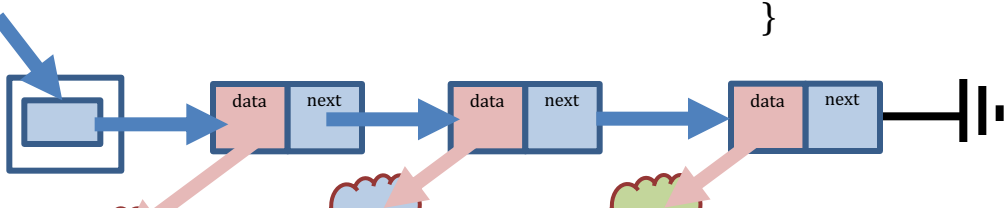
- Eliminar elemento da posição k+1
- Eliminar primeiro elemento
- Eliminar valor v
- Inserir valor v antes do elemento apontado por p
- Criar uma lista com registros numerados
- Eliminar sucessor de p
- Imprimir recursivamente

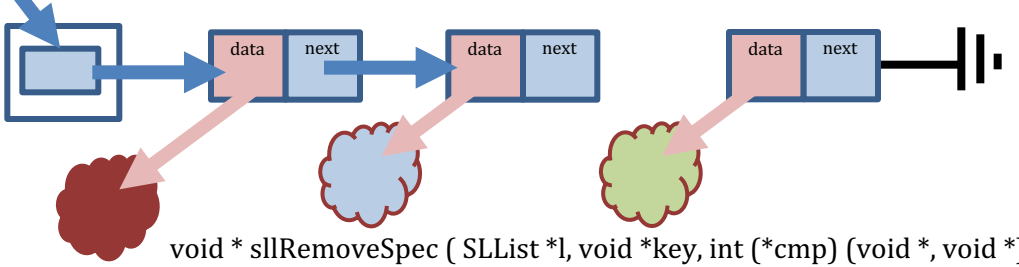


Consultar um elemento da lista identificado por uma chave




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






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

Remover no do meio da lista
salvar os dados (data)
prev->next = spec->next
free(spec);

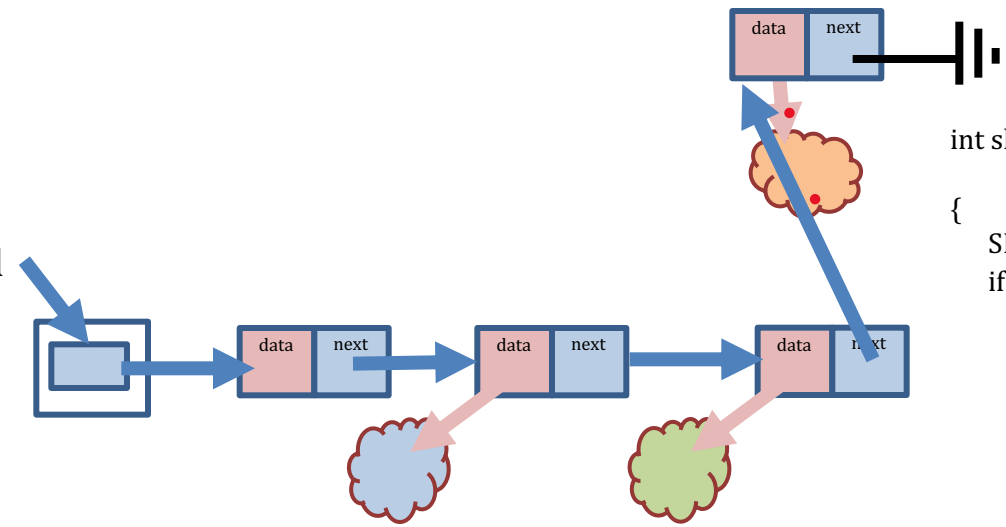
Remover o no especificado quando ele é o ultimo
Salvar os dados (data)
prev->next = spec->next;
Free(spec)

Spec é o primeiro

Salvar os dados (data)
L->first = spec->next
Free(spec);

Achar o No especificado, sabendo quem é o anterior

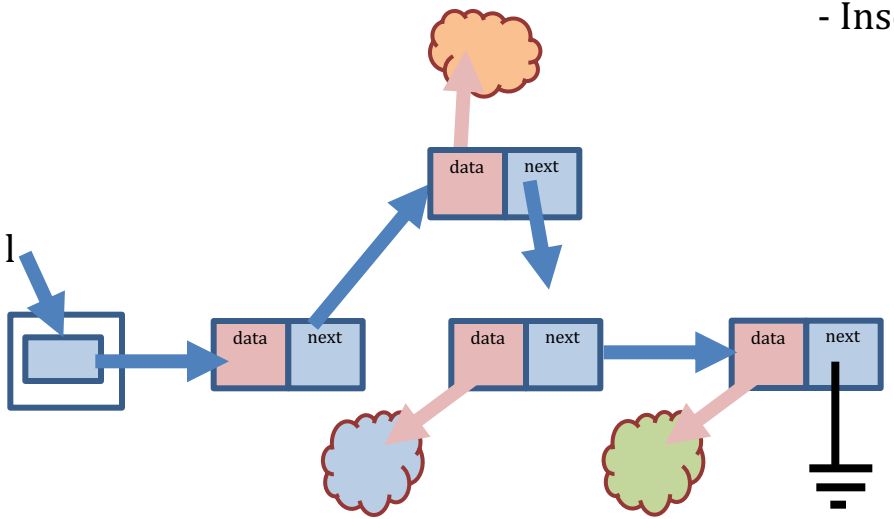
```
 spec = l->first;
 prev = NULL
 while ( spec>next != NULL && cmp(key, cur->data) != TR
    prev = spec;
    spec = spec>next;
}
```

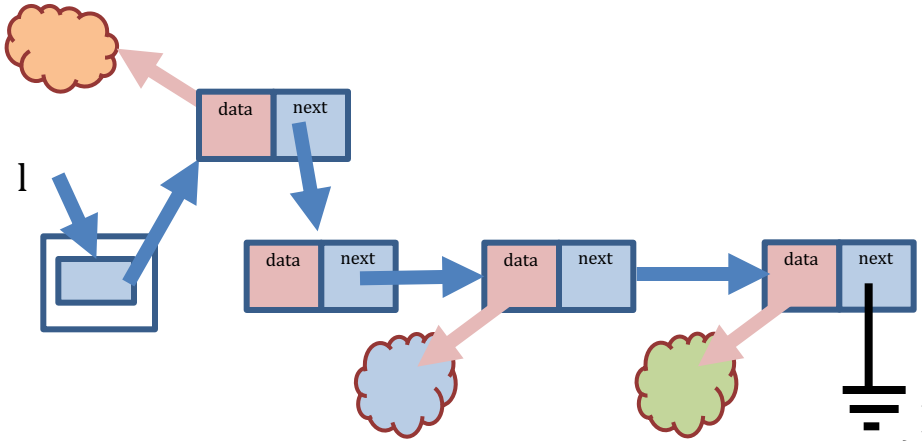



```
int sllInsertAfterSpec ( SLList *l, void *key, int (*cmp) (void *, void *),
                                                    void *data)
{
    SLNode *newnode, *spec;
    if ( l != NULL ) {
        if ( l->first != NULL ) {
            spec = l->first;
            while ( spec>next != NULL &&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

Cria newnode
Coloca os dados em newnode
Faz newnode->next = spec->next
Faz spec->next = newnode

- Inserir antes de um nó especifciado pela chave

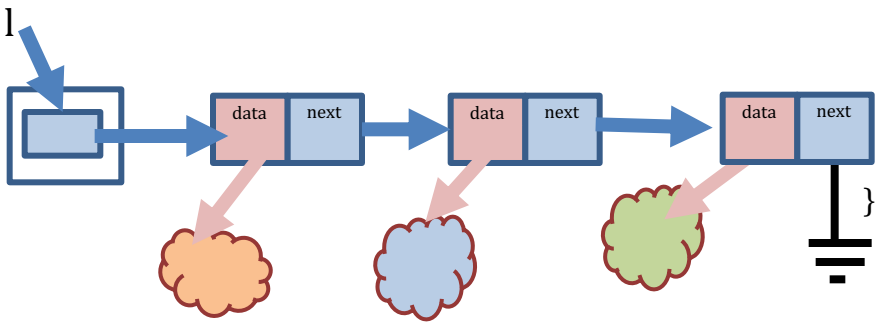





```
int sllInsertBeforeSpec ( SLList *l, void *key,
                          int (*cmp) (void *, void *), void *data)
{.   SLNode *prev, *spec,*newnode;
     if ( l != NULL ) {
        if ( l->first != NULL ) {
           spec = l->first;
           prev = NULL;
           while ( cmp(key, spec->data)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

b) Remover o nó após um nó especifciado pela chave

```
void * sllRemoveAfterSpec ( SLList *l, void *key, int (*cmp) (void *, void *
{
  SLNode *next, *spec;   void *data;
     if ( l != NULL ) {
        if ( l->first != NULL ) {
           spec = l->first;
           while ( cmp(key, spec->data) != TRUE && spec->next != NULL ){
               spec= spec->next;
           }
           if ( cmp(key, spec->data) = TRUE) {
             next  = spec->next;
             if (next != NULL ) {
                 data = next->data;
                 spec->next = next->next;
                 free(next);
                 return data;
             }
           }
        }
}
return NULL
}
```



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

```
void *sllGetFirst( SLList *l)
{
    if ( l!= NULL ) {
       l->cur = l->first;
       return l->first; // return  l->cur
    } else {
       return NULL;
    }
}
```

```
l->cur = l->cur->next;
```

```
next = l->cur;
next = next->next;
```

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

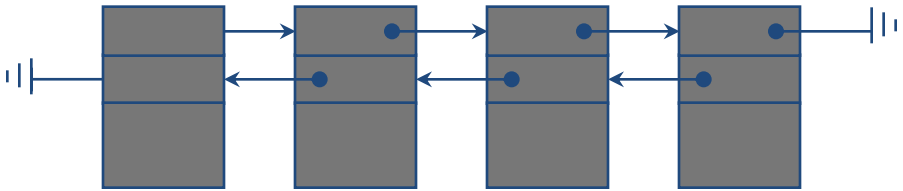
```
void *sllGetNext( SLList *l)
{
    if ( l!= NULL ) {
       if ( l->cur != NULL ) {
          l->cur = l->cur->next;
          return l->cur;
       }
    }
    return NULL;
}
```

Lista – Especificação do TAD

- Uma lista L de elementos do tipo T é uma sequência de elementos T, em conjunto com as seguintes operações:
  - Construir uma lista vazia
  - Determinar quando a lista está vazia ou não
  - Determinar quando a lista está cheia ou não
  - Encontrar o número de elementos na lista
  - Limpar uma lista para torná-la uma lista vazia
  - Inserir um elemento em uma posição especificada
  - Remover um elemento de uma posição especificada
  - Substituir um elemento em uma posição especificada
  - Percorrer a lista, realizando uma certa operação em cada entrada da lista

Listas Duplamente Encadeadas (LDE)

- Lista linear encadeada onde cada elemento sabe onde está localizado os seus dois vizinhos. Possui um ponteiro para o próximo(next) e para o anterior(previous)
- Permite movimentação bidirecional ao longo da lista



Operações sobre LDE

1. Criar lista vazia
2. Inserir numa das extremidades
3. Inserir após um elemento da lista
4. Inserir antes de um elemento da lista
5. Remover um elemento da lista
6. Localizar um elemento da lista

Implementação das Operações

```
typedef struct _dlnode_ {
  struct _dlnode_ *next,
                  *prev;
  void * data;
}DLNode;

typedef struct _dllist_
{
      DLNode *first;
       DLNode *cur;
}DLList;
```

```
DLList *dllCreate (void ) /* 1-  Cria lista vazia  */
{
    DLList *l;
    l = (DLList *)malloc(sizeof(DLList));
    if ( l != NULL ) {
        l->first = NULL;
        l->cur= NULL;
        return l;
    }
    return NULL;
}
int dllDestroy( DLList *l)
{
    if ( l != NULL ) {
       if ( l->first ==  NULL ) {
          free (l);
          return TRUE;
       }
    }
   return FALSE;
}
```

```
int dllInsertAsFirst(DLList *l, void *data)
{
      DLNode *newnode;
      if ( l != NULL ) {
         newnode = ( DLNode *) malloc (sizeof(DLNode));
         if (newnode != NULL ) {
            newnode->data = data;
             newnode->prev = NULL;
            newnode->next = l->first;
            first = l->first;
             if ( first != NULL ) {
                 first->prev = newnode;
             }
             l->first = newnode;
             return TRUE;
          }
       }
       return FALSE;
}
```

```
int dllInsertAsLast(DLList *l, void *data)
{
    DLNode *newnode, *last;
    if ( l != NULL ) {
       newnode = ( DLNode *) malloc (sizeof(DLNode));
       if (newnode != NULL ) {
          newnode->data = data;
          newnode->next = NULL;
           If ( l->first != NULL ) {
               last= l->first;
               while (last->next != NULL ) {
                   last = last->next;
                }
                last->next = newnode;
                newnode->prev = last;
            } else {
               l->first = newnode;
               newnode->prev = NULL;
           }
           return TRUE;
        }
     }
     return FALSE;
}
```

Primeiro da lista : ->prev == NULL
Ultimo da lista : ->next == NULL

Criar um newnode
Colocar os dados em newnode
newnode ->next = NULL;
Achar o ultimo (last)
newnode->prev = last;
If (last != NULL) {
    last->next = newnode;
} else {
    l->first = newnode;
}

```
int dllINsertAfterSpec ( DLList *l, void *key, void *data, int (*cmp) (void *, void *))
{
    DLNode *newnode, *spec, *next;
    if ( l != NULL ) {
       If ( l->first != NULL ) {
           spec= l->first;
            while (spec>next != NULL && cmp( key, spec->data) != TRUE) ) {
               spec = spec->next;
            }
             if (cmp( key, spec->data) == TRUE) {
               newnode = ( DLNode *) malloc (sizeof(DLNode));
              if (newnode != NULL ) {
                  newnode->data = data;
                   newnode->prev = spec;
                   next = spec->next;
                   Newnode->next = next;
                   Spec->next = newnode;
                   If (next != NULL) {
                      next->prev = newnode;
                   }
                   return TRUE;
             }
            }
       }
    }
    return FALSE;
}
```

Fazer a operação de inserir após um nó especificado pela chave

int dllINsertAfterSpec (DLList *l, void *key, v
*data, int (*cmp) (void *, void *))

Achar o spec

Criar o newnode

Coloca os dados em newnode

newnode->prev = spec;

Achar o next - Next = spec->next;

Newnode->next = next;

Spec->next = newnode;

Se o next existe - If (next != NULL) {

next->prev = newnode;

```
int dllINsertBeforeSpec ( DLList *l, void *key, void *data, int (*cmp) (void *, void *))
{
    DLNode *newnode, *spec, *prev;
    if ( l != NULL ) {
        If ( l->first != NULL ) {
            spec= l->first;
             while (spec>next != NULL && cmp( key, spec->data) != TRUE) ) {
                spec = spec->next;
             }
              if (cmp( key, spec->data) == TRUE) {
                newnode = ( DLNode *) malloc (sizeof(DLNode));
                if (newnode != NULL ) {
                   newnode->data = data;
                    Newnode->next = spec;
                    Newnode->prev = spec->prev;
                    prev = spec->prev;
                   Spec->prev= newnode;
                   If (prev != NULL) {
                       Prev->next = newnode;
                   } else {
                       l->first = newnode;
                   }
                   return TRUE;
             }
            }
        }
     }
     return FALSE;
}
```

Fazer a operação de inserir antes de um nó especificado pela chave
int dllINsertbeforeSpec (DLList *l, void *key, void *data, int (*cmp) (void *, void *))

Achar o spec
Criar newnode
Colocar os dados em newnode

```
Newnode->next = spec;
Newnode->prev = spec->prev;
prev = spec->prev;
Spec->prev= newnode;
If (prev != NULL) {
   Prev->next = newnode;
} else {
   l->first = newnode;
}
```

```
Void * dllRemoveSpec ( DLList *l, void *key, int (*cmp) (void *, void *))
{
    DLNode *newnode, *spec, *prev, *next; void *data;
    if ( l != NULL ) {
        If ( l->first != NULL ) {
            spec= l->first;
             while (spec>next != NULL && cmp( key, spec->data) != TRUE) ) {
                spec = spec->next;
             }
              if (cmp( key, spec->data) == TRUE) {
                 data = spec->data;
                  next = spec->next;
                  prev = spec->prev;
                  if ( prev != NULL) {
                     prev->next = next;
                  } else {
                     l_>first = next;
                  }
                  if (next != NULL ) {
                      next->prev = prev;
                  }
                  free (spec);
                 return data;
              }
             }
        }
     }
     return FALSE;
}
```

Achar o spec
Salvar os dados
Next = spec->next
Prev = spec->prev

```
If ( prev != NULL) {
    prev->next = next;
} else {
   l->first = next;
}
If ( next != NULL) {
    Next->prev = prev;
}
Free(spec)
```

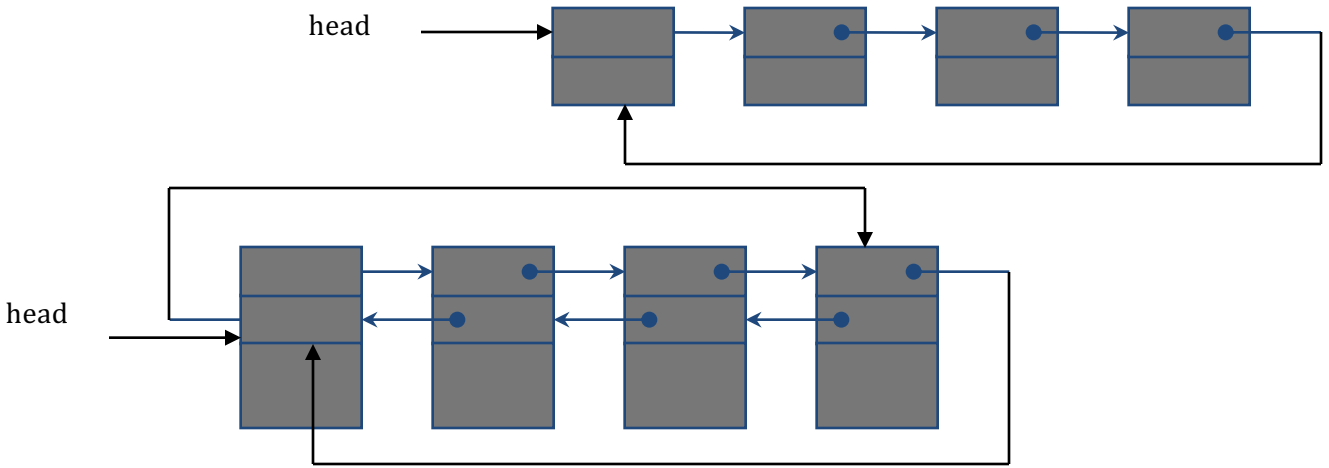
```
Void * dllGetFirst (DLList *l)
{
    if ( l!= NULL) {
        l->cur = l->first;
        return l->cur;
    }
    return NULL;
}
```

```
Void * dllGetNext (DLList *l)
{
    if ( l!= NULL) {
        if ( l->cur != NULL) {
            l->cur = l->cur->next;
             return l->cur;
        }
    }
    return NULL;
}
```

a) Fazer a operação de inserir antes do anterior a um nó especificado pela chave
int dllINsertBefBefSpec (DLList *l, void *key, void *data, int (*cmp) (void *, void *))
b) Fazer a operação de remover após um nó especificado pela chave (se existir, senão insere após o spec)
int dllRemoveAfterSpec (DLList *l, void *key, void *data, int (*cmp) (void *, void *))

Listas Encadeadas Circulares

- O último elemento recebe o endereço do primeiro (Lista simplesmente encadeada)
- O campo next do último elemento aponta para o primeiro e o campo previous do primeiro aponta para o último



Listas Duplamente Encadeadas

- O que muda??
  - A forma de identificar o último elemento da lista - agora é o elemento cujo campo next é igual a head.
  - Não precisa entrar na lista somente pelo head, agora basta saber o endereço de um Node e pode percorrer a lista. Nesse caso para qdo o campo next for igual ao Node fornecido
- Exercícios
  - Escreva os algoritmos para as seguintes operações em listas circulares duplamente encadeadas
    - Busca de elemento
    - Remoção de um elemento
    - Inserção após um elemento.