Um pouco de análise

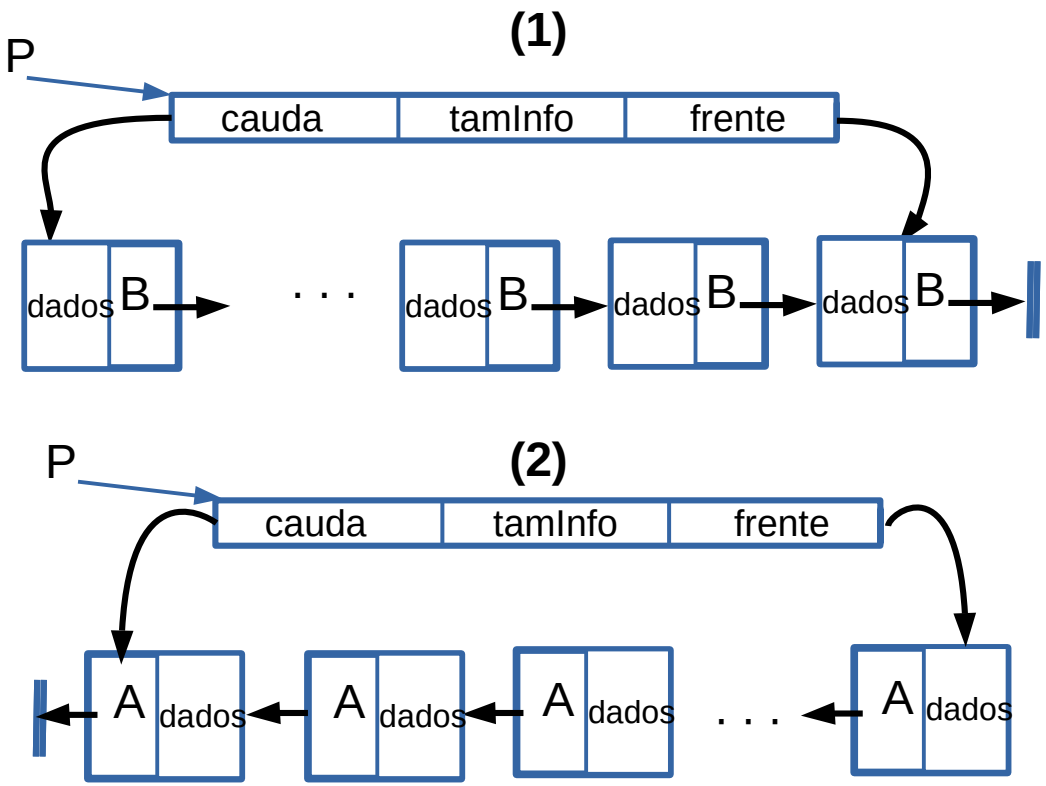
- É importante estar atento ao desempenho das operações servidas pela estrutura de dados;
- Iremos analisar algumas situações para a LSE e LDE tomando como base o clássico algoritmo de busca sequencial.

Um pouco de análise

Pequenas alterações podem gerar resultados surpreendentes no desempenho das operações do objeto “estrutura de dados”;

Ao comparar as duas FSEs ao lado, notamos que uma alteração aparentemente inócua (no sentido do encadeamento) gerou uma diferença significativa na eficiência da remoção:

- A remoção de dados da FSE-1 utiliza um laço, o qual adiciona passos extras e desnecessário quando se trata da FSE-2.



Um pouco de análise

- Iremos analisar algumas situações para a LSE e LDE tomando como base o clássico algoritmo de busca sequencial;
- Dessa vez utilizaremos a busca sequencial como aplicação sobre diferentes “desenhos” de listas e verificar como esses “ desenhos” afetam a busca sequencial.

Busca Sequencial

Sobre um vetor desordenado

- Para cada célula do vetor, se o item procurado não ocorre na célula *i*, visite a célula *i+1*, até que o item seja encontrado ou o vetor seja totalmente vasculhado determinando a ausência do item.

Sobre um vetor ordenado

- Para cada célula do vetor, se o item procurado não ocorre na célula *i*, visite a célula i+1, até que o item procurado seja encontrado ou que se determine a sua ausência, a qual fica caracterizada quando a busca sequencial encontra um elemento de ordem maior que a do item procurado ou se o vetor for totalmente vasculhado sem que se encontre o item procurado.

Link para video sobre busca sequencial (busca linear) e busca binária.

Busca Sequencial

Qual seria a implementação de uma função de aplicação que realiza a Busca Sequencial para a Lista Simplesmente Encadeada (LSE) contendo dados desordenados?

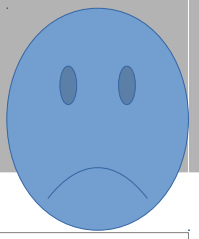
```
int buscaSeq(struct                    eg, chave CPF)
{       struct NoLDSE

        while (aux!=NU                  != CPF)
                aux=aux->
        if (aux)
        {  memcpy(reg,                 >tamInfo);
           return SUCE
        }
        return FRACAS
}
```

```
int buscaSeq(struct                    reg, chave CPF)
{
        int i=1;

        while(buscaNa                  ==SUCESSO)
            if(reg->id ==

                                       SO;

        return FRACAS
}
```

Busca Sequencial

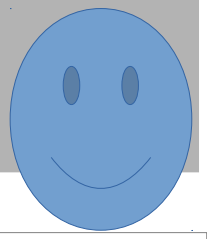
No caso da Lista Simplesmente Encadeada contendo dados desordenados, qual seria a implementação de uma função de aplicação que realiza a Busca Sequencial?

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       struct NoLDSE *aux=p->inicio;

        while (aux!=NULL && aux→id != CPF)
                aux=aux->prox;
        if (aux)
        {memcpy(reg,&(aux->dados), p->tamInfo);
        return SUCESSO;
        }
        return FRACASSO;
}
```



Essa função de aplicação, **não** atende às restrições de encapsulamento

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{
        int i=1;

        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```



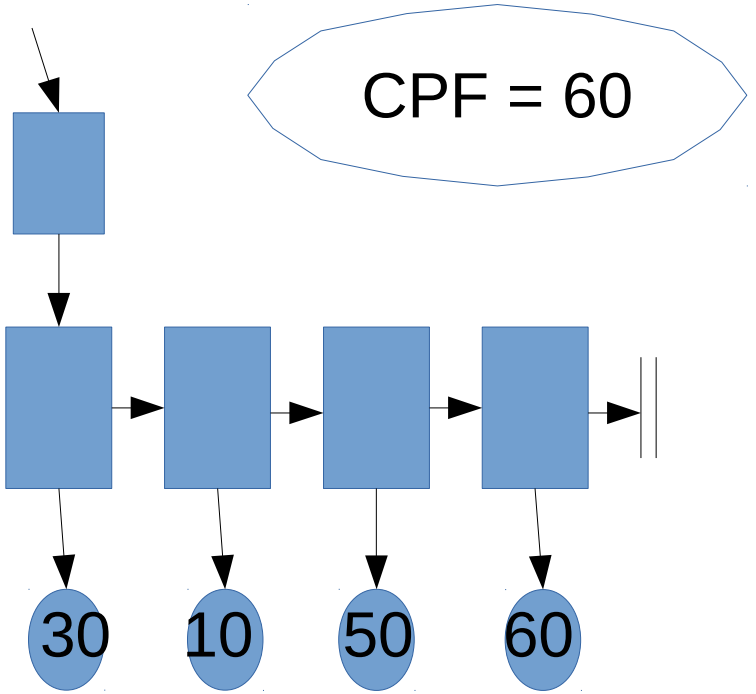
Essa outra versão da função de aplicação **atende** às restrições de encapsulamento

Busca Sequencial

Vamos analisar essa aplicação sobre a LSE contendo dados desordenados, na busca pelos dados associados ao CPF=60

Sucessivas chamadas a buscaNaPosLog( )

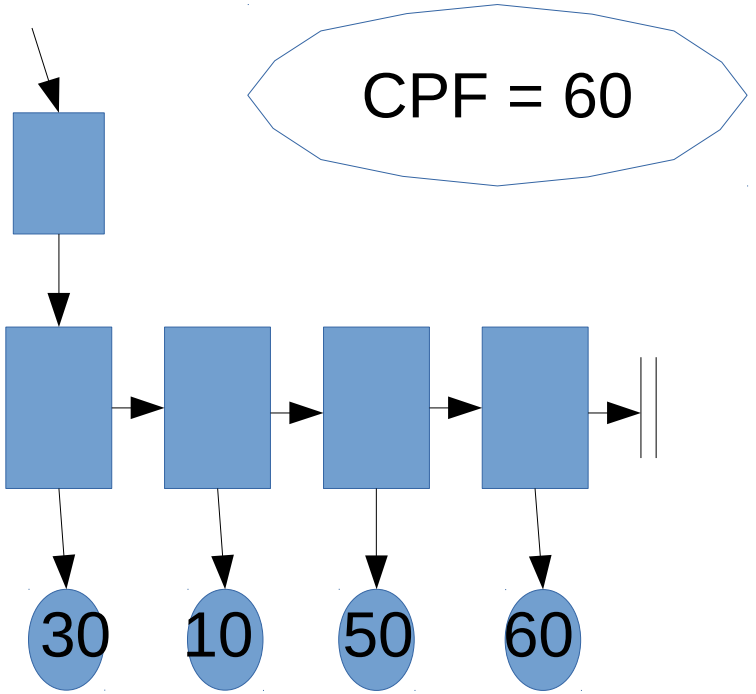
```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{
        int i=1;

        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```



Busca Sequencial

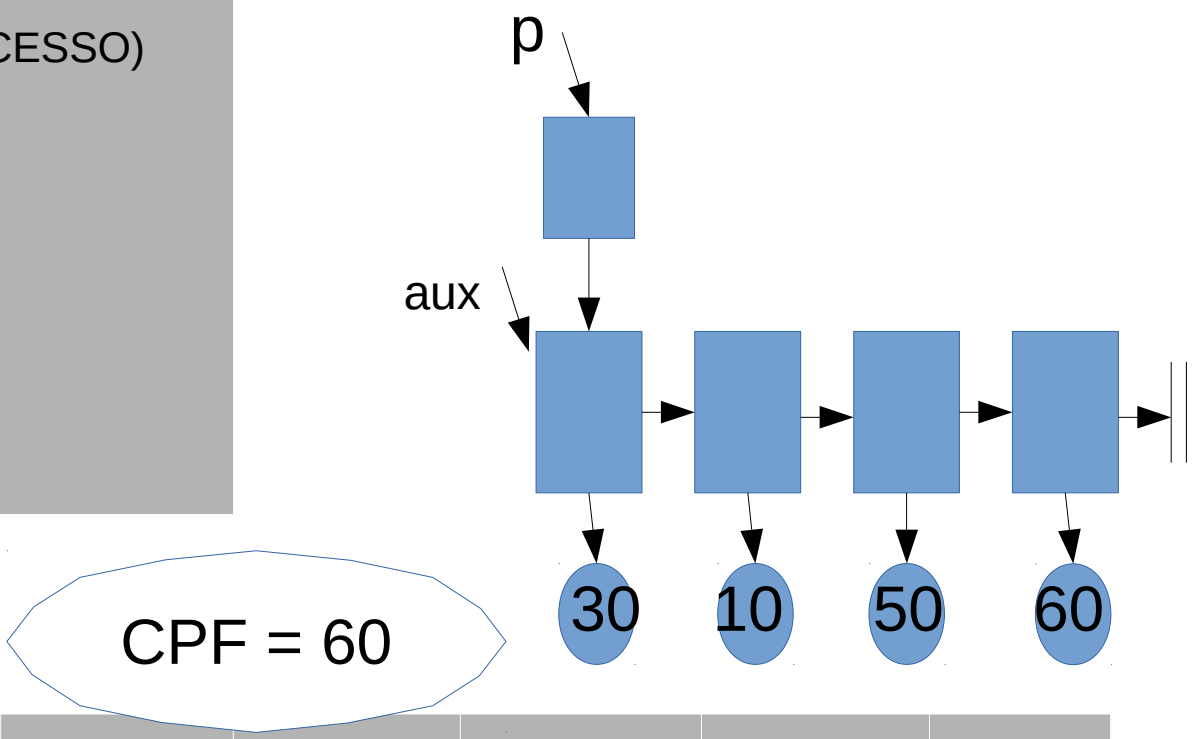
LSE contendo dados desordenados, busca pelos dados associados ao CPF=60

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{
        int i=1;

        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```



A cada chamada, buscaNaPoslog(…) utiliza um ponteiro auxiliar que percorre a lista a partir de p->inicio

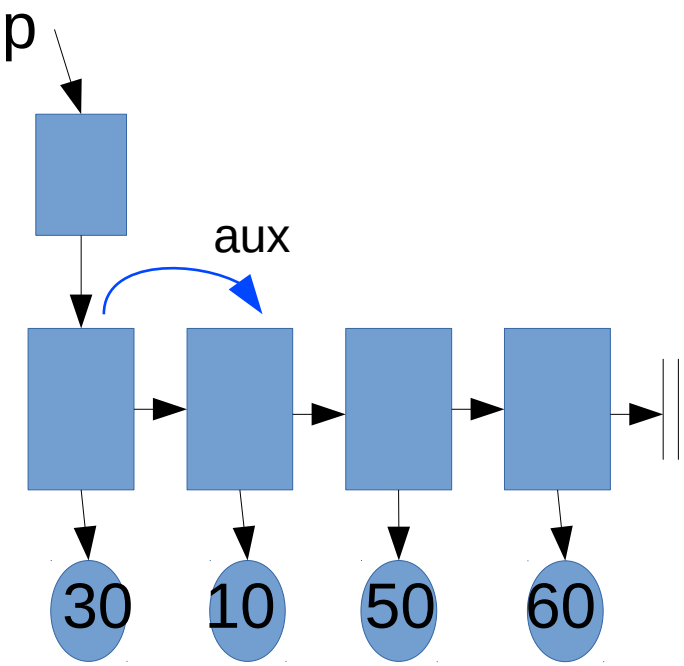
```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```



	i=1	i=2	i=3	i=4	
	1º chamada	2º chamada	3º chamada	4º chamada	Total
saltos do “aux” entre nós de dados	0				

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```

A cada chamada, *aux* percorre a lista a partir de *p->inicio*

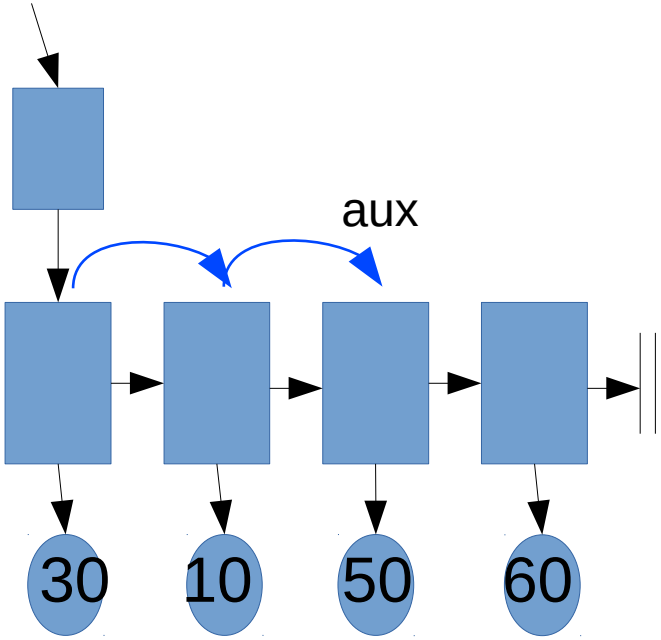


CPF = 60

	i=1	i=2	i=3	i=4	
	1º chamada	2º chamada	3º chamada	4º chamada	Total
saltos do “aux” entre nós de dados	0	1			

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```

A cada chamada, *aux* percorre a lista a partir de *p->inicio*

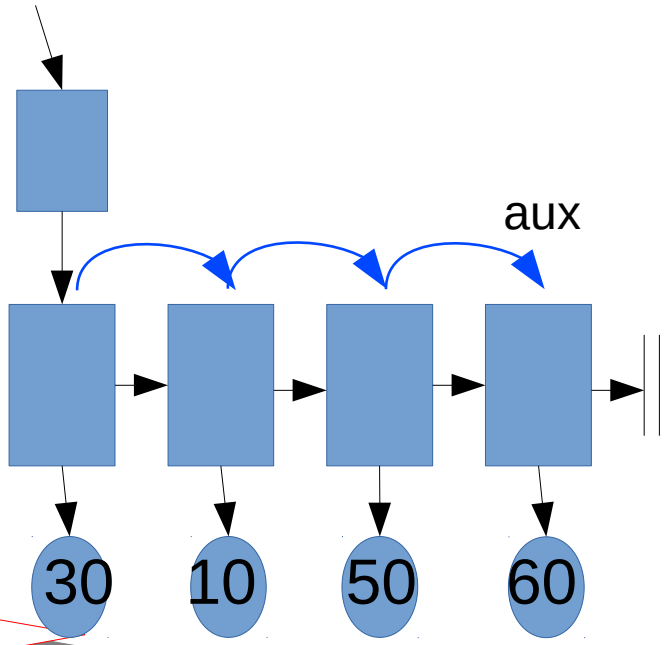


CPF = 60

	i=1	i=2	i=3	i=4	
	1º chamada	2º chamada	3º chamada	4º chamada	Total
saltos do “aux” entre nós de dados	0	1	2		

A cada chamada, *aux* percorre a lista a partir de *p->inicio*

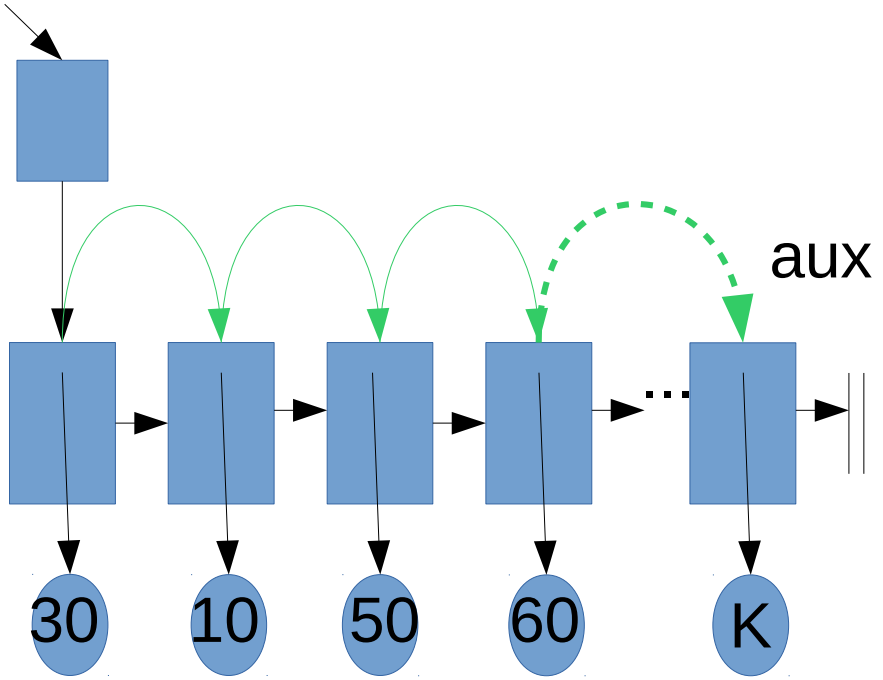
```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
               i++;
        }
       return FRACASSO;
}
```



Encontrou CPF = 60

	i=1	i=2	i=3	i=4	
	1º chamada	2º chamada	3º chamada	4º chamada	Total
saltos do “aux” entre nós de dados	0	1	2	3	6

```
int buscaSeq(struct LSE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {     if(reg->id == CPF)
                        return SUCESSO;
              i++;
        }
       return FRACASSO;
}
```



CPF = K

Para um tamanho M significativamente grande:
Total de saltos=M^2
O "comportamento" da função de aplicação é quadrático: O(M^2)

Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total: soma de uma PA
saltos do "aux" entre nós de dados	0	1	2	3	...	M-1	$\sum \approx M^2$

Podemos aprimorar o desempenho geral utilizando variações de LDEs, vamos analisar três delas:

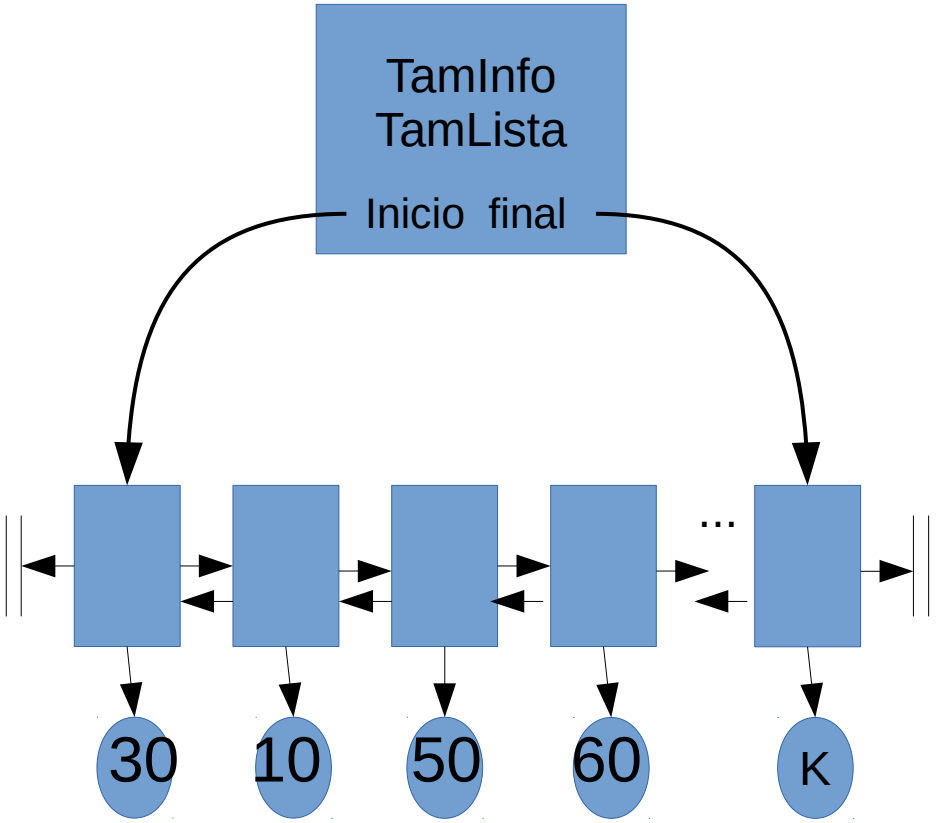
1. LDE referenciando as extremidades;

2. LDE com referencial flexível (“móvel”) para a lista;

3. LDE circular e com referencial flexível (“móvel”) para a lista.

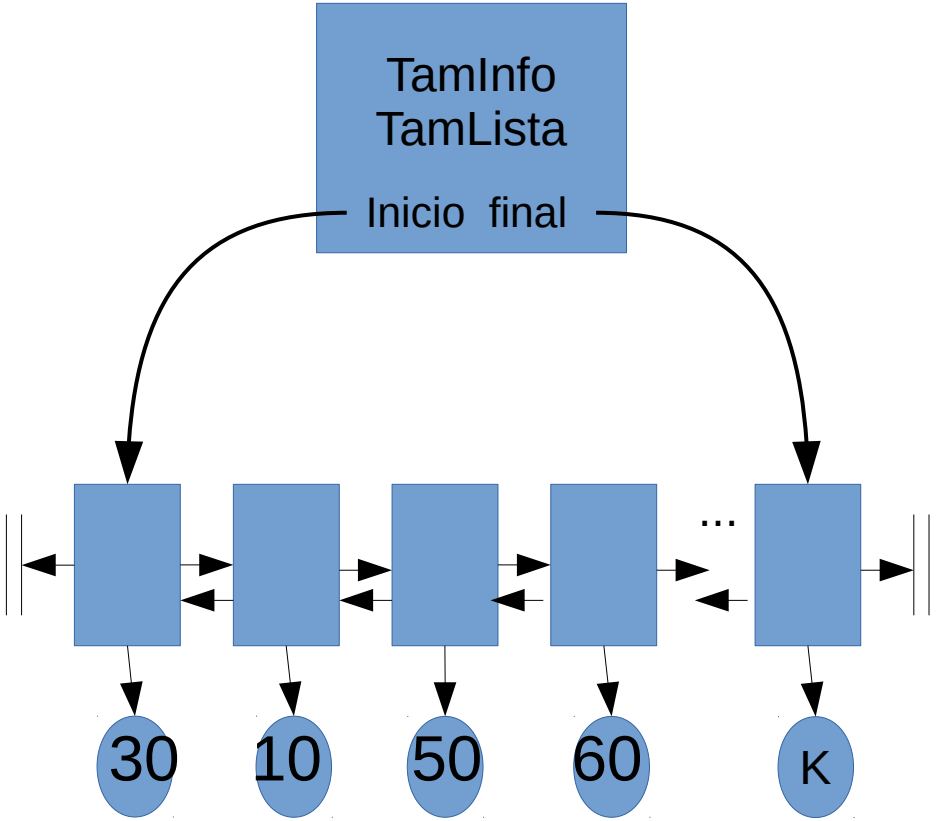
Variações da LDE: referenciando as extremidades

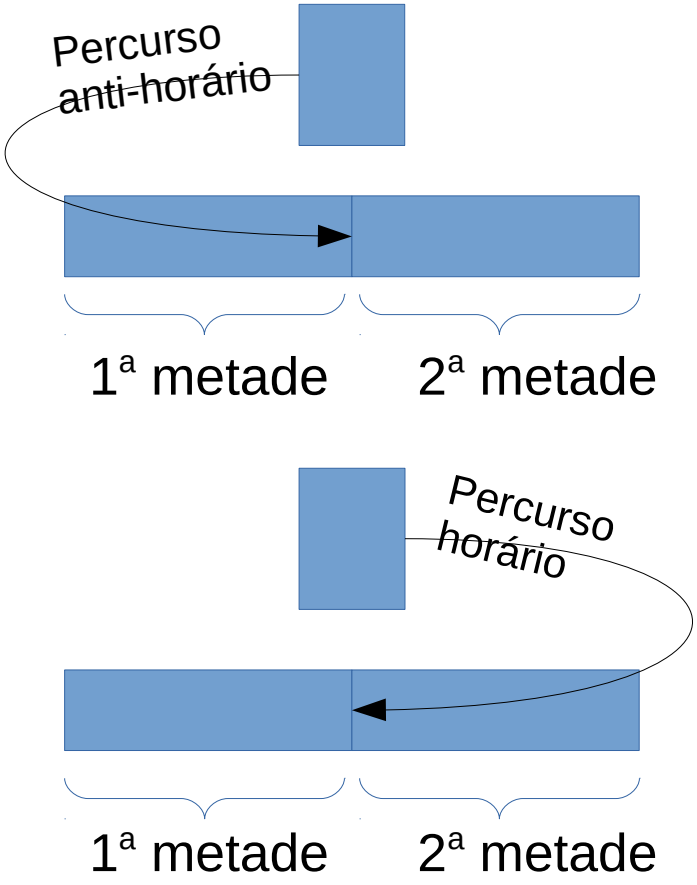
Podemos aprimorar o desempenho utilizando variações de LDEs. Por exemplo: descritor contendo referências para o início e o final da sequência, provendo duas possíveis direções para uma busca.



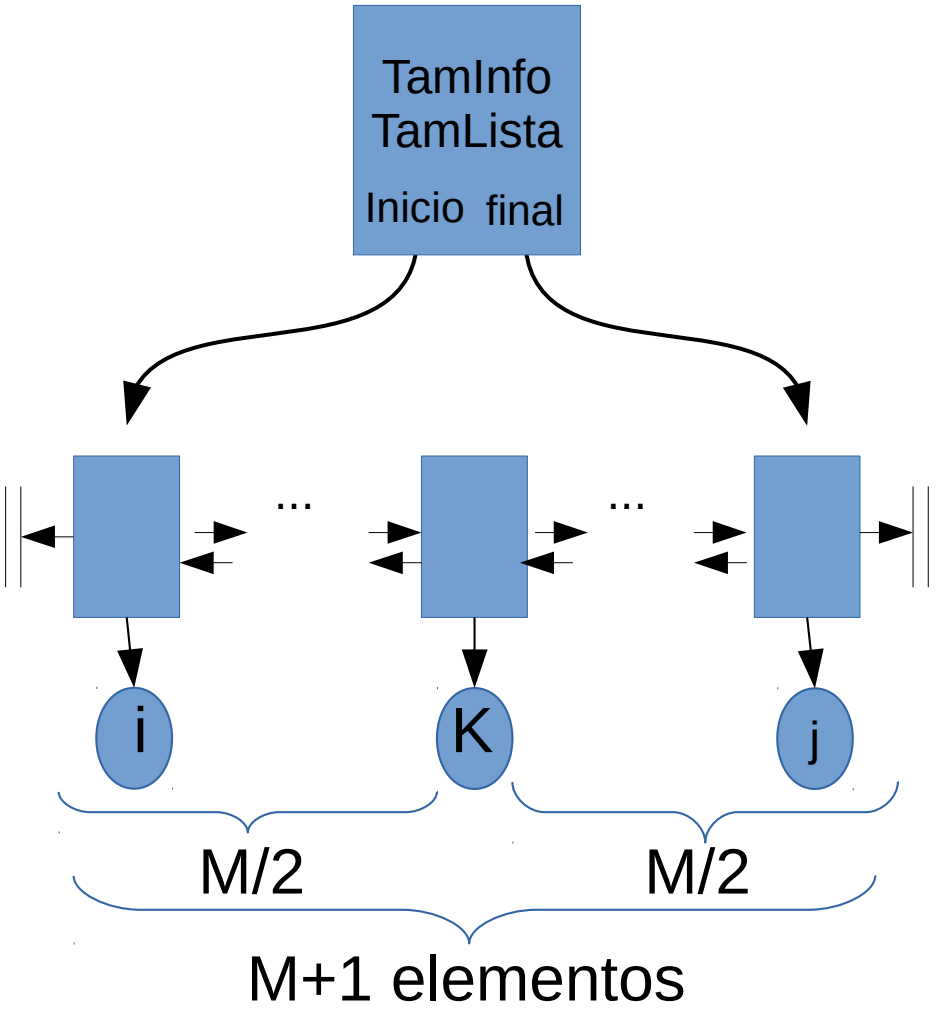
Variações da LDE: referenciando as extremidades

Aprimorando: as duas possíveis direções para a busca são limitadas pela metade do tamanho da lista. Se a posição-alvo da busca ocorrer na primeira metade, o menor percurso será o anti-horário, caso contrário será o percurso horário.





```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {        if(reg->id == CPF)
                 {    memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                 }
                 i++;
        }
        return FRACASSO;
}
```



CPF = K, K no centro da lista.

O custo total de saltos do “aux” para acesso ao nó independente do percurso será a metade do total do caso anterior (busca na LSE):

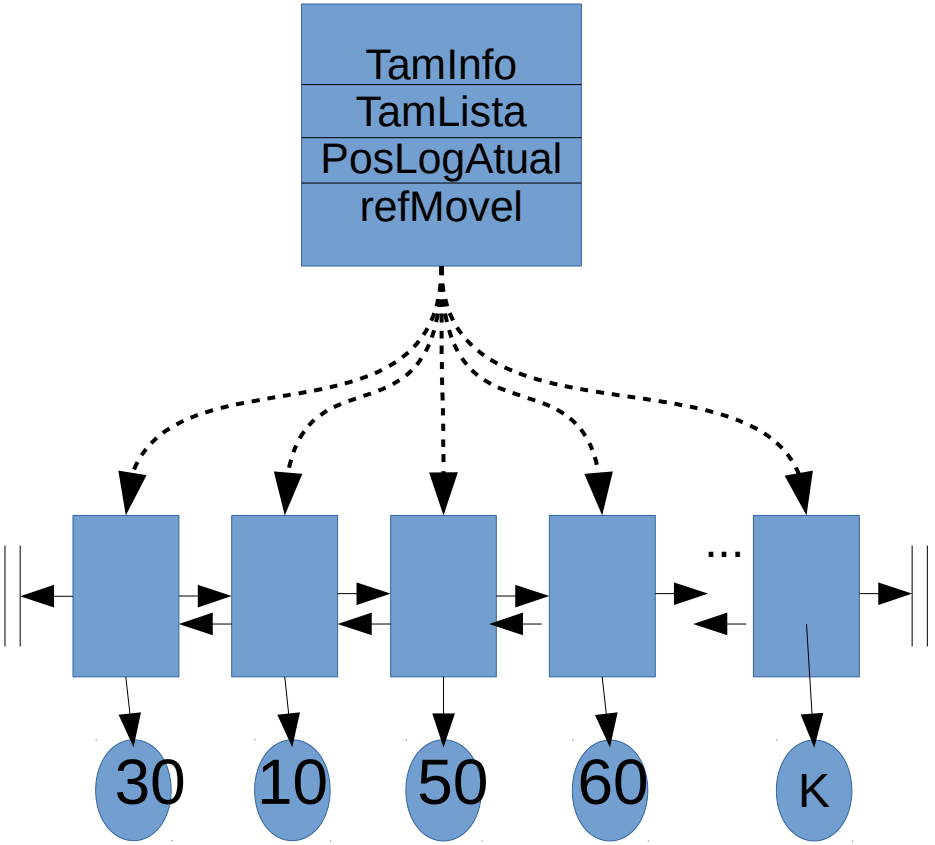
Total de saltos=$M^2/2$

Comportamento quadrático: $O(M^2)$

Variações da LDE: LDE refMovel

Podemos aprimorar o desempenho utilizando variações de LDEs.

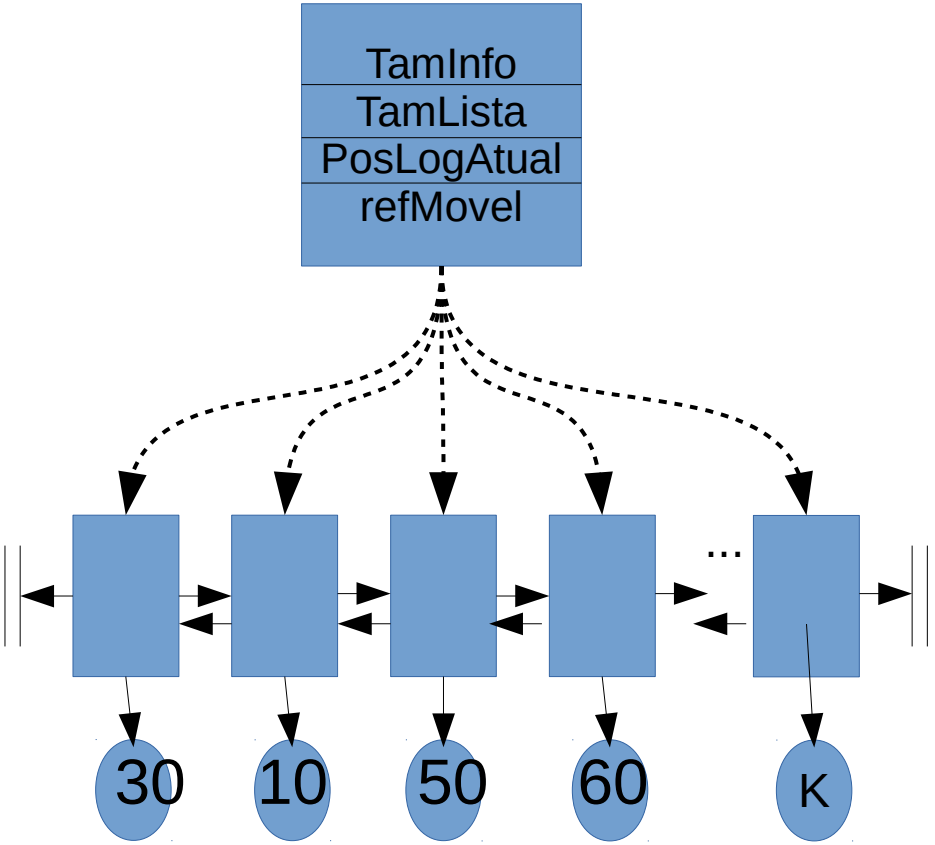
Por exemplo: descritor possui uma referência móvel (refMovel) para um nó qualquer. Interessante quando se necessita acesasr posições em sequência.



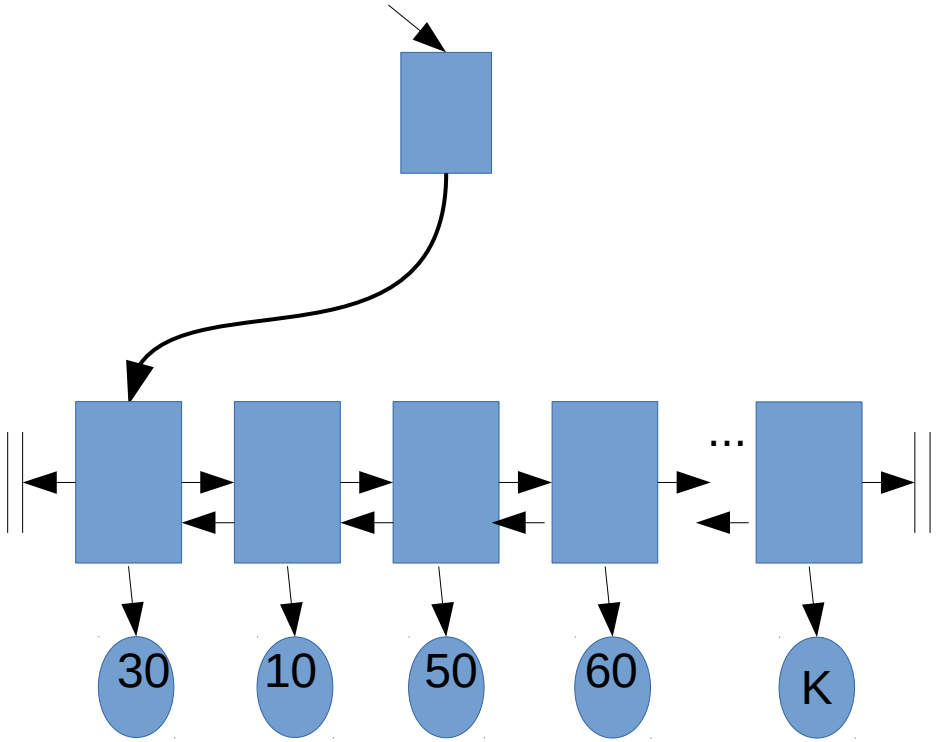
Variações da LDE: LDE refMovel

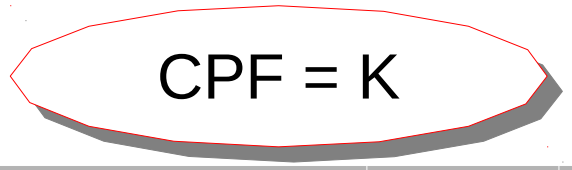
Podemos aprimorar o desempenho utilizando variações de LDEs.

- RefMovel: estará no endereço do elemento recém-buscado ou recém-inserido ou registra o endereço do vizinho do removido. Se a lista estiver vazia ele será aterrado (NULL);
- PosLogAtual: anota a posição sequencial do nó apontado por refMovel



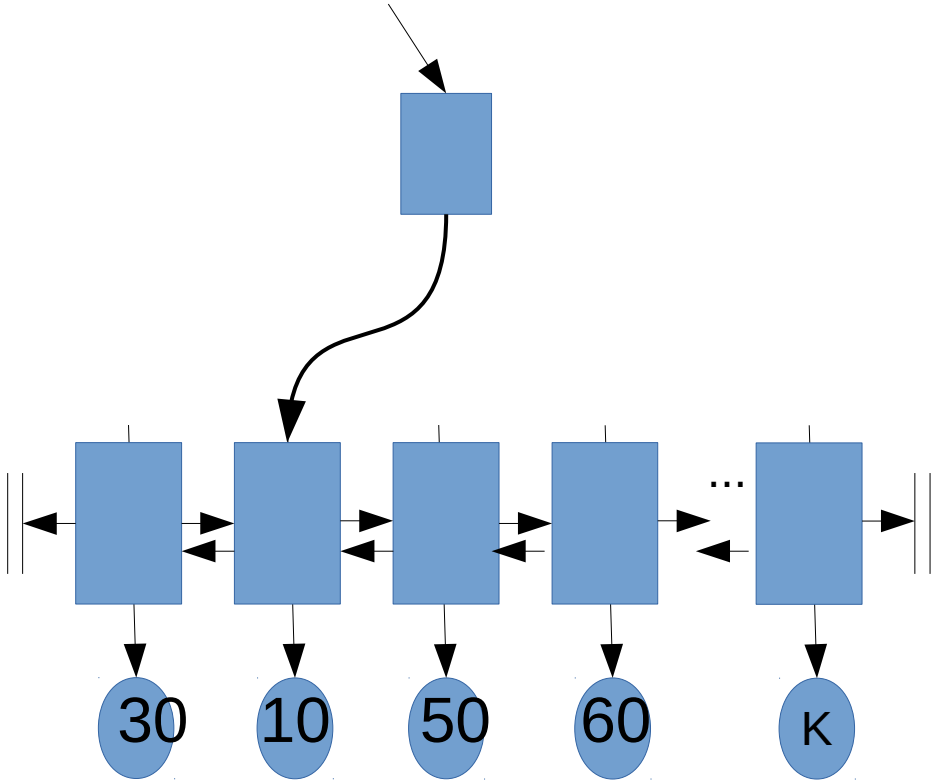
```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {     memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```

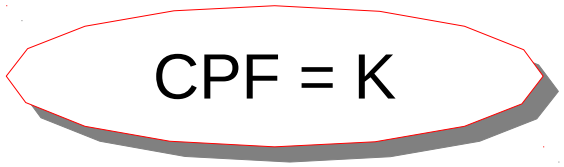




Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	.. .	M-ésimo acesso	Total: soma de uma PA
saltos do “refMovel” entre nós de dados	0				.. .		

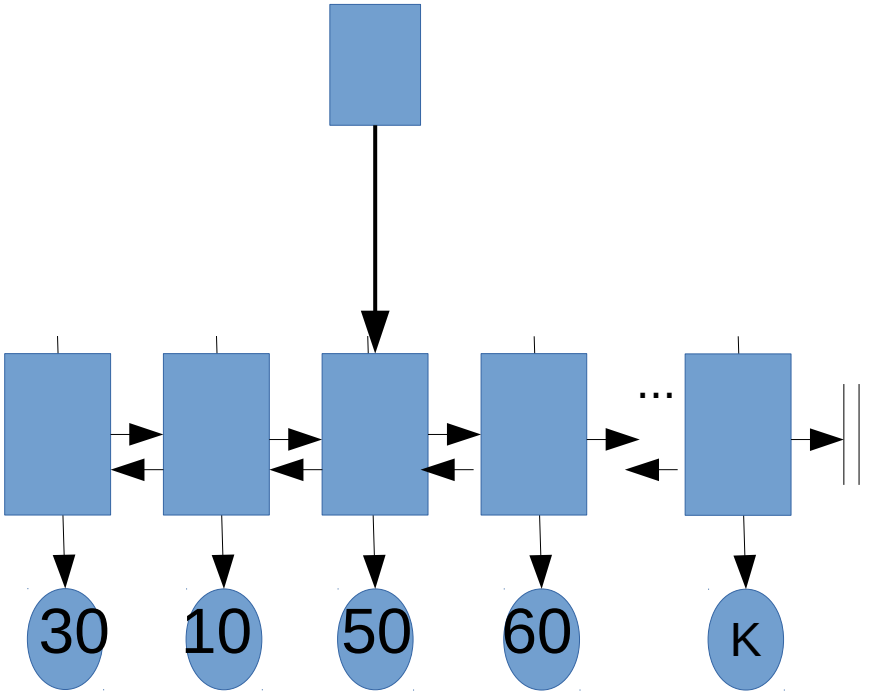
```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {     memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```

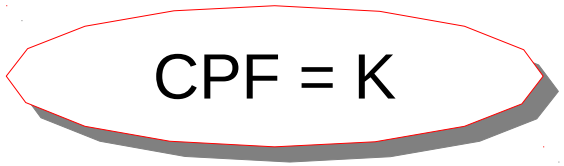




Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total:
saltos do "refMovel" entre nós de dados	0	1			...		

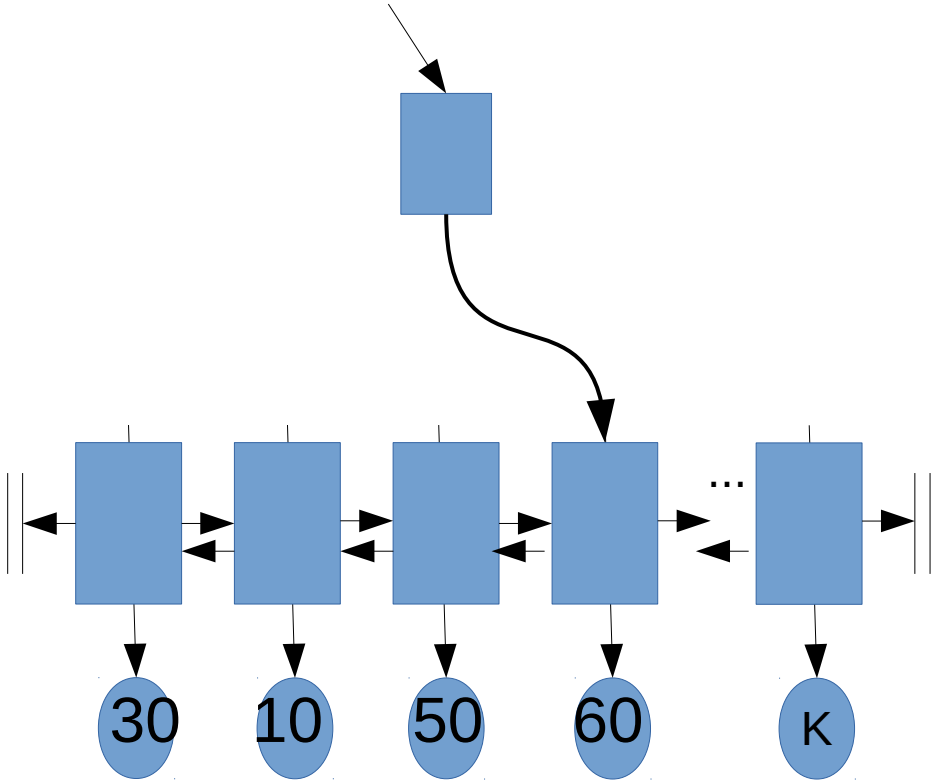
```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {       memcpy(reg,&(aux->dados), p->tamInfo);
                        return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```

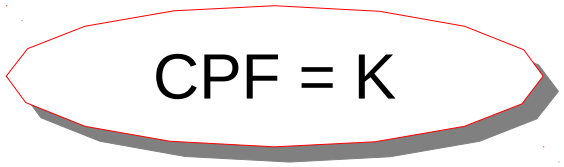




Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total:
saltos do “refMovel” entre nós de dados	0	1	1		...		

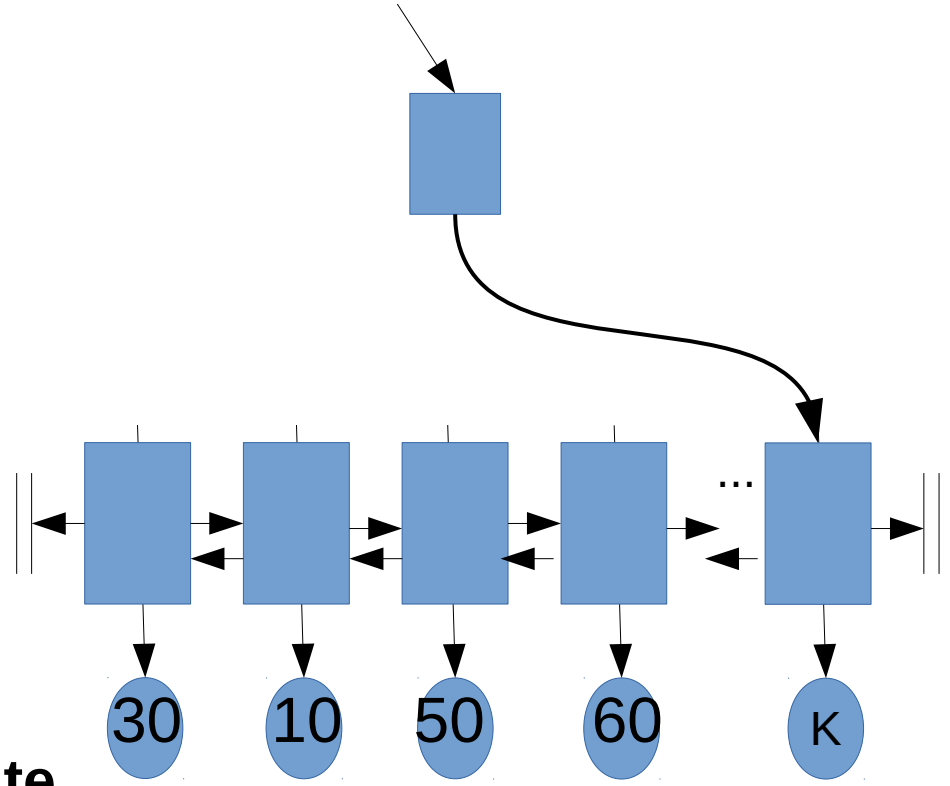
```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {     memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```





Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total:
saltos do "refMovel" entre nós de dados	0	1	1	1	...		

```
int buscaSeq(struct LDE *p,  info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {     memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```

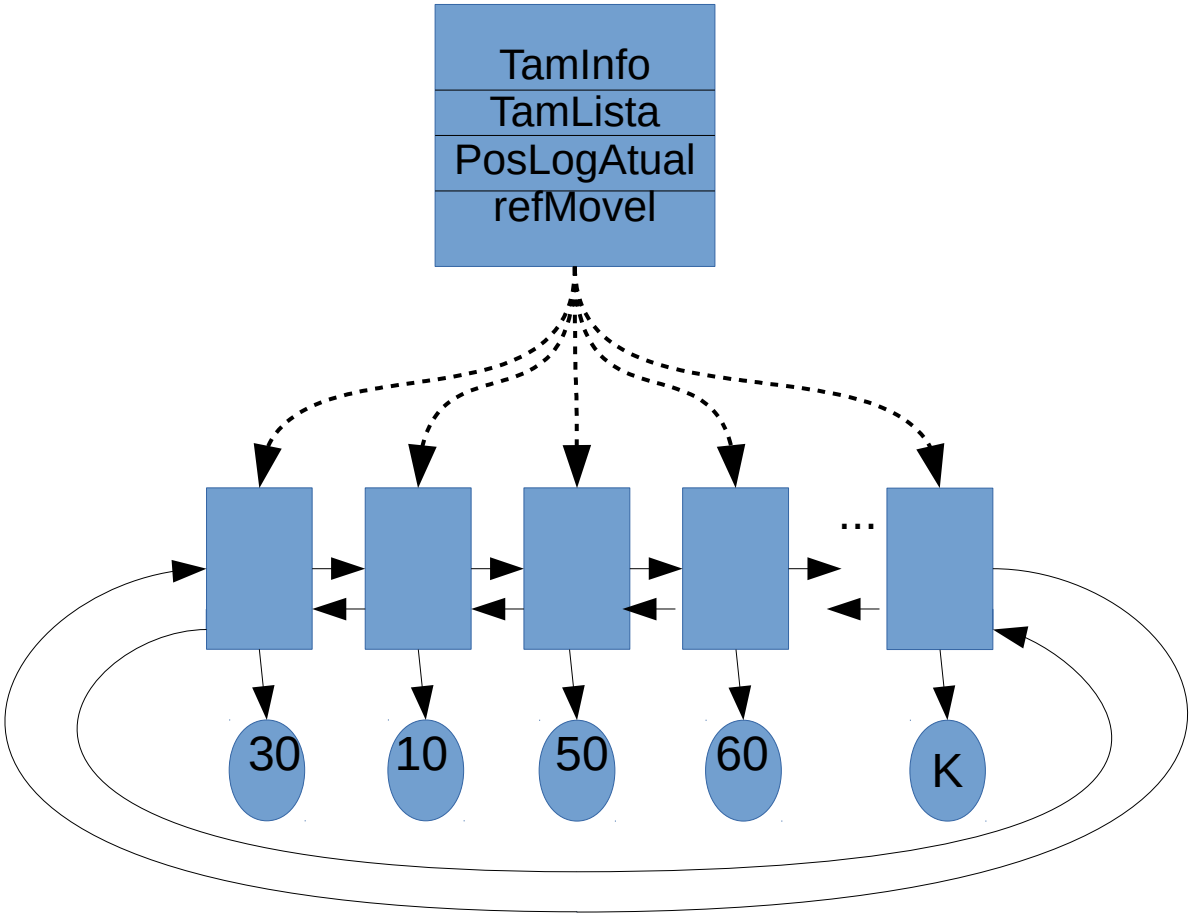


Para um tamanho M significativamente grande:
Total de saltos=M
Comportamento linear: O(M)

Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total:
saltos do "refMovel" entre nós de dados	0	1	1	1	...	1	M

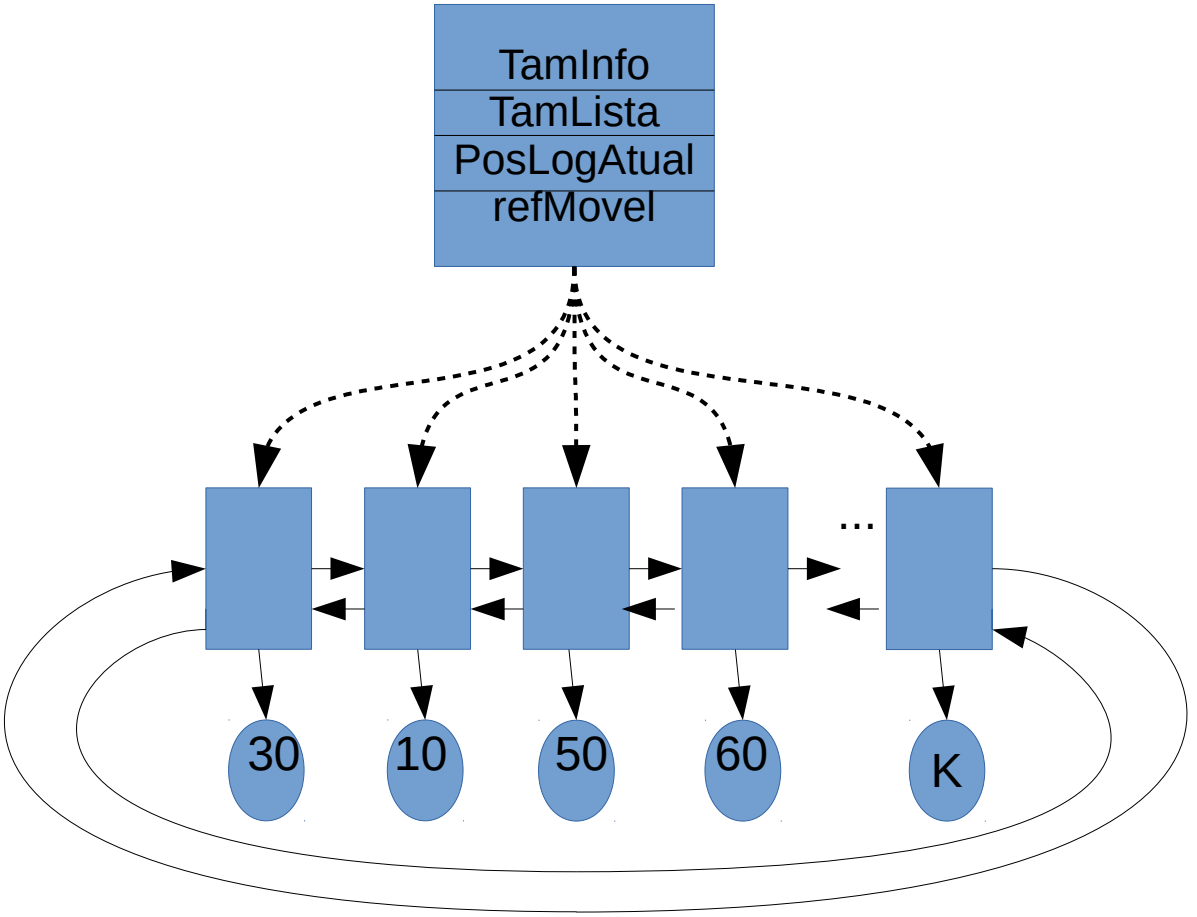
Variações da LDE: LDE Circular com refMovel

Podemos aprimorar o desempenho utilizando variações de LDEs. Por exemplo: descritor possui uma referência móvel para um nó qualquer e a lista é circular.



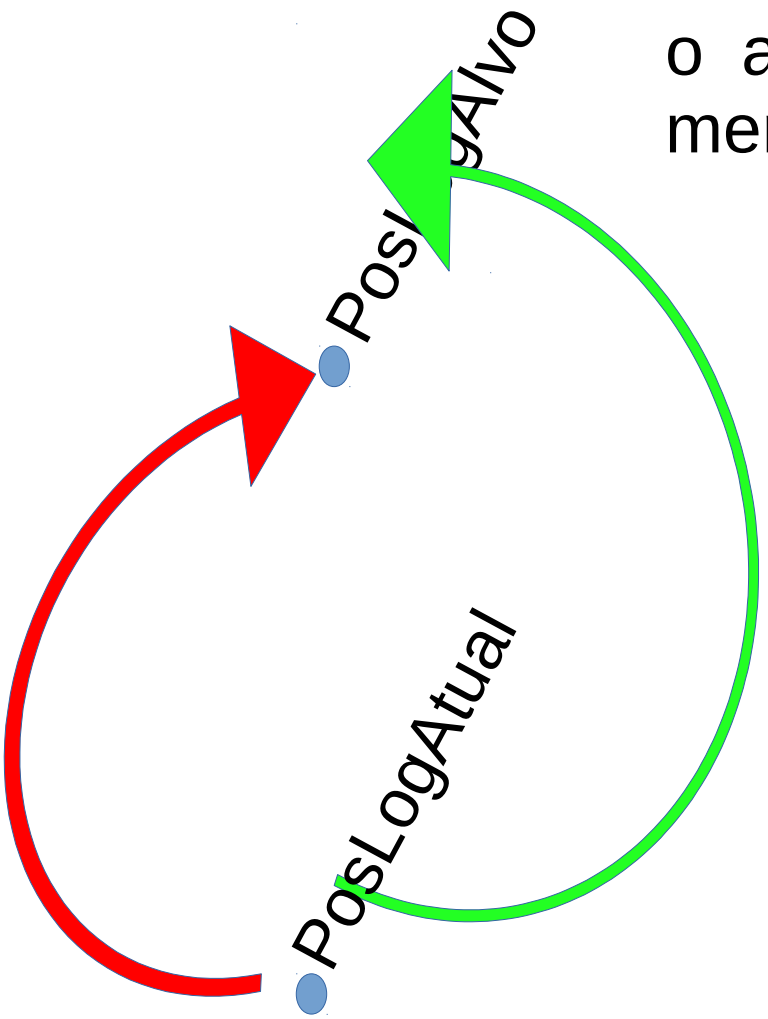
Variações da LDE: LDE Circular com refMovel

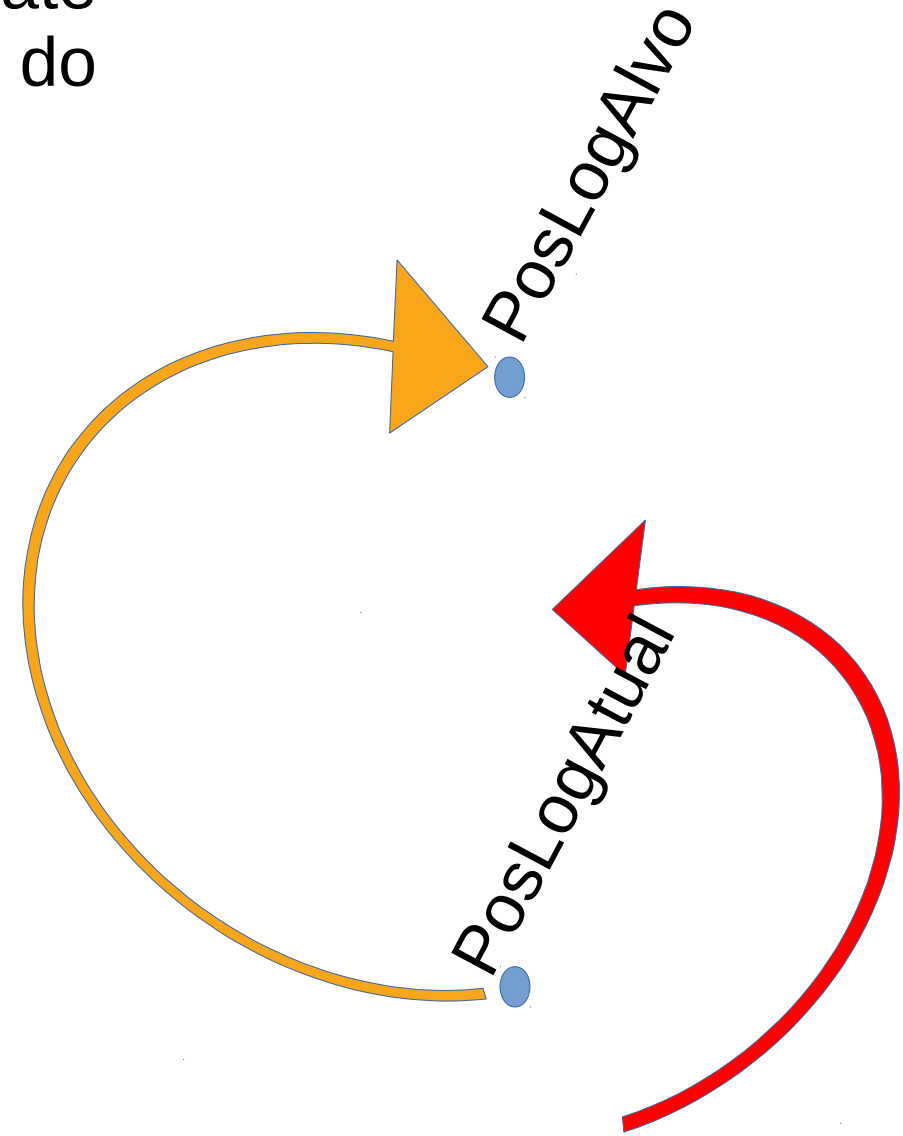
RefMovel: estará no endereço do elemento recém-buscado ou recém-inserido ou no vizinho do removido. Se a lista estiver vazia ele será aterrado (NULL);
PosLogAtual: anota a posição sequencial do nó apontado por refMovel



Variações da LDDE: LDDE Circular com refMovel

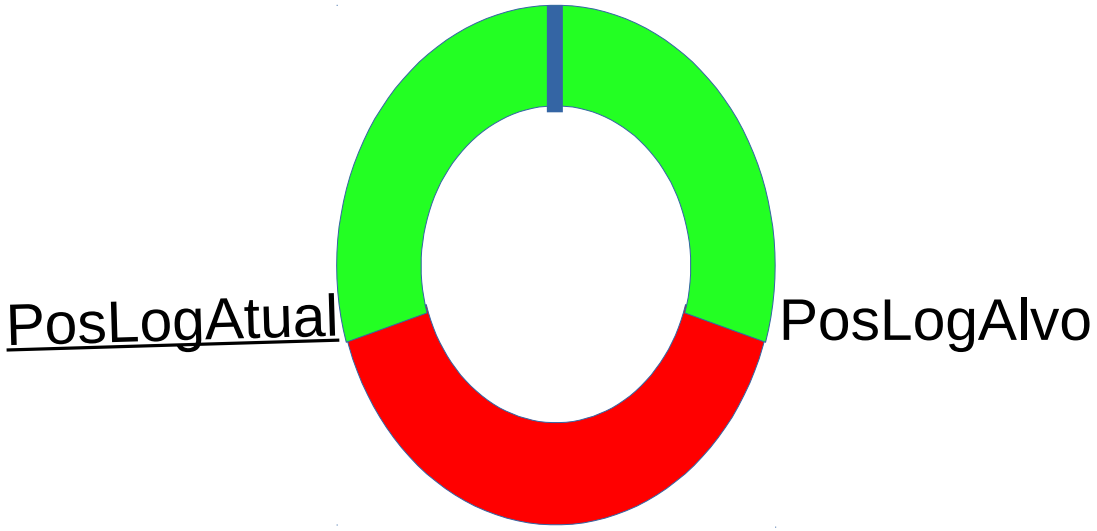
O deslocamento até o alvo depende do menor caminho.

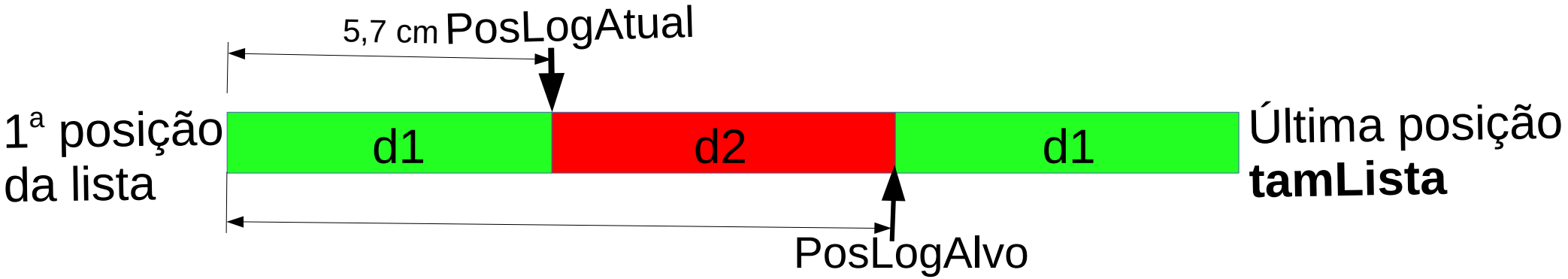




Variações da LDDE: LDDE Circular com refMovel

A) PosLogAtual < PosLogAlvo

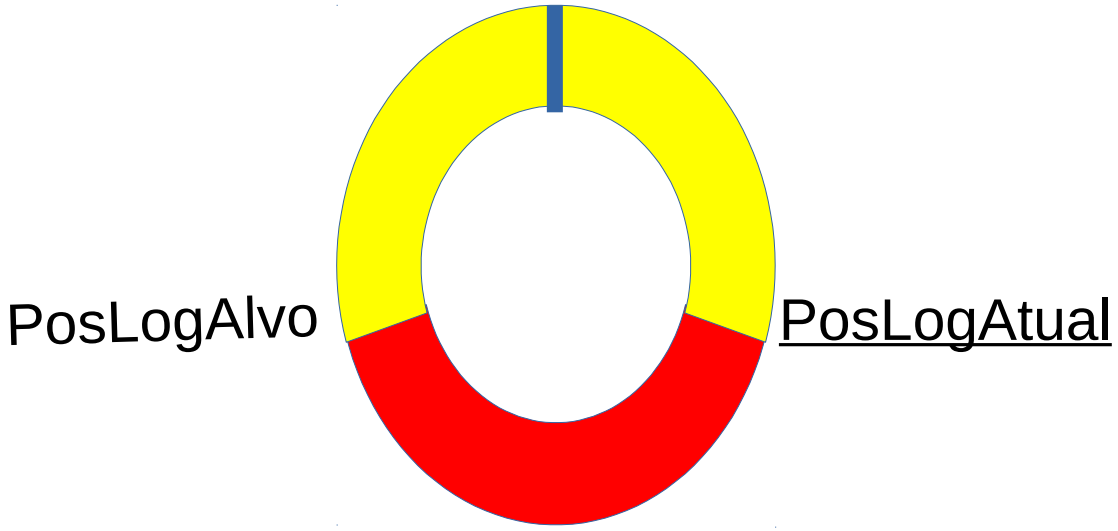


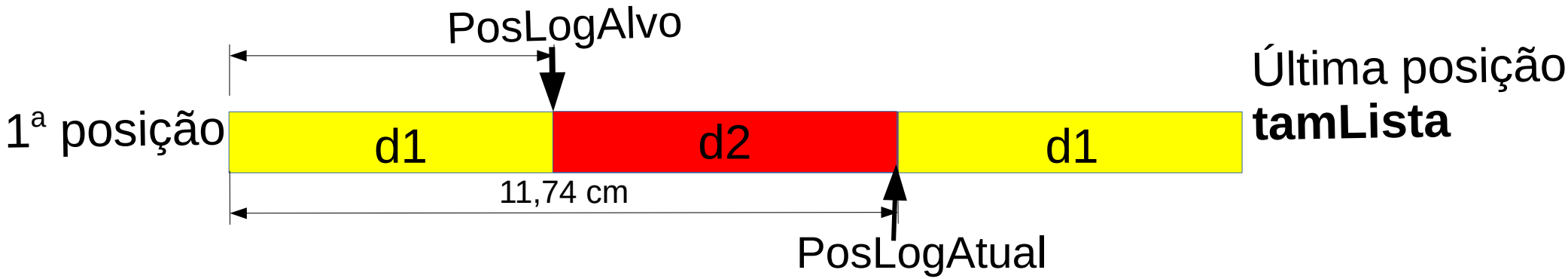


d1= PosLogAtual + (TamLista-PosLogAlvo)
d2= PosLogAlvo - PosLogAtual

Variações da LDDE: LDDE Circular com refMovel

B) PosLogAtual > PosLogAlvo



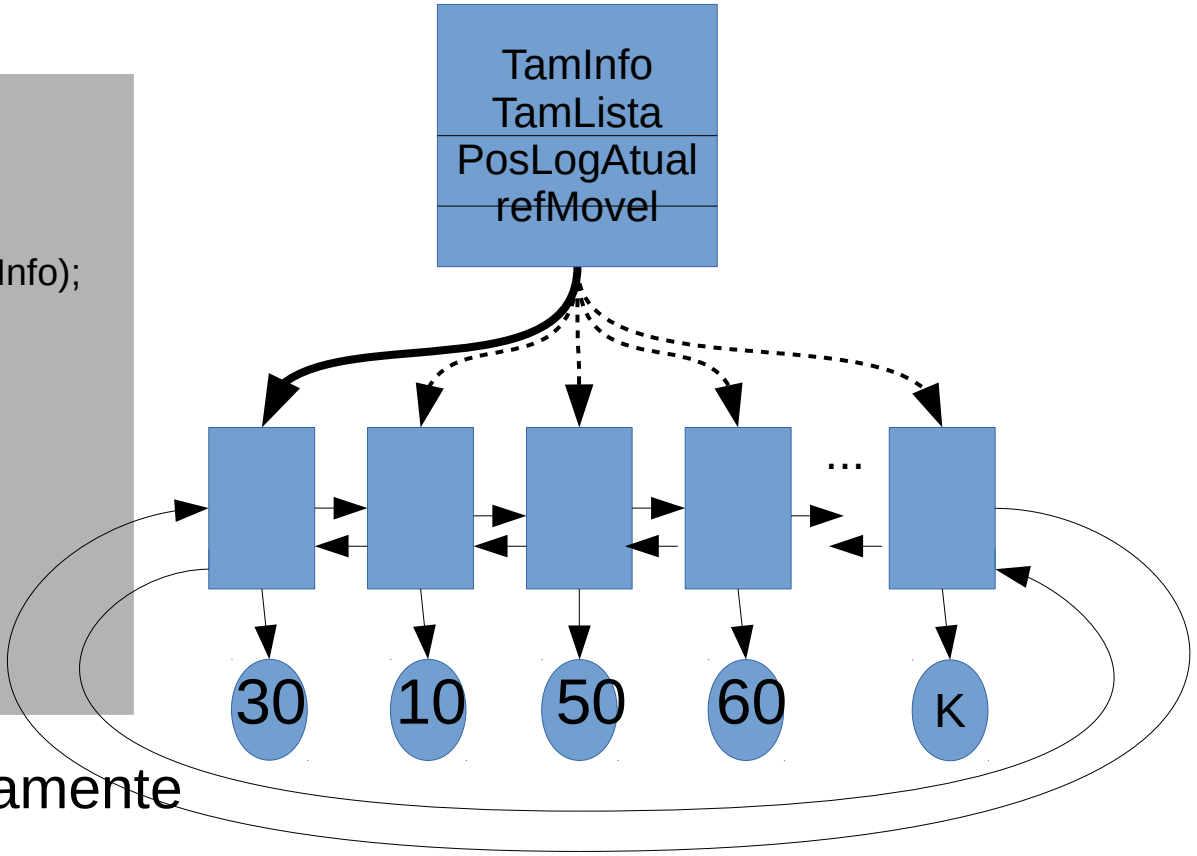


d1= PosLogAlvo + (TamLista-PosLogAtual)
d2= PosLogAtual - PosLogAlvo

1 ≤ PosLogAtual ≤ tamLista e *1 ≤ PosLogAlvo ≤ tamLista.*
Percurso pelo menor caminho da *posLogAtual* até a *PosLogAlvo*

```
SE (PosLogAtual < PosLogAlvo)
    d1= PosLogAtual + (TamLista-PosLogAlvo)
    d2= PosLogAlvo - PosLogAtual
    SE (d1 < d2)
        percorre no sentido horário
    SENAO
        percorre no sentido anti-horário
    FIMSE
SENAO
    d1= PosLogAlvo + (TamLista-PosLogAtual)
    d2= PosLogAtual – PosLogAlvo
    SE (d1 < d2)
        percorre no sentido anti-horário
    SENAO
        percorre no sentido horário
    FIMSE
FIMSE
```

```
int buscaSeq(struct LDE *p, info *reg, chave CPF)
{       int i=1;
        while(buscaNaPoslog(p,reg,i)==SUCESSO)
        {       if(reg->id == CPF)
                {     memcpy(reg,&(aux->dados), p->tamInfo);
                      return SUCESSO;
                }
                i++;
        }
        return FRACASSO;
}
```



Para um tamanho M significativamente grande:
Total de saltos equivale ao caso da LDDE comp refMovel. Total =M
Comportamento linear: O(M)

Para *M* elementos	1º acesso	2º acesso	3º acesso	4º acesso	...	M-ésimo acesso	Total:
saltos do "refMovel" entre nós de dados	0	1	1	1	...	1	M

Busca sequencial com callback

Na fila de prioridade foi utilizada uma função auxiliar para comparação de prioridades na inserção na fila. O endereço dessa função era passada como argumento para a operação de inserção. Poderíamos “importar” essa ideia e adaptar a busca sequencial por uma chave como uma função da interface de uma LDE.

Analise esse caso para a função buscaPorChave(…) a qual é implementada no arquivo da estrutura de dados:

int buscaPorChave(pLista, int(*compara)(info *novo,info *visitado), info *reg)

Busca Binária

Na busca binária uma coleção linear de dados ordenados é pesquisada de maneira que o espaço de busca é reduzido à metade a cada iteração do algoritmo (ao lado).

Discuta as implementações de LDE como base de dados sob uma aplicação de busca binária.

```
binary_search(pLista, CPF):
    esquerda=1, tam=tamanho(pLista)
    direita=tam
    for i=1 to ⌈log2^tam⌉ :
        meio = (esquerda + direita) / 2
        buscaNaPoslog(pLista, &reg)
        if CPF == reg.chave:
            return reg
        elseif CPF > reg.chave:
            esquerda = meio
        else:
            direita = meio
```