Variações de Listas Encadeadas: Pilhas, Filas, Duplas e com Prioridade

Prof. Lilian Berton

São José dos Campos, 2018

Introdução

Algoritmos

- Sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.
- Ex: como montar um aparelho, fazer um bolo, etc.

Estrutura de Dados

- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a **definição de um conjunto de dados que representa a situação real.**
- A seguir, deve ser escolhida a **forma de representar esses dados** (vetores, matrizes, structs, listas, árvores, grafos).

Programas

- Programar é basicamente estruturar dados e construir algoritmos.
- **Programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.**

Tipos de Dados

- **Caracteriza o conjunto de valores a que uma constante pertence**, ou que podem ser assumidos por uma **variável** ou expressão, ou que podem ser gerados por uma **função**.

- Tipos simples de dados são grupos de valores indivisíveis (**como os tipos básicos integer, boolean, char e real**).

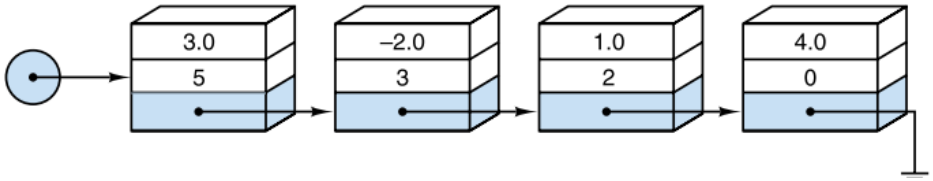
Tipos Abstratos de Dados (TADs)

- TAD é um **modelo matemático, acompanhado das operações definidas sobre o modelo**.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é bastante simples).
  - Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (que pode ser mais complexa).

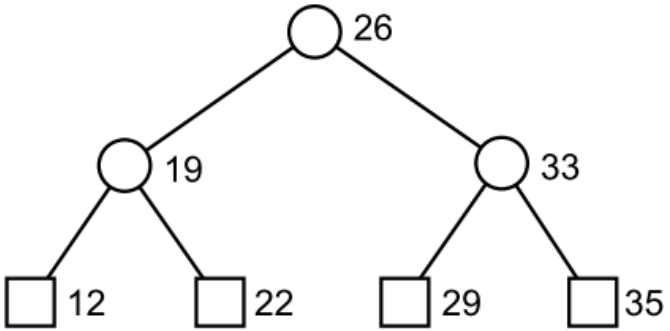
- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma **estrutura de dados.**

Exemplos de TADs

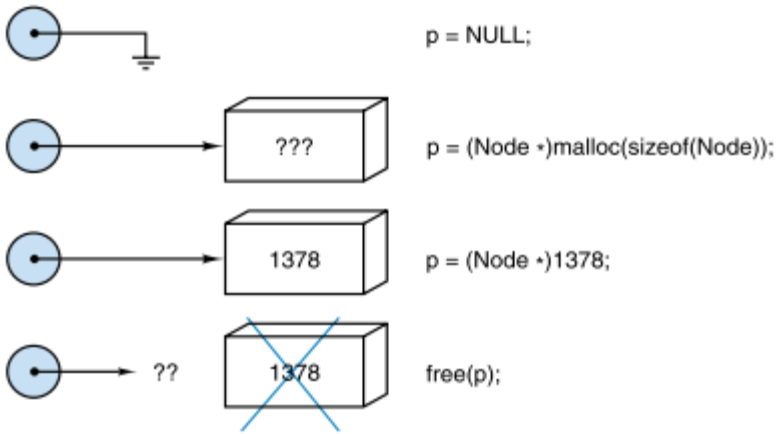
Listas



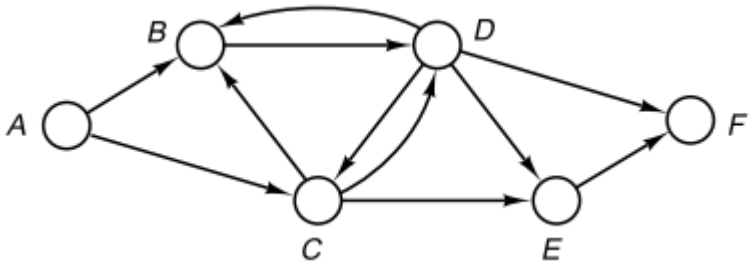
Árvores



Fila



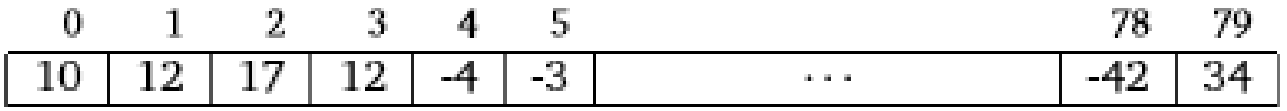
Grafos



Vetores

- **Vetores estáticos**: O tamanho é constante, só mudando a sua declaração é que podemos alterar o seu tamanho. Isso significa que podemos estar "desperdiçando" algum espaço da memória que fica no final do vetor.

- int v[80];



- **Vetores dinâmicos**: permite alterar, durante a execução do programa, o tamanho do bloco de bytes alocado por *malloc*. Nesse caso podemos recorrer a função *realloc* para redimensionar o bloco de bytes (aumentar ou diminuir).

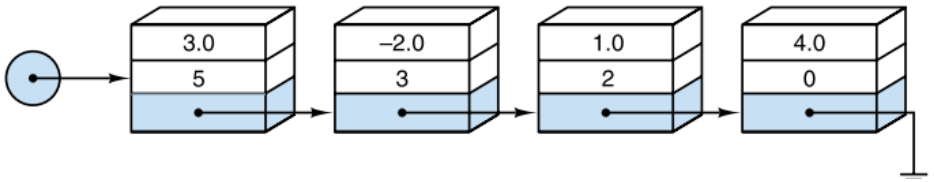
```
int *v;
v = malloc (1000 * sizeof (int));
v = realloc (v, 2000 * sizeof (int));
```

Listas Encadeadas

- Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem), sem precisar mover uma série de elementos.

- Podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda.

- Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido a priori).

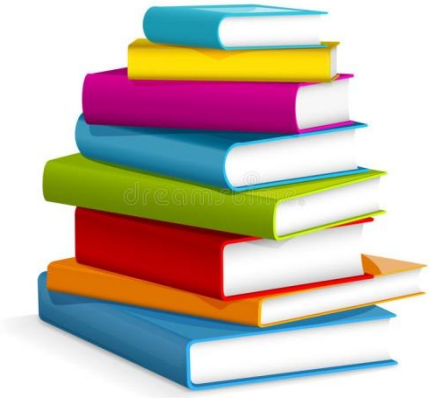


Pilha

- Pilhas (*stacks*) são uma estrutura de dados que usam listas encadeadas, onde **elementos são inseridos no fim e removidos do fim.**

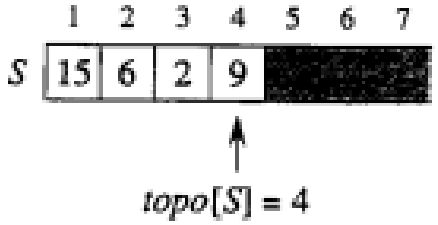
- O nome pilha é baseado em uma pilha de objetos (livros, pratos, etc). Quando montamos uma pilha o novo objeto é inserido no topo da mesma. E quando retiramos o último objeto inserido na pilha será o primeiro a ser retirado.

- Segue a regra LIFO –
  Last In, First Out
  (último a entrar, primeiro a sair)

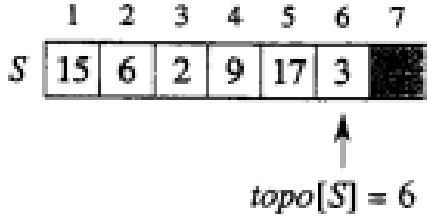


Pilha usando vetor estático

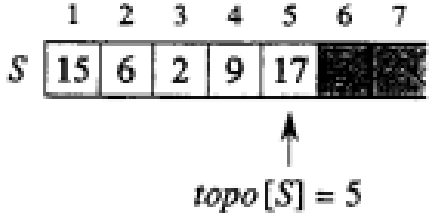
```
STACK-EMPTY(S)
1  if topo[S] = 0
2      then return TRUE
3      else return FALSE
```




```
PUSH(S, x)
1  topo[S] ← topo[S] + 1
2  S[topo[S]] ← x
```




```
POP(S)
1  if STACK-EMPTY(S)
2      then error "underflow"
3      else topo[S] ← topo[S] – 1
4          return S[topo[S] + 1]
```



Obtido em Cormen et. al

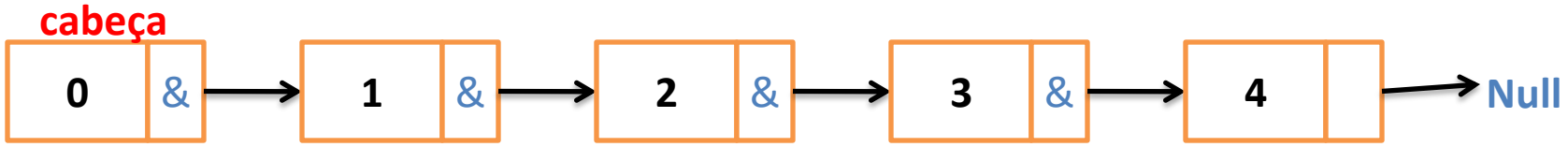
Pilha usando lista encadeada

- **Uma pilha é uma lista encadeada, logo uma sequencia de registros denominados célula.**
- **Cada célula contém um objeto de determinado tipo e o endereço da próxima célula.**

```
//estrutura da pilha
typedef struct cel {
   int conteudo;
   cel *prox;
}
```

```
//cria pilha com cabeça, o 1º elemento da pilha não contém informações
cel *pilha;
pilha= (cel) malloc (sizeof (cel));
pilha-> conteudo = 0;
pilha-> prox = NULL;
```

Inserção dos elementos 1, 2, 3, 4 e 5:



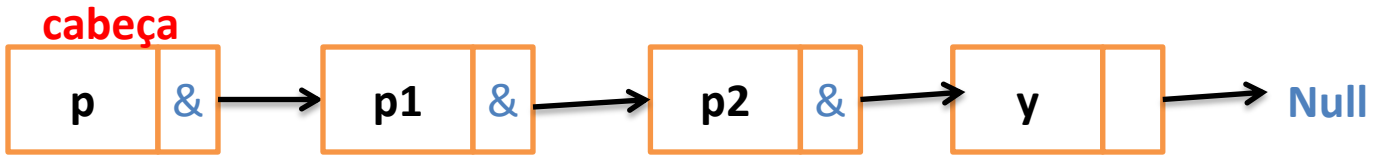
Pilha – Inserção (push/empilha)

1. Alocar espaço para o novo nó
2. O último elemento (que é o novo nó alocado) deve apontar para NULL
3. Se a pilha estiver vazia, fazer o ponteiro p->prox apontar para o novo nó
4. Se a pilha não estiver vazia, procurar último elemento por meio de um ponteiro auxiliar (*pAUX) que inicia no primeiro nó da pilha
5. Para saber qual é o último elemento da pilha basta checar se pAUX->prox == NULL
6. Se não apontar para NULL não é o último, então fazer ele apontar para o seguinte nó até chegar ao fim
7. Quando apontar para NULL chegou ao fim, fazer ele apontar para o novo nó pAUX ->prox = novo

Pilhas – Inserção (push/empilha)

```
//insere no fim da lista
void push(int y, cel *p) {
     cel *nova;
1     nova = (cel) malloc (sizeof (cel));
2    nova->prox = NULL;
     nova->conteudo = y;

    if(p ->prox == NULL)
 3        p->prox = nova;
    else
 4    cel  *pAUX = p->prox;
 5    while(pAUX ->prox != NULL)
 6        pAUX = pAUX ->prox ;
 7    pAUX ->prox  = nova;
}
```



Pilha – remoção (pop/desempilha)

1. Declarar dois ponteiros, um que aponta para o último elemento e outro para o penúltimo. O último elemento será removido e o penúltimo se tornará o novo último

2. Buscar o último nó, aquele onde ultimo->prox == NULL

3. Se não for o último, avança o ponteiro ultimo para o seguinte nó e atualiza o penultimo

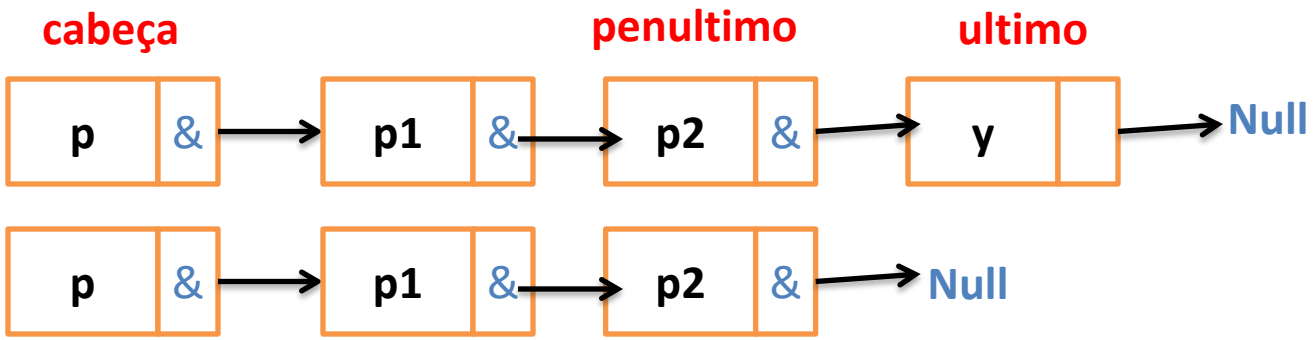
4. Quando chegar ao fim da pilha, fazer penultimo->prox = NULL, tornando o penultimo nó o último da pilha

5. O ultimo deixa de existir, liberando a memória.

Pilha- Remoção (pop/desempilha)

```
//remove do fim da lista
void pop (cel *p) {
1   cel *ultimo = p->prox;
1   cel *penultimo = p;

2   while(ultimo->prox != NULL) {
3       penultimo = ultimo;
3      ultimo = ultimo->prox;
    }
4    penultimo->prox = NULL;
5   free(ultimo);
}
```



Filas

- Filas (*queue*) são uma estrutura de dados que usam listas encadeadas, onde **elementos são inseridos no fim e removidos do início.**

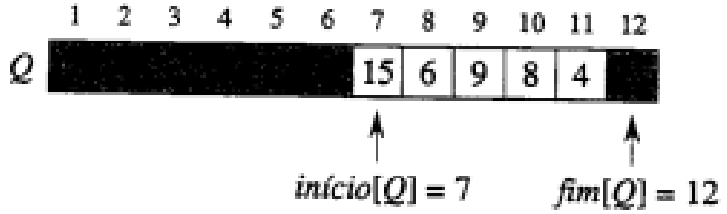
- O nome fila é baseado em uma fila de pessoas a serem atendidas. A primeira pessoa a chegar será a primeira a ser atendida, a última a chegar será a última a ser atendida.

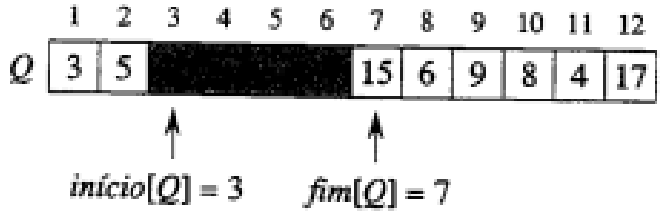
- Segue a regra FIFO –
  First In, First Out
  (primeiro a entrar, primeiro a sair)

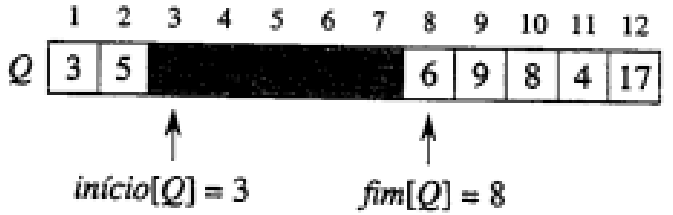
Fila usando vetor estático

```
ENQUEUE(Q, x)
1  Q[fim[Q]] ← x
2  if fim[Q] = comprimento[Q]
3     then fim[Q] ← 1
4     else fim[Q] ← fim[Q] + 1
```

```
DEQUEUE(Q)
1  x ← Q[início[Q]]
2  if início[Q] = comprimento[Q]
3     then início[Q] ← 1
4     else início[Q] ← início[Q] + 1
5  return x
```







Obtido em Cormen et. al

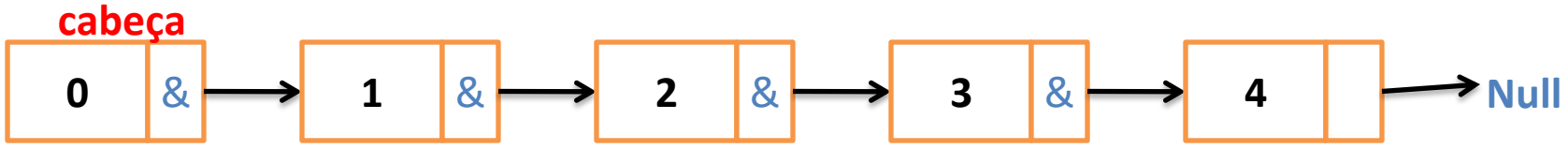
Fila usando lista encadeada

- **Uma fila é uma lista encadeada, logo uma sequencia de registros denominados célula.**
- **Cada célula contém um objeto de determinado tipo e o endereço da próxima célula.**

```
//estrutura da fila
typedef struct cel {
   int conteudo;
   cel *prox;
}
```

```
//cria fila com cabeça, o 1º elemento da fila não contém informações
cel *fila;
fila= (cel) malloc (sizeof (cel));
fila-> conteudo = 0;
fila-> prox = NULL;
```

Inserção dos elementos 1, 2, 3, 4 e 5:



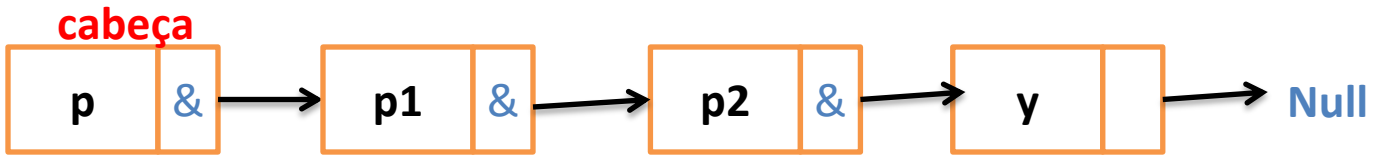
Fila – Inserção (enqueue)

1. Alocar espaço para o novo nó
2. O último elemento (que é o novo nó alocado) deve apontar para NULL
3. Se a pilha estiver vazia, fazer o ponteiro p->prox apontar para o novo nó
4. Se a pilha não estiver vazia, procurar último elemento por meio de um ponteiro auxiliar (*pAUX) que inicia no primeiro nó da pilha
5. Para saber qual é o último elemento da pilha basta checar se pAUX->prox == NULL
6. Se não apontar para NULL não é o último, então fazer ele apontar para o seguinte nó até chegar ao fim
7. Quando apontar para NULL chegou ao fim, fazer ele apontar para o novo nó pAUX ->prox = novo

Fila – Inserção (enqueue)

```
//insere no fim da lista
void enqueue (int y, cel *p) {
     cel *nova;
1     nova = (cel) malloc (sizeof (cel));
2    nova->prox = NULL;
     nova->conteudo = y;

    if(p ->prox == NULL)
 3        p->prox = nova;
    else
 4    cel  *pAUX = p->prox;
 5    while(pAUX ->prox != NULL)
 6        pAUX = pAUX ->prox ;
 7    pAUX ->prox  = nova;
}
```



Fila – remoção (dequeue)

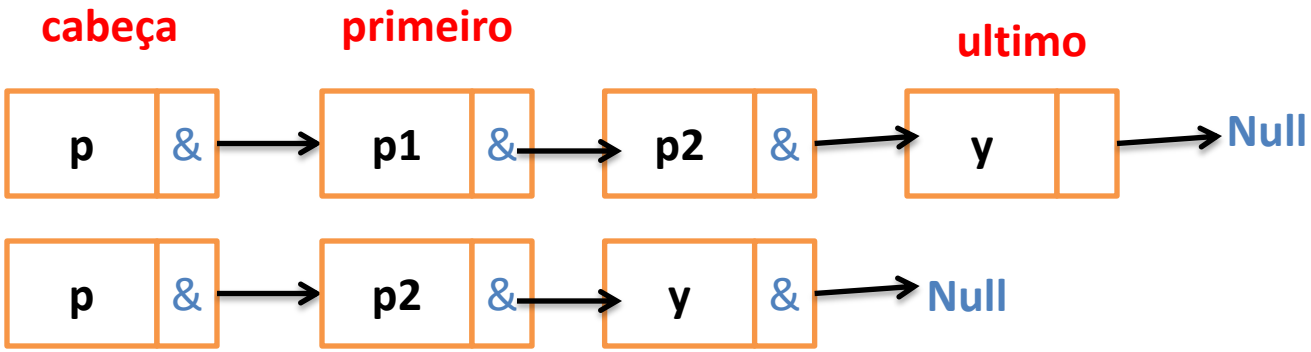
1. Declarar um ponteiro temporário para apontar para o primeiro elemento da fila (que será removido)

2. Fazer a ligação entre o ponteiro cabeça da fila e o segundo elemento: p->prox = primeiro->prox

3. O primeiro deixa de existir, pois se ningúem aponta para ele, não faz mais parte da estrutura de dados, pode liberar a memória.

Fila – remoção (dequeue)

```
//remove no inicio da lista
void dequeue (cel *p) {
1     cel *primeiro;
1    primeiro= p->prox;
2    p->prox = primeiro->prox;
3     free(primeiro);
}
```



Discussão

Vantagens:

- Filas são utilizadas em diversas aplicações reais, como Sistema Operacional (SO) para organizar os processos, o que deve ser processado primeiro, etc.

- Recursão usa pilha para inserir resultados parciais, e depois desempilha para retornar o resultado final.

- Pilhas podem ser usadas para controlar expressões aritméticas:
  (a + b) / c.

Desvantagem:

- Utilização de memória extra para armazenar os apontadores.

Lista duplamente ligada

- A conexão entre os elementos é feita por meio de **dois ponteiros** (um que aponta para o elemento anterior, e o outro, para o seguinte).

- Para acessar um elemento a lista pode ser percorrida por ambos os lados.
- Do primeiro ao último elemento e do último ao primeiro.

Código da lista duplamente ligada

- **Uma estrutura célula contém os dados necessários para o TAD: conteúdo e dois ponteiros, para o elemento anterior e próximo.**
- **Pode ser usada outra estrutura para armazenar variáveis adicionais, como o início e fim da lista, e tamanho.**

```
//estrutura da célula
typedef struct cel {
   int conteudo;
   cel *ant;
   cel *prox;
}
```

```
//estrutura adicional para conter informações
da lista, como primeiro e último elemento e tam.
typedef struct Lista{
   cel *inicio;
   cel *fim;
   int tamanho;
}
```

```
//Inicialização da lista
void inicializa (Lista *lista) {
   lista->inicio = NULL;
   lista->fim = NULL;
   tamanho = 0;
}
```

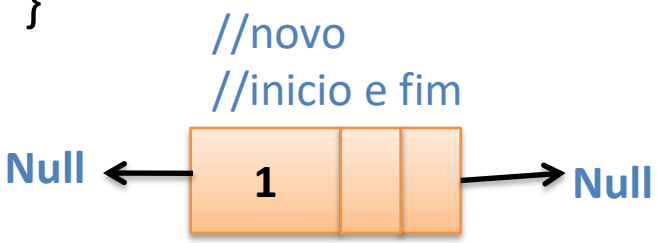
Inserção de um elemento na lista vazia

1. Alocar memória para o novo elemento;
2. Preencher campos de dados do novo elemento;
3. O ponteiro anterior do novo aponta para NULL;
4. O ponteiro seguinte ao novo aponta para NULL;
5. Os ponteiros de início e fim indicam o novo elemento;
6. O tamanho é atualizado.

Inserção de um elemento na lista vazia

```
void inserir_Lista_Vazia (Lista *lista, int dado) {
   cel *novo;
1    novo = (cel) malloc (sizeof (cel));
2    novo->conteudo = dado;
3    novo->ant = lista->inicio;
4    novo-> prox = lista->fim;

5    lista->inicio = novo;
5    lista->fim = novo;
6    lista->tamanho++;
}
```



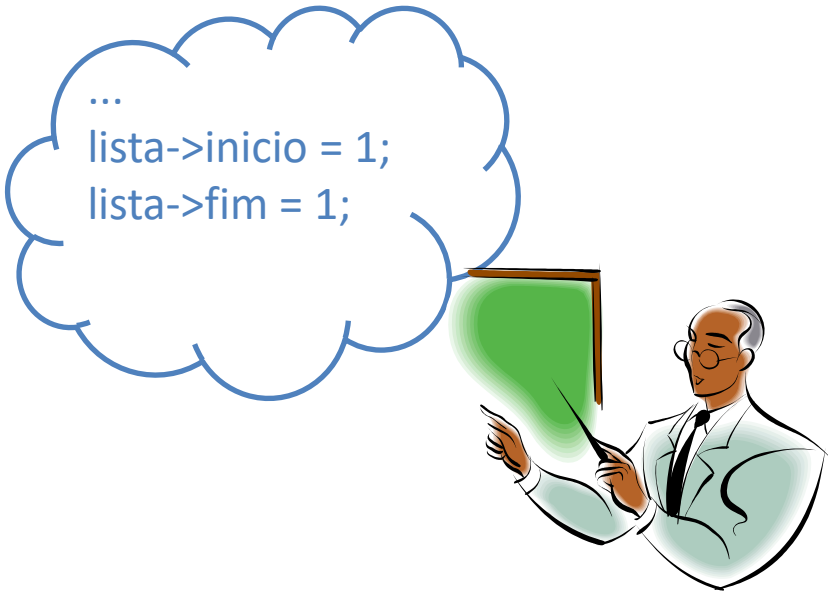
Inserção de um elemento no início da lista

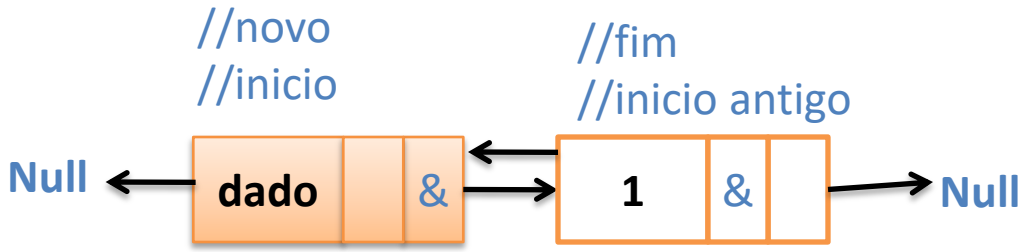
1. Alocar memória para o novo elemento;
2. Preencher campos de dados do novo elemento;
3. O ponteiro anterior do novo aponta para NULL;
4. O ponteiro seguinte aponta para 1º elemento;
5. O ponteiro anterior ao 1º elemento indica o novo;
6. O ponteiro de início indica o 1º elemento;
7. O ponteiro fim não muda;
8. O tamanho é incrementado.

Inserção de um elemento no início da lista

```
void inserir_Inicio_Lista (Lista *lista, int dado) {
   cel *novo;
1    novo = (cel) malloc (sizeof (cel));
2    novo->conteudo = dado;
3    novo->ant = NULL;
4    novo-> prox = lista->inicio;

5    lista->inicio->ant = novo;
6    lista->inicio= novo;
8    lista->tamanho++;
}
```





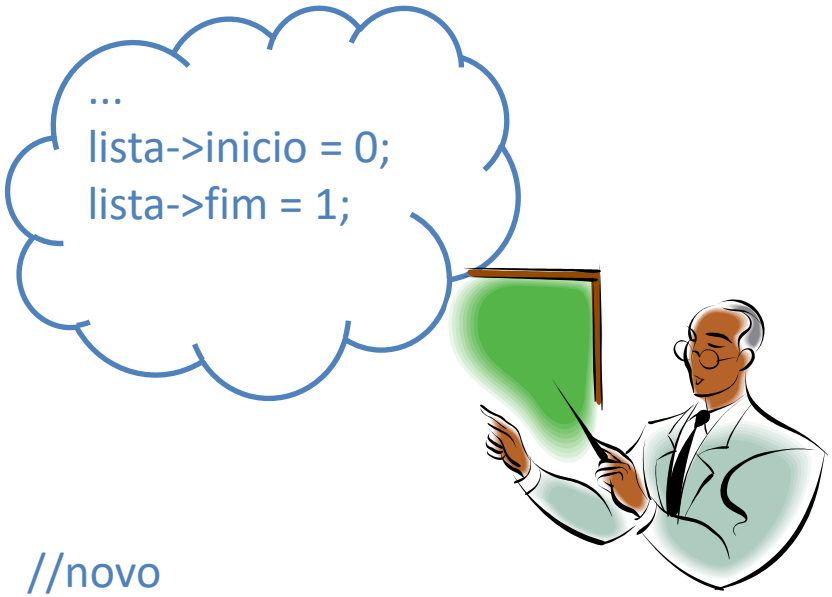
Inserção de um elemento no fim da lista

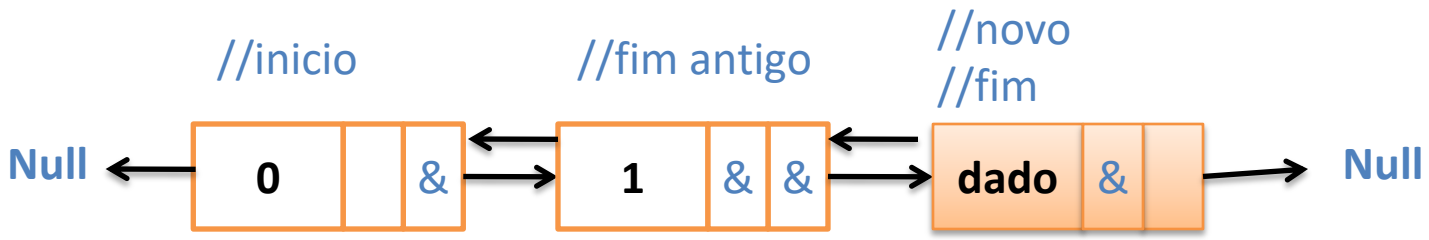
1. Alocar memória para o novo elemento;
2. Preencher campos de dados do novo elemento;
3. O ponteiro anterior ao novo aponta para último elemento;
4. O ponteiro seguinte do novo aponta para NULL;
5. O ponteiro seguinte em relação ao último aponta para o novo;
6. O ponteiro de início não muda;
7. O ponteiro fim aponta para o novo;
8. O tamanho é incrementado.

Inserção de um elemento no fim da lista

```
void inserir_Fim_Lista (Lista *lista, int dado) {
   cel *novo;
1    novo = (cel) malloc (sizeof (cel));
2    novo-> conteudo = dado;
3    novo->ant = lista->fim;
4    novo-> prox = NULL;

5    lista->fim->prox= novo;
6    lista->fim= novo;
8    lista->tamanho++;
}
```





Inserção depois de uma posição qualquer da lista

1. Alocar memória para o novo elemento;
2. Preencher campos de dados do novo elemento;
3. Selecionar a posição (AUX) da lista para inserir depois dela;
4. O ponteiro anterior ao novo aponta para AUX;
5. O ponteiro seguinte do novo aponta para o seguinte de AUX;
6. O ponteiro anterior ao elemento depois de AUX aponta para o novo;
7. O ponteiro seguinte em relação a AUX aponta para o novo;
8. O ponteiro início não muda;
9. O ponteiro fim não muda;
10. O tamanho é incrementado.

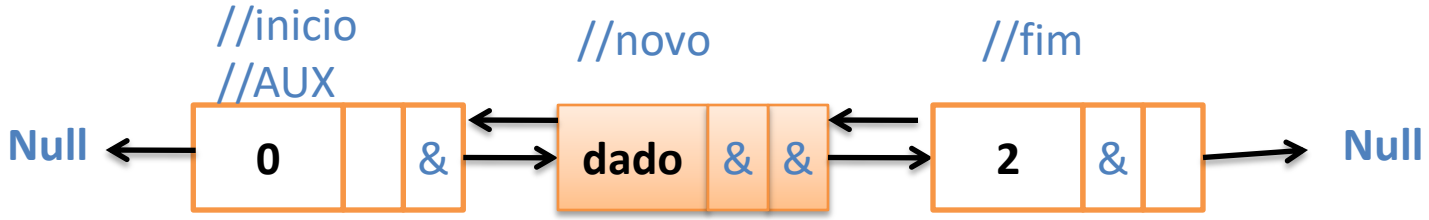
Inserção depois de uma posição qualquer da lista

```
void inserir_Posicao_Lista (Lista *lista, int dado, int pos) {
   cel *novo, *AUX;
1    novo = (cel) malloc (sizeof (cel));
2    novo->conteudo = dado;
3    AUX = lista->inicio;
3         for(int i = 1; i < pos; i++)
3             AUX = AUX->prox;
4    novo->ant = AUX;
5    novo->prox = AUX->prox;

6    AUX->prox->ant = novo;
7    AUX->prox= novo;
10    lista->tamanho++;
}
```

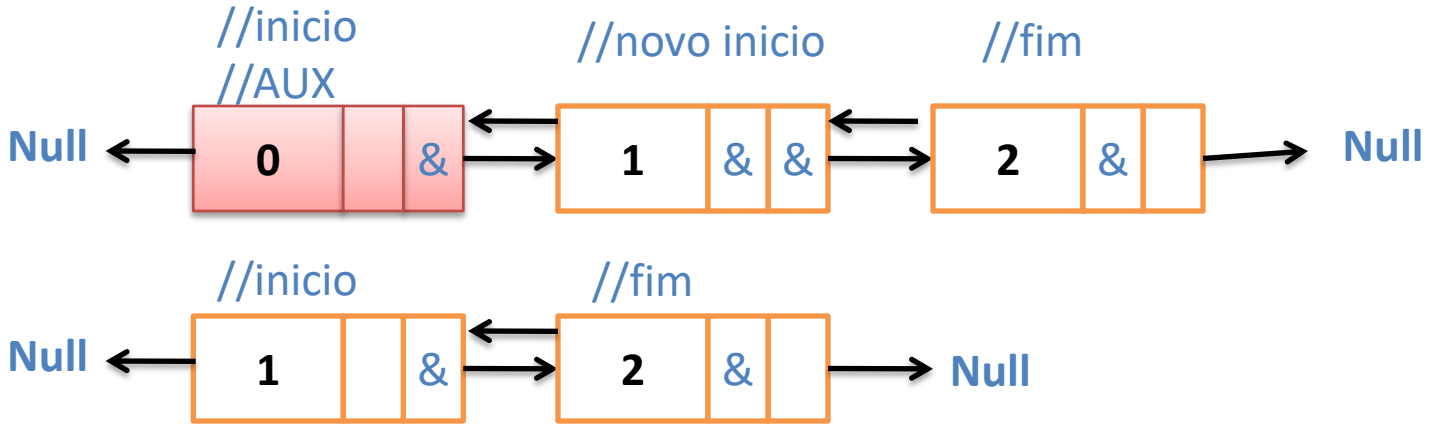
1

Nas listas duplamente ligadas
Pode-se remover o elemento anterior
E posterior da posição informada



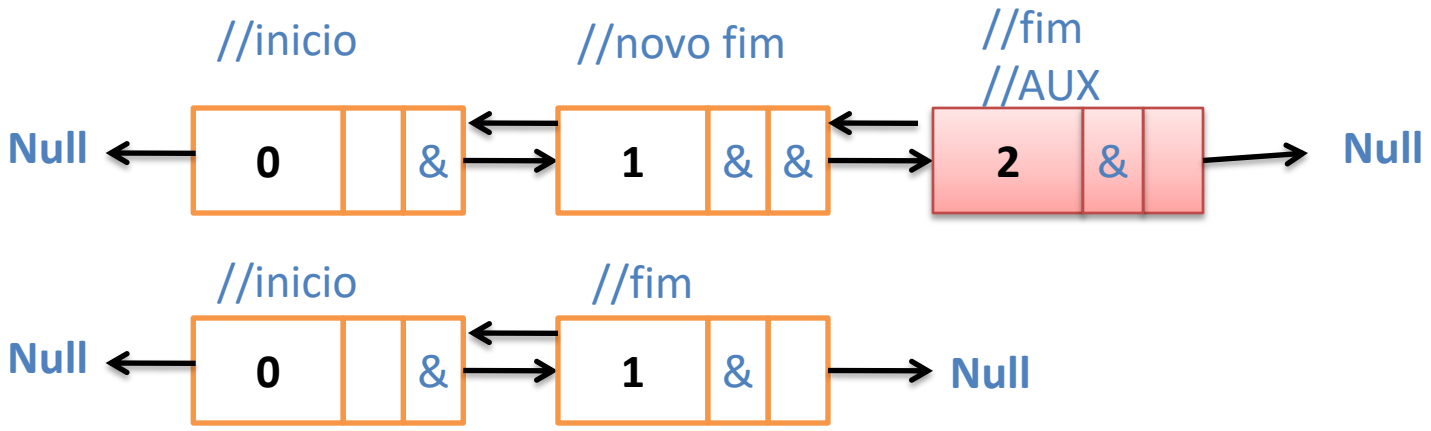
Remoção do 1º elemento da lista

```
void remove_Primeiro_Lista (Lista* lista, int pos) {
   if(lista->tamanho == 0) printf("Lista vazia");
   else if(pos == 1)
   cel *AUX = lista->inicio;
   lista->inicio = lista->inicio->prox;
   if(lista->inicio == NULL)
       lista->fim = NULL;
   else lista->inicio->ant = NULL;
   lista->tamanho--;
}
```



Remoção do último elemento da lista

```
void remove_Ultimo_Lista (Lista* lista, int pos) {
   if(lista->tamanho == 0) printf("Lista vazia");
   else if(pos == lista->tamanho)
   cel *AUX = lista->fim;
   lista->fim->ant->prox= NULL;
   lista->fim = lista->fim->ant;
   lista->tamanho--;
}
```



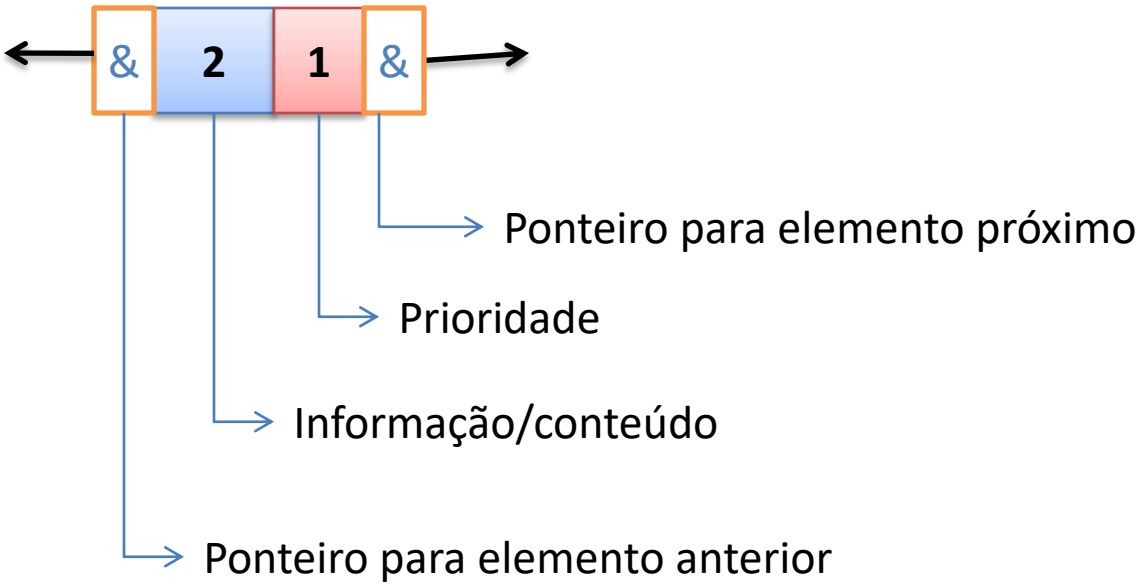
Fila com prioridade

- Permite que elementos adicionados na fila possuam uma prioridade.
- O elemento de maior prioridade deve ser o primeiro a ser removido da fila.

- Ex: Filas de atendimento, algumas pessoas tem atendimento preferencial (idosos, grávidas, etc). Filas de processos para serem executados num computador, alguns devem ser executados primeiro.

Estrutura

```
typedef struct cel {
    int conteudo;
    cel *ant;
    cel *prox;
    int prioridade;
}
```




```
typedef struct Fila{
    cel *inicio;
    cel *fim;
    int tamanho;
}
```

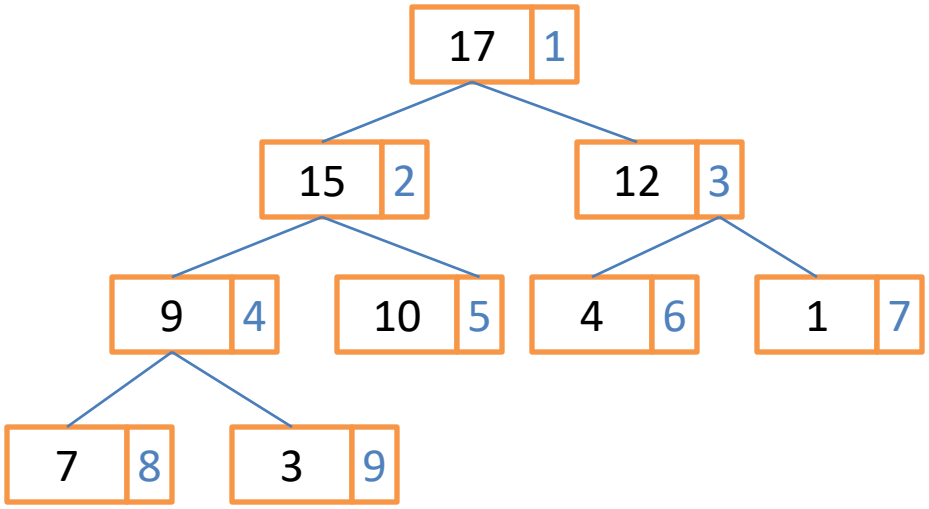
Inserção e remoção dos elementos

- **Inserção**:
  - não-ordenada: inserção sempre no final da fila, custo O(1);
  - ordenada: inserir na posição correta de acordo com a prioridade, custo O(n);

- **Remoção**:
  - não-ordenada: percorrer todos os elementos até encontrar o maior, custo O(n);
  - ordenada: o maior está sempre na primeira/última posição, custo O(1);

Heap

- É **uma estrutura de árvore binária** em que cada nó tem uma prioridade maior ou igual a do seus filhos.
- O nó **i** possui dois filhos nas posições: **2i e 2i+1**.
- **Custo para inserir e remover é sempre log(n).**



1	2	3	4	5	6	7	8	9
17	15	12	9	10	4	1	7	3

i	2i	2i+1
1	2	3
2	4	5
3	6	7
4	8	9

Exercícios

- Resolver exercícios no URI Judge:
- 1242 Ácido Ribonucleico Alienígena
- 1083 LEXSIM - Avaliador Lexico e Sintático

O sucesso é a soma de pequenos esforços repetidos dia após dia.

Robert Collier