FILA DE PRIORIDADE & DEQUE

Prof. André Backes | @progdescomplicada

Fila de Prioridades | Definição

- Tipo especial de fila que generaliza a ideia de **ordenação**
- Os elementos inseridos possuem um dado extra associados a eles: a sua **prioridade**
- Esse valor determina a
  - posição de um elemento na fila
  - o primeiro a ser removido da fila, quando necessário

Fila de Prioridades | Definição

- Elementos com prioridades iguais?
  - Dois elementos podem ter a mesma prioridade na fila
  - O elemento inserido primeiro fica a frente de quem foi inserido posteriormente se as prioridades forem iguais

Fila de Prioridades | Aplicações

- Basicamente, qualquer problema em que seja preciso estabelecer uma prioridade de acesso aos elementos
- Exemplos
  - processador, processos com maior prioridade são executados primeiros
  - fila de pacientes esperando transplante de fígado
  - busca em grafos (algoritmo de Dijkstra)
  - compressão de dados (código de Huffman)
  - sistemas operacionais (manipulação de interrupções)
  - fila de pouso de aviões (prioridade por combustível disponível)

Fila de Prioridades | Implementação

- Existem diversas implementações disponíveis para a fila de prioridades
  - lista dinâmica encadeada
  - array desordenado
  - array ordenado
  - heap binária

Fila de Prioridades | Implementação

- A escolha depende da sua aplicação!
  - Algumas implementações são eficientes na operação de inserção, outras na de remoção
  - Há também implementações que são eficientes nas duas operações

Implementação	Inserção	Remoção
lista dinâmica encadeada	*O(N)*	*O(1)*
array desordenado	*O(1)*	*O(N)*
array ordenado	*O(N)*	*O(1)*
heap binária	*O(log N)*	*O(log N)*

Fila de Prioridades | TAD

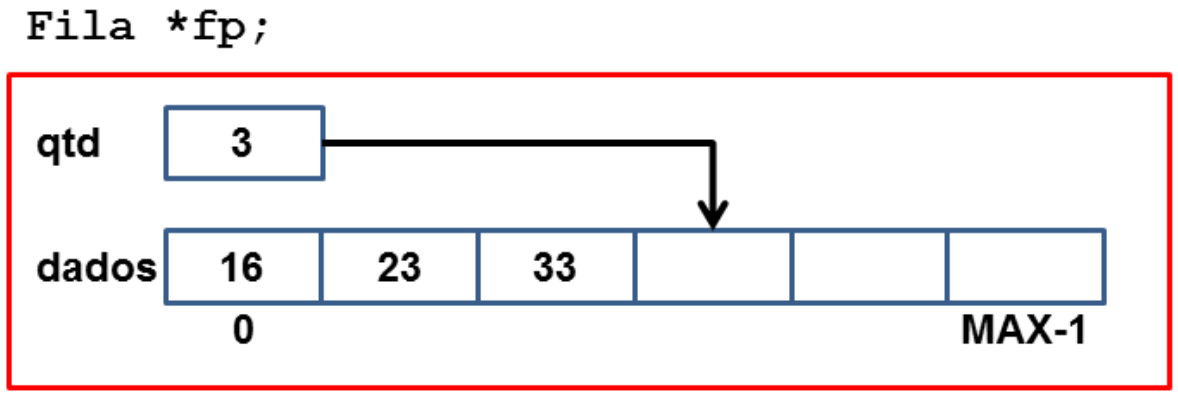
- Fila de Prioridades Estática
  - É a mesma estrutura da Lista Sequencial Estática
    - Utiliza um array para armazenar os elementos
    - Neste caso, o **nome** e a **prioridade** de pacientes em um pronto-socorro
  - Vantagem: fácil de criar e destruir
    Desvantagem: necessidade de definir previamente o tamanho da fila

```c
#define MAX 100
struct paciente{
    char nome[30];
    int prio;
};
//Definição do tipo fila de prioridade
struct fila_prioridade{
    int qtd;
    struct paciente dados[MAX];
};

typedef struct fila_prioridade FilaPrio;
```

Fila de Prioridades | TAD

- O fato de utilizarmos um array permite que a mesma estrutura seja usada para duas implementações distintas
  - array ordenado
  - heap binária

- Basta modificar as funções de
  - inserção
  - remoção
  - consulta



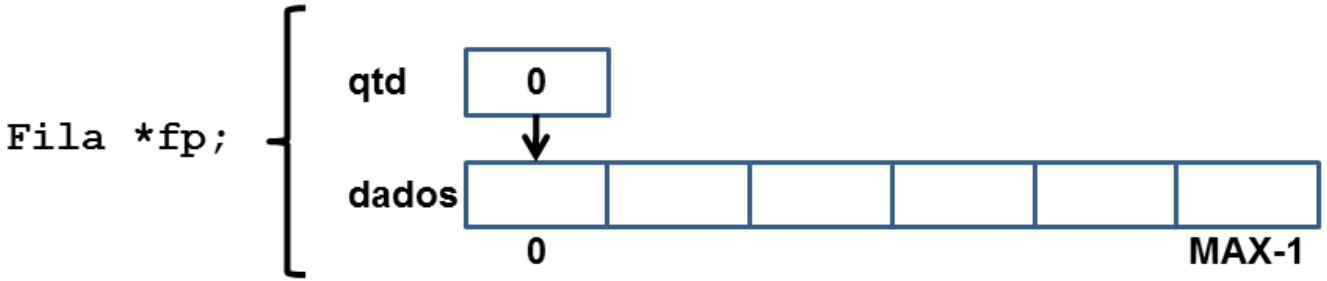
Fila de Prioridades | Criação e liberação

- Criação
  - Aloca uma área de memória para a fila
  - Corresponde a memória necessária para armazenar a estrutura da fila
- Liberação
  - Basta liberar a memória alocada para a estrutura fila

```
FilaPrio* cria_FilaPrio(){
    FilaPrio *fp;
    fp = (FilaPrio*) malloc(sizeof(struct fila_prioridade));
    if(fp != NULL)
        fp->qtd = 0;
    return fp;
}

void libera_FilaPrio(FilaPrio* fp){
    free(fp);
}
```

Fila de Prioridades | Criação e liberação

- Neste caso, a criação produz uma fila de prioridades que está vazia
  - qtd igual a zero



FILA DE PRIORIDADE USANDO ARRAY ORDENADO

Fila de Prioridades | Implementação

- Os elementos na fila de prioridade são ordenados de forma crescente dentro do array
  - A maior prioridade estará sempre no final do array (início da fila)
  - A menor prioridade estará na primeira posição do array (final da fila)
- Custo
  - Inserção: *O(N)*, precisamos procurar o ponto de inserção no array
  - Remoção: *O(1)*, tempo constante

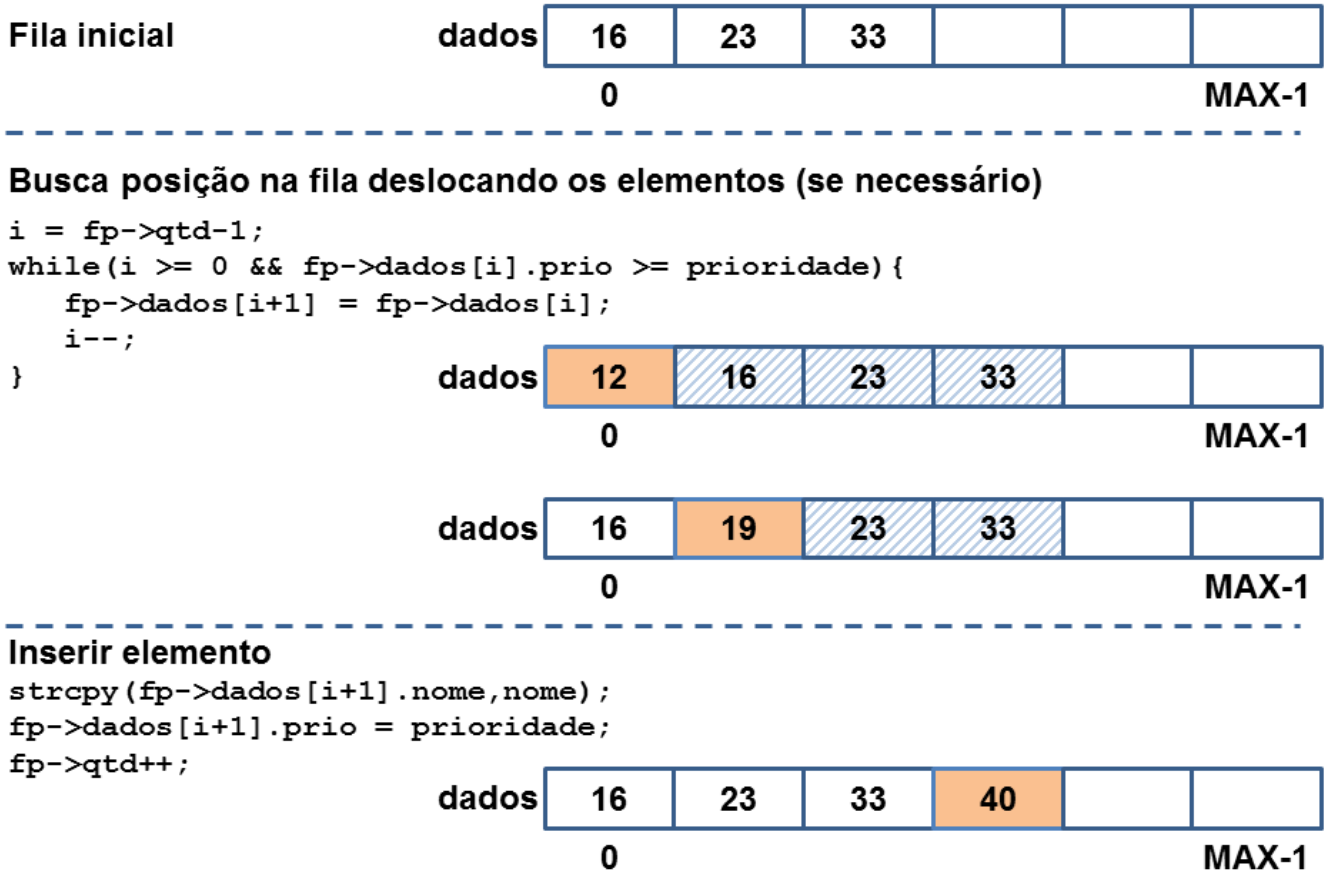
Fila de Prioridades | Inserção

- Tarefa simples, mas trabalhosa
  - Envolve procurar a posição de inserção de acordo com a prioridade e deslocar os demais elementos do array
- Precisamos verificar
  - se a fila existe
  - se a fila está cheia
- E só depois
  - procurar a posição de inserção
  - copiar os dados
  - incrementar a quantidade

```
int insere_FilaPrio(FilaPrio* fp, char *nome, int prio){
    if(fp == NULL)
        return 0;
    if(fp->qtd == MAX)//fila cheia
        return 0;

    int i = fp->qtd-1;
    while(i >= 0 && fp->dados[i].prio >= prio){
        fp->dados[i+1] = fp->dados[i];
        i--;
    }

    strcpy(fp->dados[i+1].nome,nome);
    fp->dados[i+1].prio = prio;
    fp->qtd++;
    return 1;
}
```

Fila de Prioridades | Inserção

- Elemento com prioridade maior do que a inserção (em laranja) são deslocados

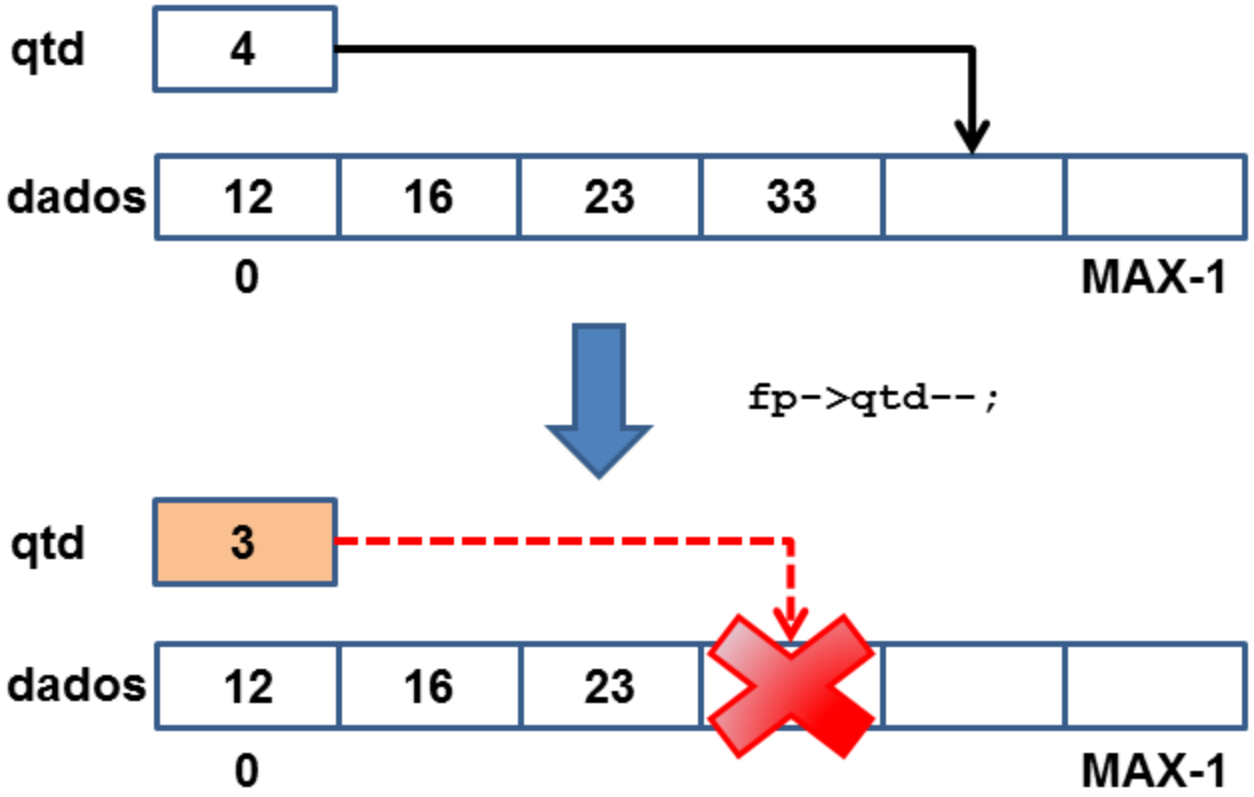


Fila de Prioridades | Remoção

- A remoção é sempre feita no início da fila: final do array
  - Equivale a remoção de uma Lista Sequencial Estática
- Precisamos verificar
  - se a fila existe
  - se a fila não está vazia
- E só depois
  - diminuir a quantidade

```
int remove_FilaPrio(FilaPrio* fp){
    if(fp == NULL)
        return 0;
    if(fp->qtd == 0)
        return 0;
    fp->qtd--;
    return 1;
}
```

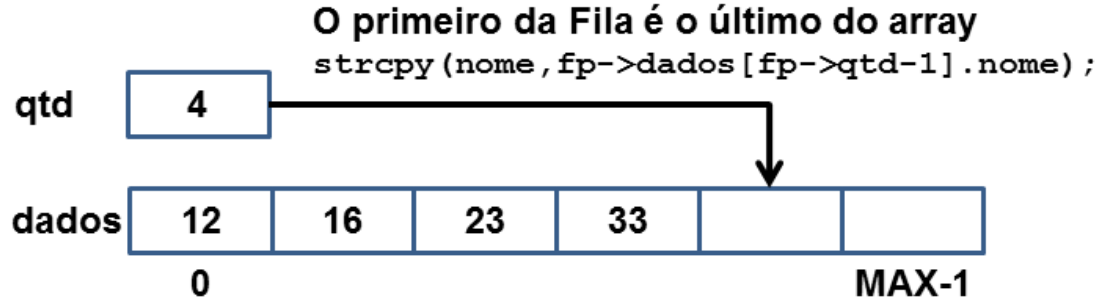
Fila de Prioridades | Remoção



Fila de Prioridades | Acesso

- Apesar de idêntica a Lista Sequencial Estática, uma fila de prioridades continua sendo uma fila
  - só podemos acessar o início da fila
  - última posição ocupada do array
- Precisamos verificar
  - se a fila existe e não está vazia
- E só depois
  - acessar os dados do início

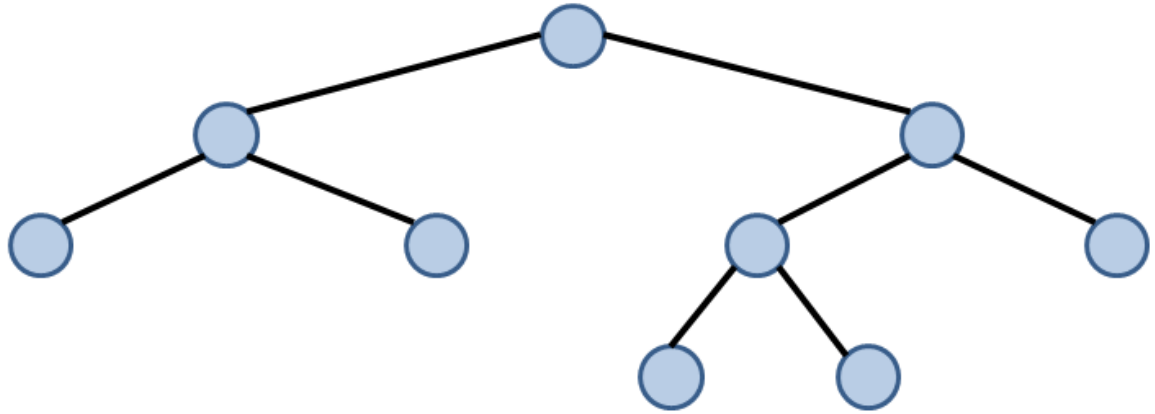
```
int consulta_FilaPrio(FilaPrio* fp, char* nome){
    if(fp == NULL || fp->qtd == 0)
        return 0;
    strcpy(nome,fp->dados[fp->qtd-1].nome);
    return 1;
}
```

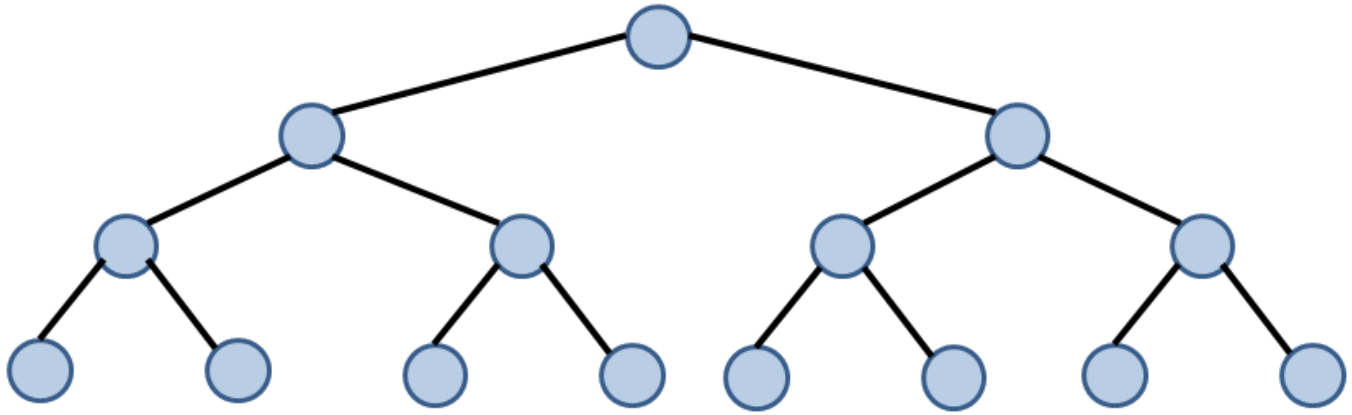


FILA DE PRIORIDADE USANDO HEAP BINÁRIA

Fila de Prioridades | Implementação

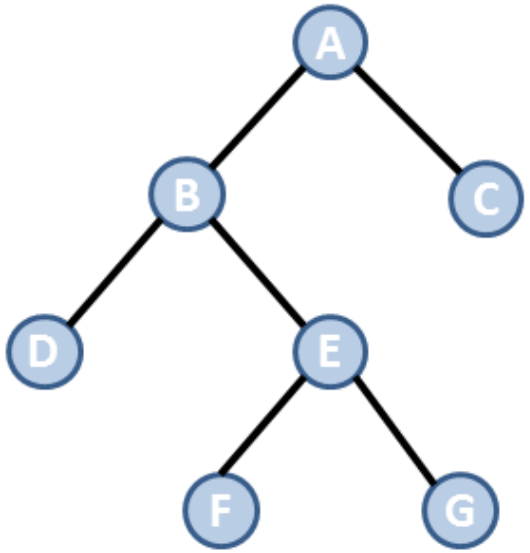
- Uma heap permite simular uma árvore binária completa ou quase completa
  - A exceção é o seu último nível





Fila de Prioridades | Implementação

- Na heap, cada posição do array é pai de duas outras posições
  - Usa 2 funções para retornar a posição dos filhos à esquerda e à direita de um pai



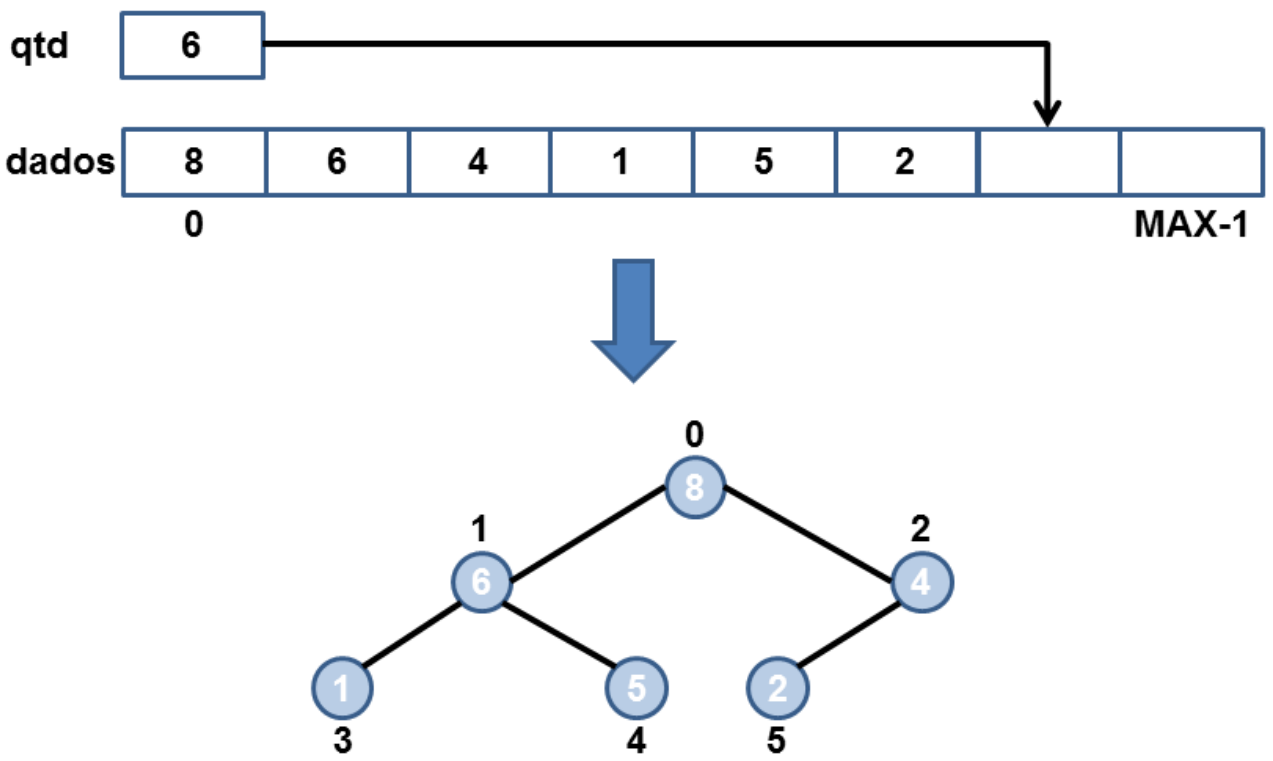
FILHO_ESQ(PAI) = 2 * PAI + 1
FILHO_DIR(PAI) = 2 * PAI + 2

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E					F	G	

FILHOS

Fila de Prioridades | Implementação

- Na heap, um nó pai tem sempre uma prioridade maior ou igual à de seus filhos
  - O elemento de maior prioridade estará sempre no início do array
    - início da fila
  - O de menor prioridade estará no último nível da heap
    - Não necessariamente na última posição do array



Fila de Prioridades | Inserção

- Tarefa simples, mas trabalhosa
- Precisamos verificar
  - se a fila existe
  - se a fila está cheia
- E só depois
  - copiar os dados para o final do array
  - incrementar a quantidade
  - **reorganizar a heap**

```
int insere_FilaPrio(FilaPrio* fp, char *nome, int prio){
    if(fp == NULL)
        return 0;
    if(fp->qtd == MAX)//fila cheia
        return 0;
    strcpy(fp->dados[fp->qtd].nome,nome);
    fp->dados[fp->qtd].prio = prio;
    promoverElemento(fp,fp->qtd);
    fp->qtd++;
    return 1;
}
```

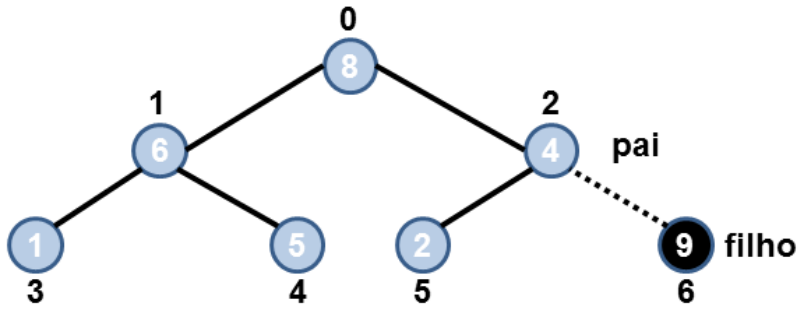
Fila de Prioridades | Inserção

- Precisamos verificar e corrigir possíveis violações na heap
  - a prioridade do pai é sempre maior ou igual a de seus filhos
- Dada a posição do último inserido
  - Verifica se o pai tem prioridade maior
  - Se o pai é menor, troca de lugar com o filho e sobe na heap

```
void promoverElemento(FilaPrio* fp, int filho){
    int pai;
    struct paciente temp;
    pai = (filho - 1) / 2;
    while((filho > 0) &&
              (fp->dados[pai].prio <= fp->dados[filho].prio)){

        temp = fp->dados[filho];
        fp->dados[filho] = fp->dados[pai];
        fp->dados[pai] = temp;

        filho = pai;
        pai = (pai - 1) / 2;
    }
}
```

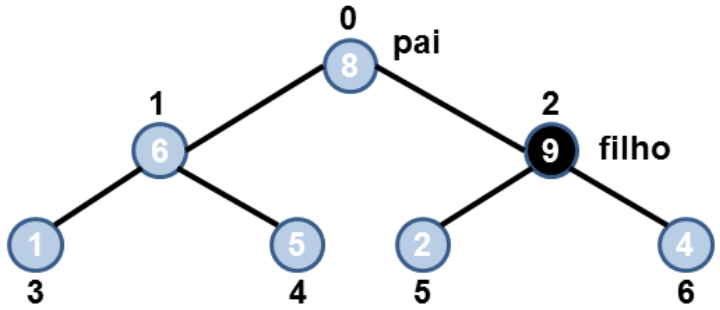
Fila de Prioridades | Inserção

Insere elemento com prioridade 9



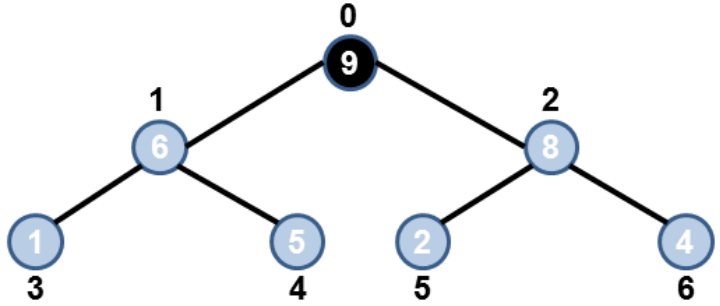
Prioridade do pai
é menor do que a
Prioridade do filho:

Trocar os dois de lugar



Prioridade do pai
é menor do que a
Prioridade do filho:

Trocar os dois de lugar



Elemento é o primeiro
da Heap

Finalizar processo

Fila de Prioridades com Heap Binária | Remoção

- Tarefa simples, mas trabalhosa
- Precisamos verificar
  - se a fila existe
  - se a fila não está vazia
- E só depois
  - diminuir a quantidade
  - mover a última posição do array para o início
    - topo da heap não é mais a maior prioridade!
  - **reorganizar a heap**

```
int remove_FilaPrio(FilaPrio* fp){
    if(fp == NULL) return 0;
    if(fp->qtd == 0) return 0;

    fp->qtd--;
    fp->dados[0] = fp->dados[fp->qtd];
    rebaixarElemento(fp,0);
    return 1;
}
```

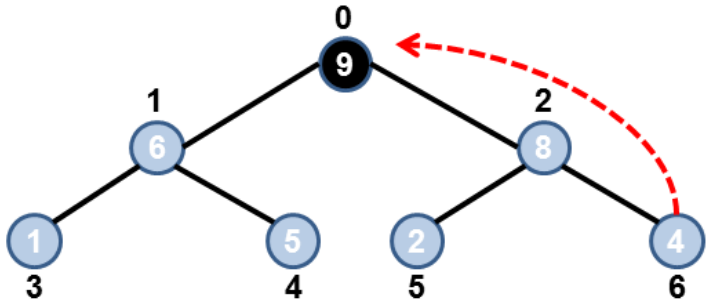
Fila de Prioridades com Heap Binária | Remoção

- Começando pelo topo da heap
  - Verifica se o pai tem dois filhos
    - Seleciona o filho de maior prioridade
  - Verifica se o pai tem prioridade maior que o filho selecionado
  - Se o pai é menor, troca de lugar com o filho e desce na heap

```
void rebaixarElemento(FilaPrio* fp, int pai){
  struct paciente temp;
  int filho = 2 * pai + 1;
  while(filho < fp->qtd){
    if(filho < fp->qtd-1)
      if(fp->dados[filho].prio < fp->dados[filho+1].prio)
          filho++;

    if(fp->dados[pai].prio >= fp->dados[filho].prio)
        break;

    temp = fp->dados[pai];
    fp->dados[pai] = fp->dados[filho];
    fp->dados[filho] = temp;

    pai = filho;
    filho = 2 * pai + 1;
  }
}
```

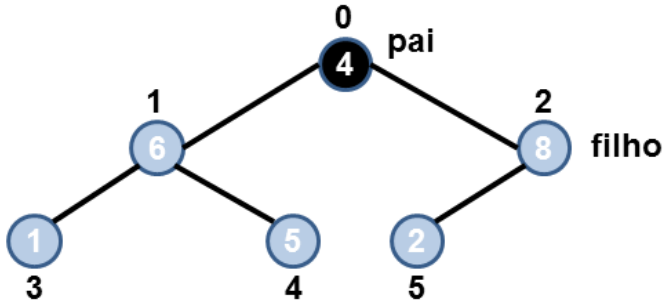
Fila de Prioridades com Heap Binária | Remoção

Remove elemento com maior prioridade



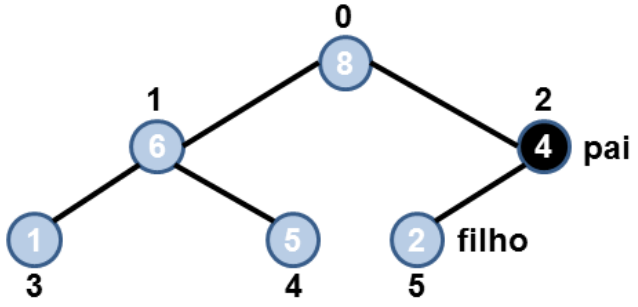
Copia o último elemento
da Heap para o topo

Iniciar ajuste da Heap



Prioridade do pai
é menor do que a
Prioridade do filho:

Trocar os dois de lugar



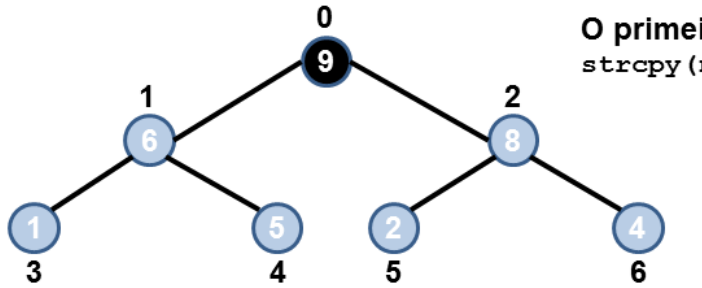
Prioridade do pai
é maior do que a
Prioridade do filho:

Finalizar processo

Fila de Prioridades com Heap Binária | Acesso

- Apesar de idêntica a Lista Sequencial Estática, uma fila de prioridades continua sendo uma fila
  - só podemos acessar o início da fila
  - última posição ocupada do array
- Precisamos verificar
  - se a fila existe e não está vazia
- E só depois
  - acessar os dados do topo da heap

```
int consulta_FilaPrio(FilaPrio* fp, char* nome){
    if(fp == NULL || fp->qtd == 0)
        return 0;
    strcpy(nome,fp->dados[0].nome);
    return 1;
}
```



O primeiro da Fila é o primeiro da Heap

```
strcpy(nome,fp->dados[0].nome);
```

DEQUE

Deque | Definição

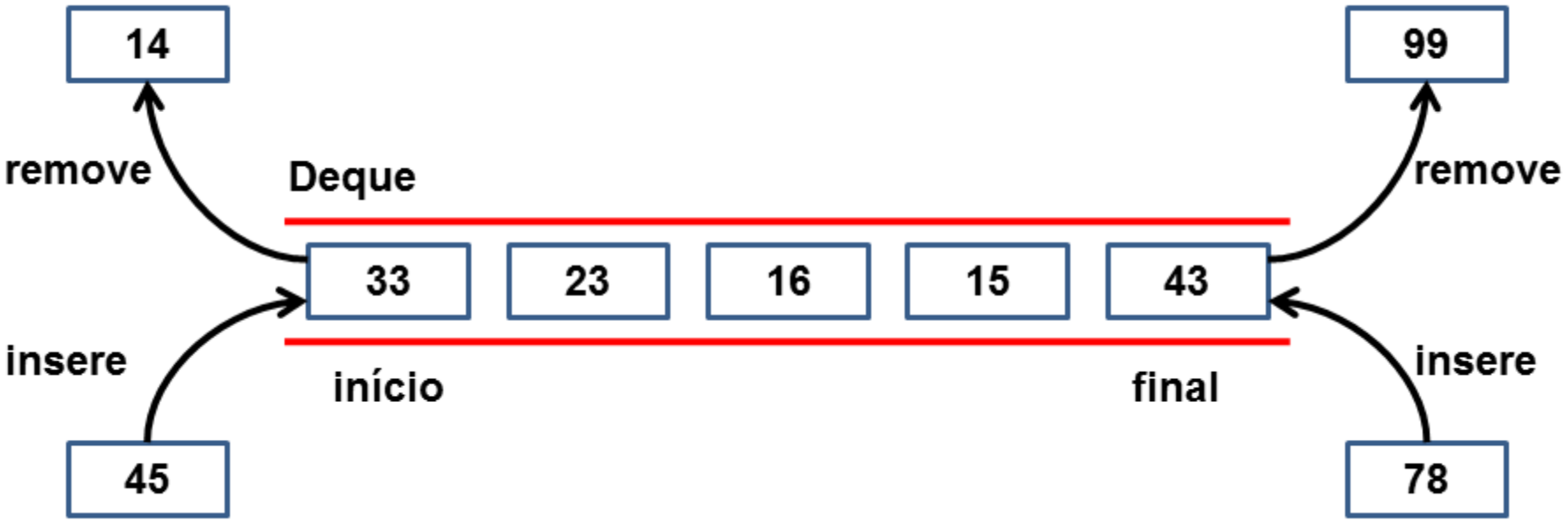
- ***D****ouble* ***E****nded* ***QUE****ue*, isto é, fila com duas saídas
  - Tipo especial de fila que permite a inserção e a remoção em ambas as extremidades da estrutura

- É uma estrutura de dados linear utilizada para armazenar e controlar o fluxo de dados em um computador
  - Sequência de elementos do mesmo tipo
  - Pode ter elementos repetidos, dependendo da aplicação

Deque | Definição

- Generaliza a ideia de fila e pilha, sendo utilizado como substituto de ambos
  - deque como pilha: inserção e remoção na mesma extremidade
  - deque como fila: inserção e remoção em extremidades opostas

Deque | Definição



Deque | Definição

- Existem duas implementações principais para um deque
- Deque estático
  - Os elementos são armazenados de forma consecutiva na memória (array)
  - É necessário definir o número máximo de elementos que a fila irá possuir

- A implementação pode ser com
  - array não circular
    - As operações de inserção e remoção no início envolvem o deslocamento de conteúdo
  - array circular
    - Simula uma lista circular, evitando deslocamentos desnecessários de elementos

Deque | Definição

- Existem duas implementações principais para um deque
- Deque dinâmico
  - O espaço de memória é alocado em tempo de execução
  - O deque cresce e diminui com o tempo
  - Cada elemento do deque armazena o endereço de memória do próximo
- A implementação pode ser com
  - encadeamento simples
    - cada elemento aponta apenas para o próximo
  - encadeamento duplo
    - cada elemento aponta para o elemento anterior e seguinte a ele

DEQUE ESTÁTICO COM ARRAY CIRCULAR

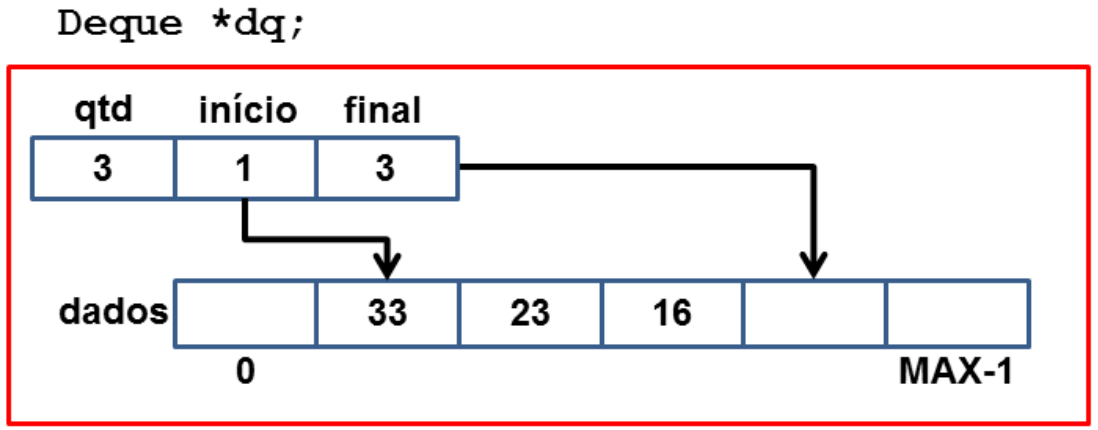
Deque Estático | TAD

- Deque Estático com array circular
  - Similar a **Fila Estática**
  - Vantagem: fácil de criar e destruir, evita deslocamentos desnecessários de elementos
  - Desvantagem: necessidade de definir previamente o tamanho do deque

```c
#define MAX 10

//Definição do tipo Deque
struct deque{
    int inicio, final, qtd;
    struct aluno dados[MAX];
};

typedef struct deque Deque;
```

Deque Estático | TAD

- O deque é mantido usando três campos
  - início
  - final
  - quantidade de elementos

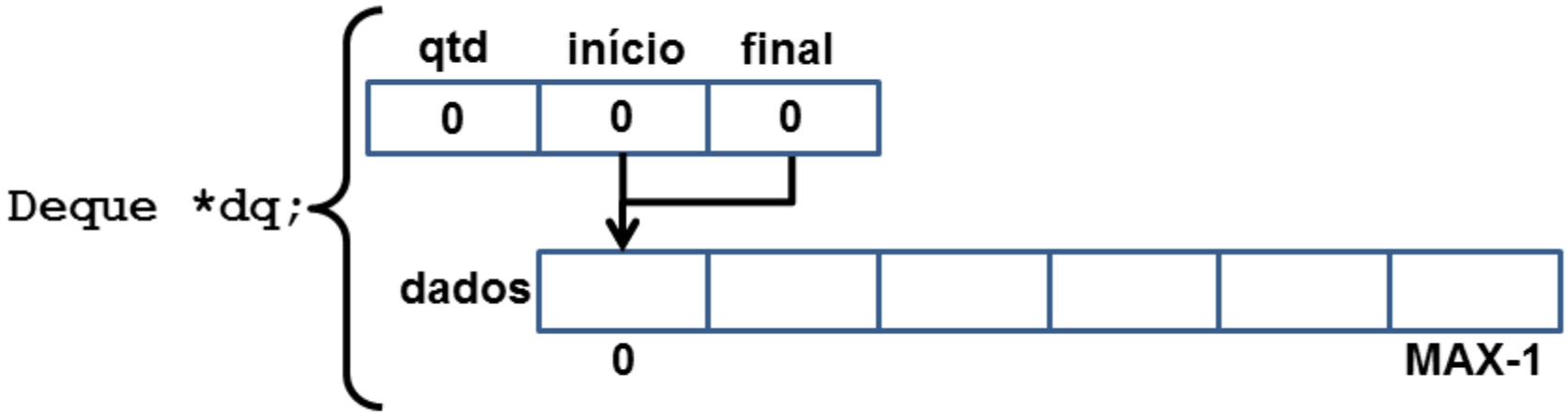


Deque Estático | Criação e liberação

- Criação
  - Aloca uma área de memória para o deque
  - Corresponde a memória necessária para armazenar a estrutura do deque
- Liberação
  - Basta liberar a memória alocada para a estrutura deque

```
Deque* cria_Deque(){
    Deque *dq;
    dq = (Deque*) malloc(sizeof(struct deque));
    if(dq != NULL){
        dq->inicio = 0;
        dq->final = 0;
        dq->qtd = 0;
    }
    return dq;
}

void libera_Deque(Deque* dq){
    free(dq);
}
```

Deque Estático | Criação e liberação

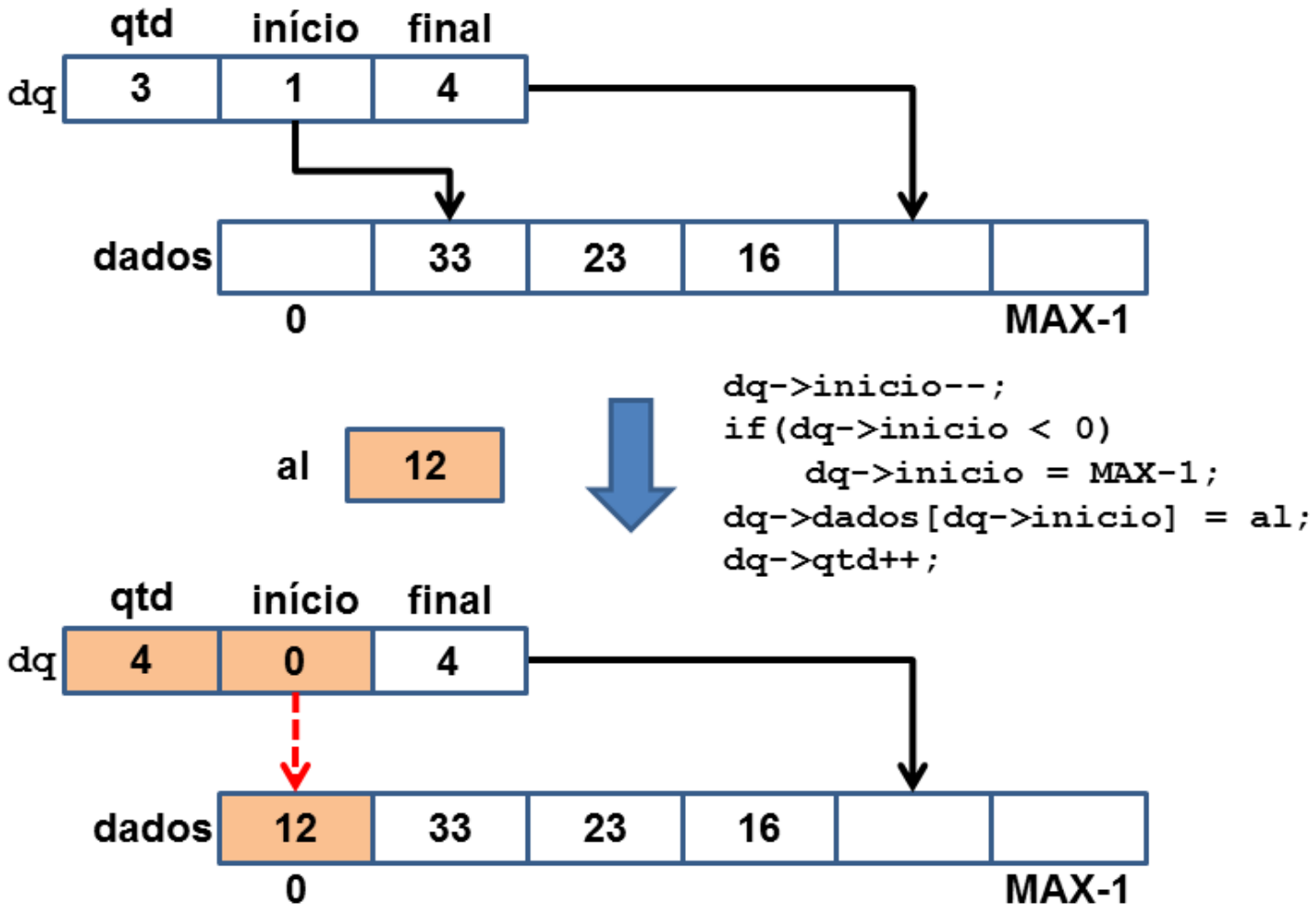


Deque Estático | Inserção no início

- Similar a inserção no final da Fila Estática
  - devemos manter a circularidade
- Precisamos verificar
  - se o deque existe
  - se o deque está cheio
- E só depois
  - mover o início
  - copiar os dados
  - incrementar a quantidade

```
int insereInicio_Deque(Deque* dq, struct aluno al){
    if(dq == NULL || dq->qtd == MAX)
        return 0;
    dq->inicio--;
    if(dq->inicio < 0)
        dq->inicio = MAX-1;
    dq->dados[dq->inicio] = al;
    dq->qtd++;
    return 1;
}
```

Deque Estático | Inserção no início

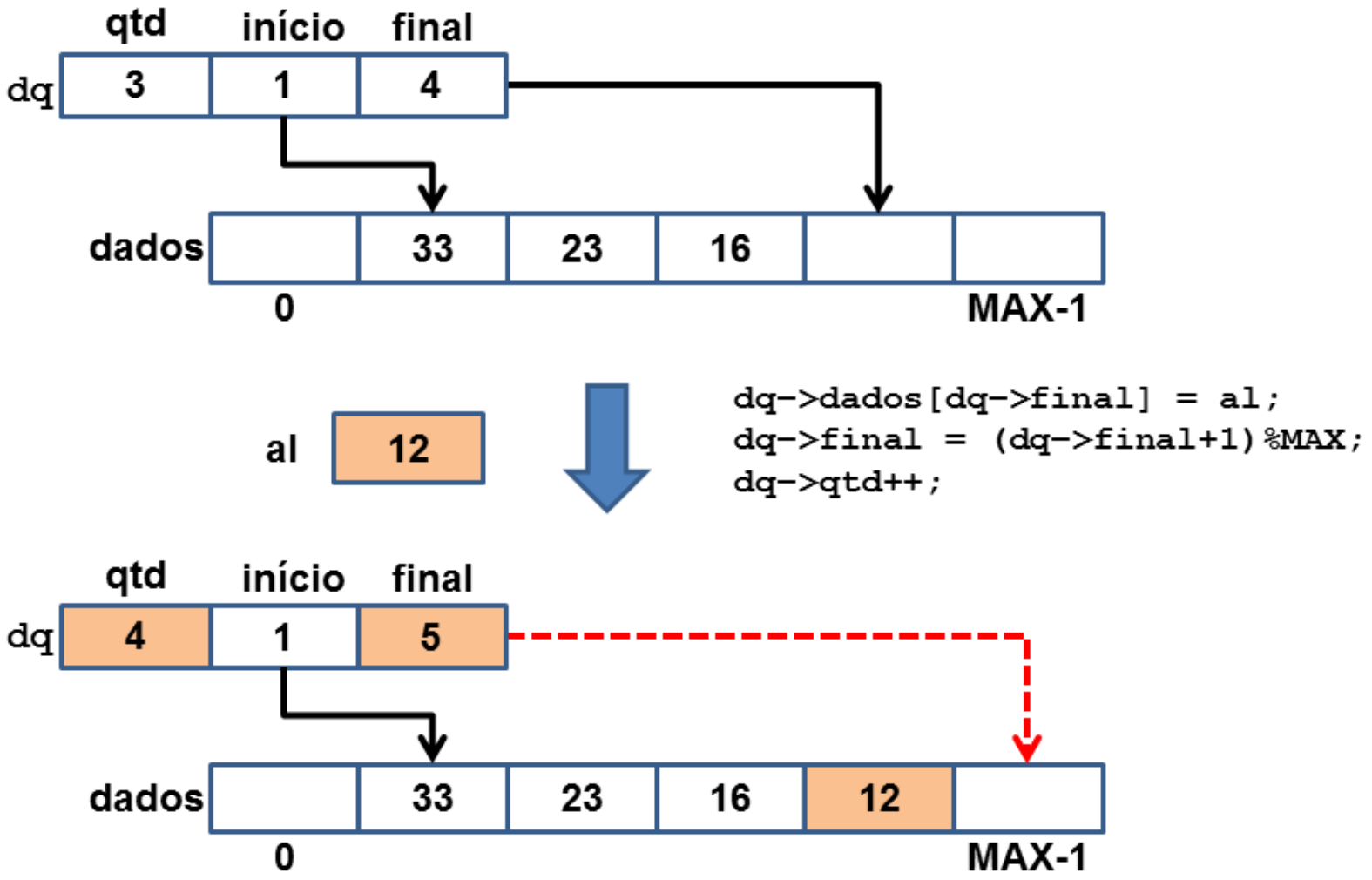


Deque Estático | Inserção no final

- Igual a inserção no final da Fila Estática
  - devemos manter a circularidade
- Precisamos verificar
  - se o deque existe
  - se o deque está cheio
- E só depois
  - copiar os dados
  - mover o final
  - incrementar a quantidade

```
int insereFinal_Deque(Deque* dq, struct aluno al){
    if(dq == NULL || dq->qtd == MAX)
        return 0;
    dq->dados[dq->final] = al;
    dq->final = (dq->final+1)%MAX;
    dq->qtd++;
    return 1;
}
```

Deque Estático | Inserção no final



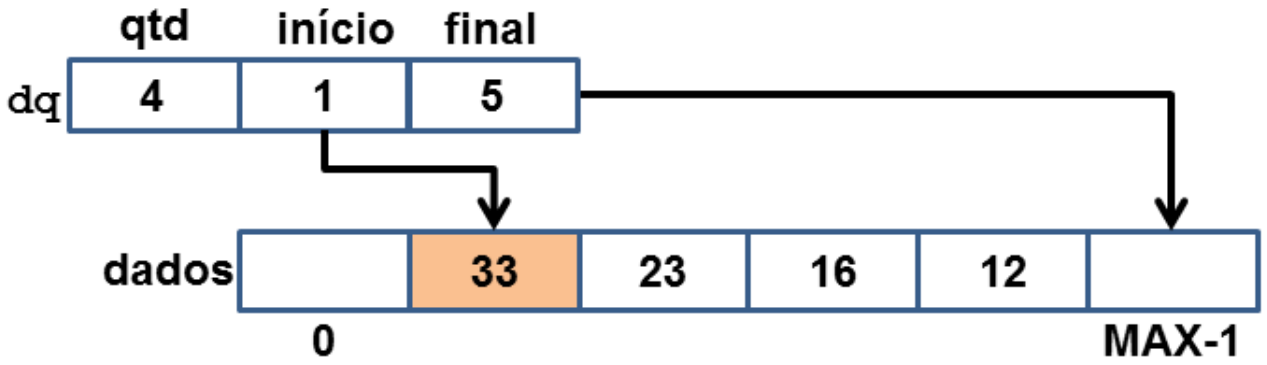
Deque Estático | Remoção do início

- Igual a remoção do início da Fila Estática
  - devemos manter a circularidade

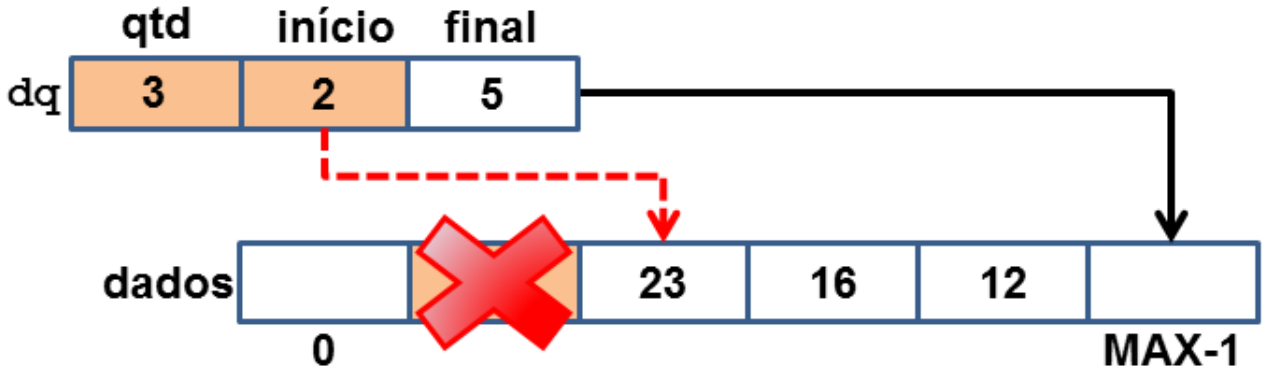
- Precisamos verificar
  - se o deque existe
  - se o deque não está vazio
- E só depois
  - mover o início
  - diminuir a quantidade

```
int removeInicio_Deque(Deque* dq){
    if(dq == NULL || dq->qtd == 0)
        return 0;
    dq->inicio = (dq->inicio+1)%MAX;
    dq->qtd--;
    return 1;
}
```

Deque Estático | Remoção do início




```
dq->inicio = (dq->inicio+1)%MAX;
dq->qtd--;
```



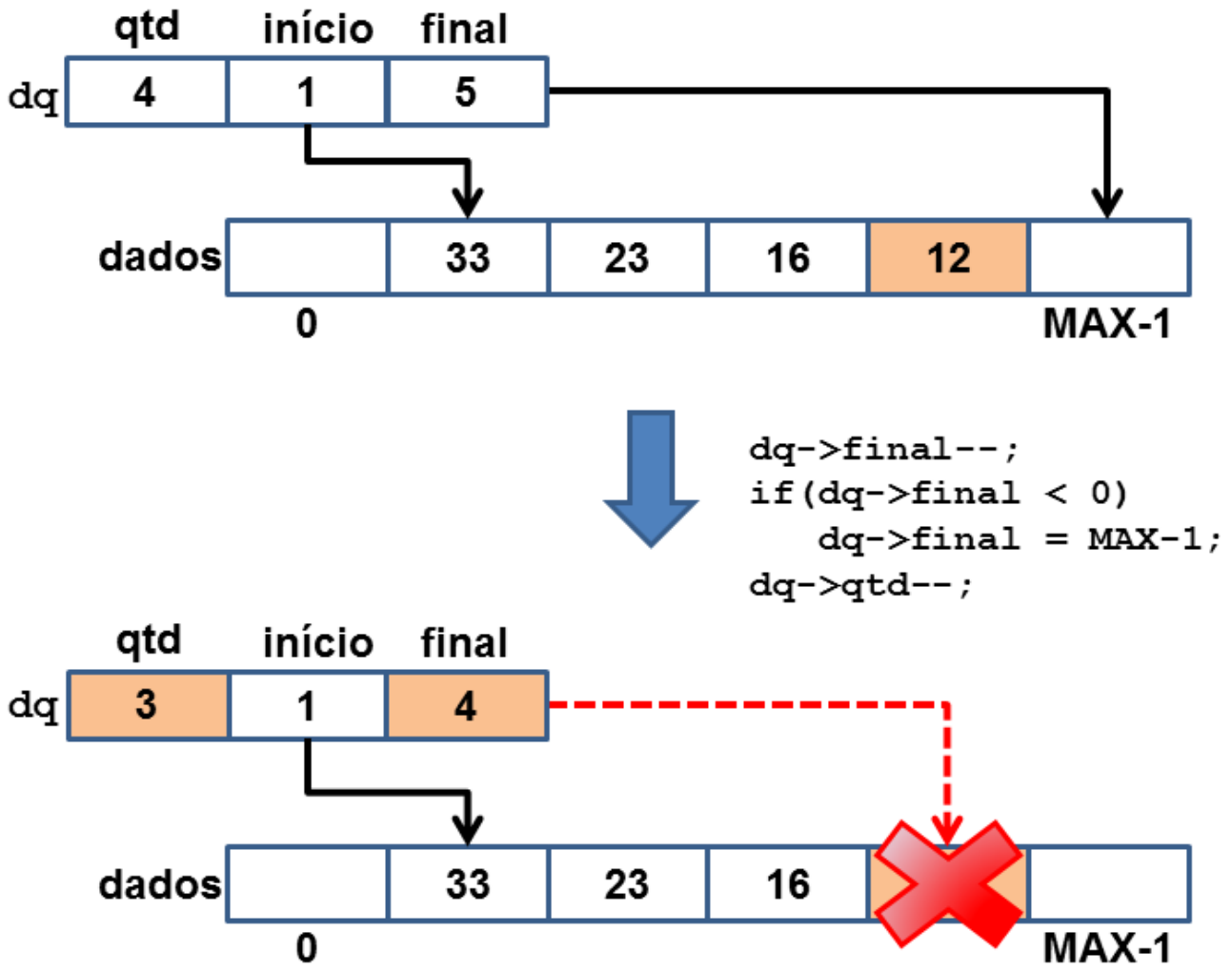
Deque Estático | Remoção do final

- Similar a remoção do início da Fila Estática
  - devemos manter a circularidade

- Precisamos verificar
  - se o deque existe
  - se o deque não está vazio
- E só depois
  - mover o final
  - verificar a circularidade
  - diminuir a quantidade

```
int removeFinal_Deque(Deque* dq){
    if(dq == NULL || dq->qtd == 0)
        return 0;
    dq->final--;
    if(dq->final < 0)
        dq->final = MAX-1;
    dq->qtd--;
    return 1;
}
```

Deque Estático | Remoção do final

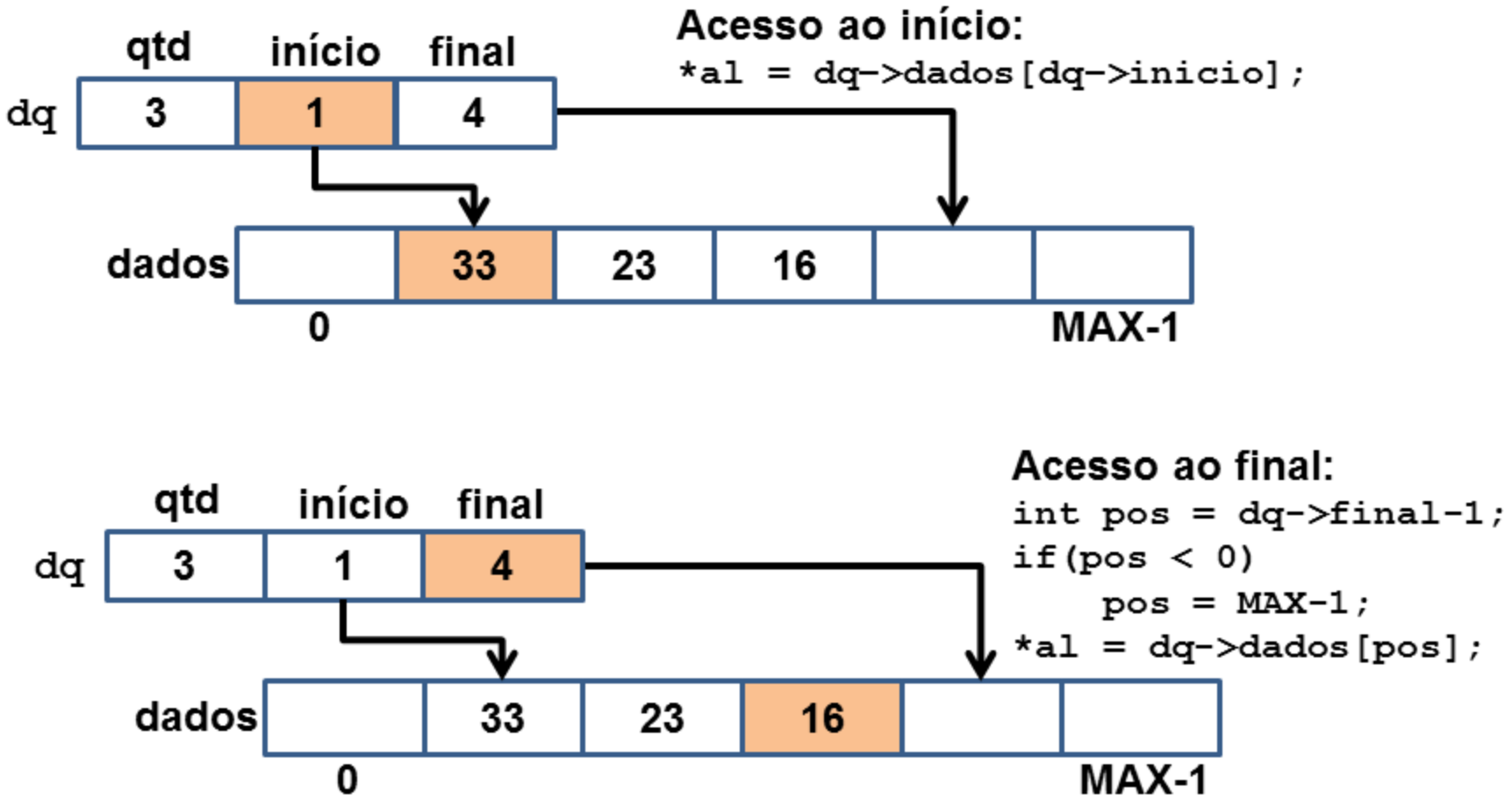


Deque Estático | Acesso

- Podemos acessar ambas as extremidades do deque
  - início: o acesso é imediato
  - final: sempre aponta para uma posição vaga. Verificar a circularidade

```
int consultaInicio_Deque(Deque* dq, struct aluno *al){
    if(dq == NULL || dq->qtd == 0)
        return 0;
    *al = dq->dados[dq->inicio];
    return 1;
}

int consultaFinal_Deque(Deque* dq, struct aluno *al){
    if(dq == NULL || dq->qtd == 0)
        return 0;
    int pos = dq->final-1;
    if(pos < 0)
        pos = MAX-1;
    *al = dq->dados[pos];
    return 1;
}
```

Deque Estático | Acesso



DEQUE DINÂMICO COM ACESSO DUPLAMENTE ENCADEADO

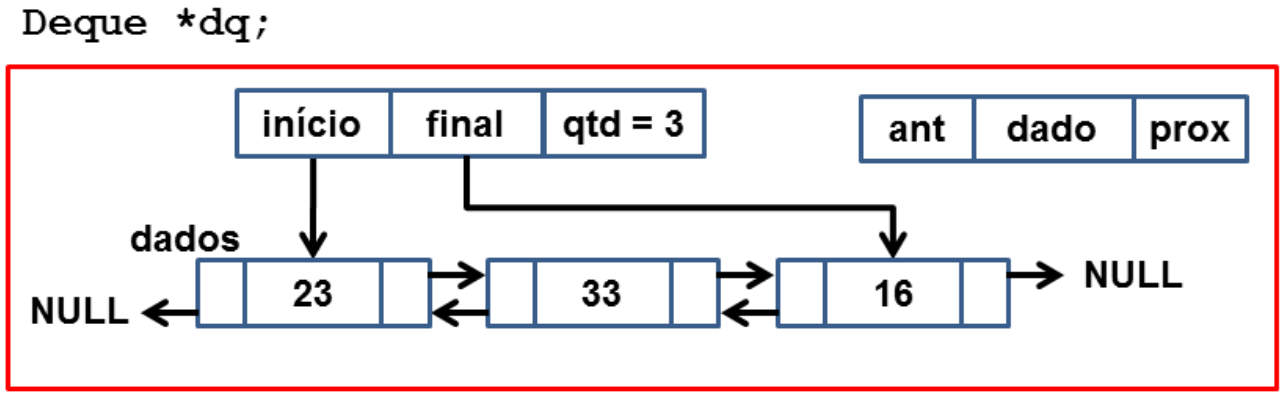
Deque Dinâmico | TAD

- Deque Dinâmico com acesso duplamente encadeado
- Cada elemento é alocado e liberado conforme a necessidade
  - Vantagem: melhor uso dos recursos de memória
  - Desvantagem: necessidade de percorrer todo o deque para destruí-lo

```c
//Definição do tipo Deque
struct elemento{
    struct elemento *ant;
    struct aluno dados;
    struct elemento *prox;
};
typedef struct elemento Elem;

//Definição do Nó Descritor do Deque
struct deque{
    struct elemento *inicio;
    struct elemento *final;
    int qtd;
};
typedef struct deque Deque;
```

Deque Dinâmico | TAD

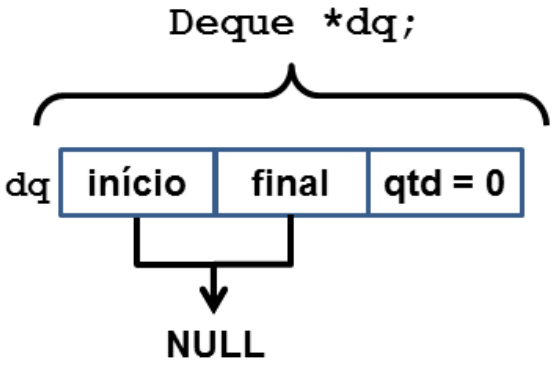
- Cada elemento possui
  - um campo de dado
  - ponteiro para o anterior
  - ponteiros para o próximo
- Utiliza um nó descritor para guardar
  - início
  - final
  - quantidade de elementos



Deque Dinâmico | Criação e liberação

- Aloca uma área de memória para a estrutura deque
- Inicializa os ponteiro para NULL e quantidade com zero (deque vazio)

```
Deque* cria_Deque(){
    Deque* dq = (Deque*) malloc(sizeof(struct deque));
    if(dq != NULL){
        dq->final = NULL;
        dq->inicio = NULL;
        dq->qtd = 0;
    }
    return dq;
}
```



Deque Dinâmico | Criação e liberação

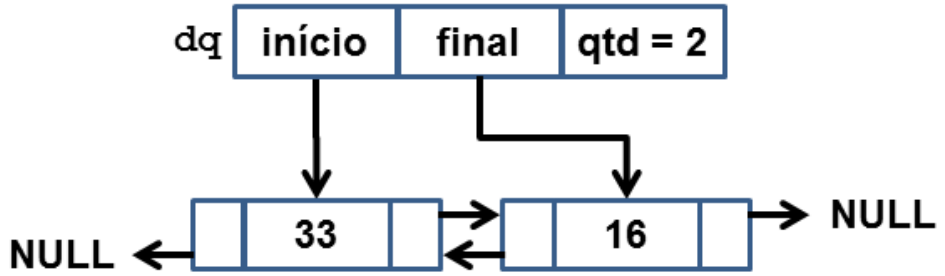
- Para liberar um deque dinâmico é preciso percorrer toda o deque liberando a memória alocada para cada elemento inserido

- Ao final, liberamos a memória do deque em si

```
void libera_Deque(Deque* dq){
    if(dq != NULL){
        Elem* no;
        while(dq->inicio != NULL){
            no = dq->inicio;
            dq->inicio = dq->inicio->prox;
            free(no);
        }
        free(dq);
    }
}
```

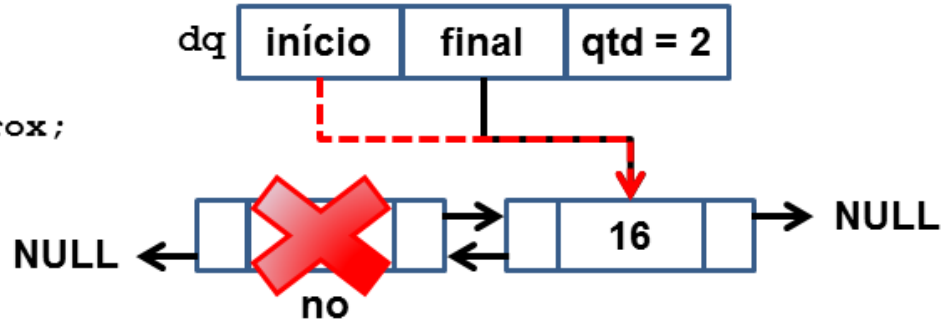
Deque Dinâmico | Criação e liberação

Deque inicial



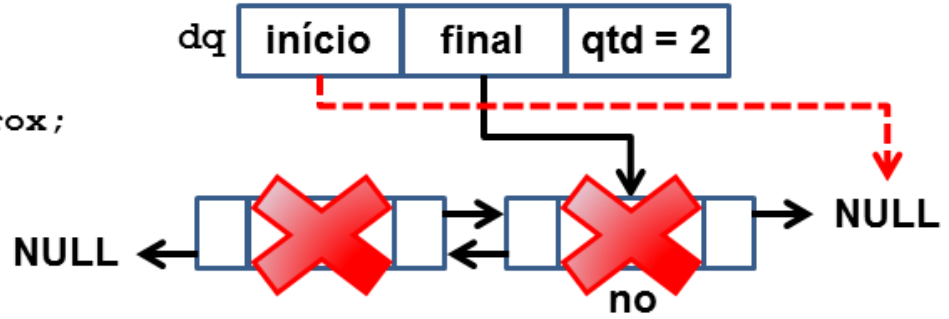
Passo 1:

```
no = dq->inicio;
dq->inicio = dq->inicio->prox;
free(no);
```



Passo 2:

```
no = dq->inicio;
dq->inicio = dq->inicio->prox;
free(no);
```



Deque Dinâmico | Inserção no início

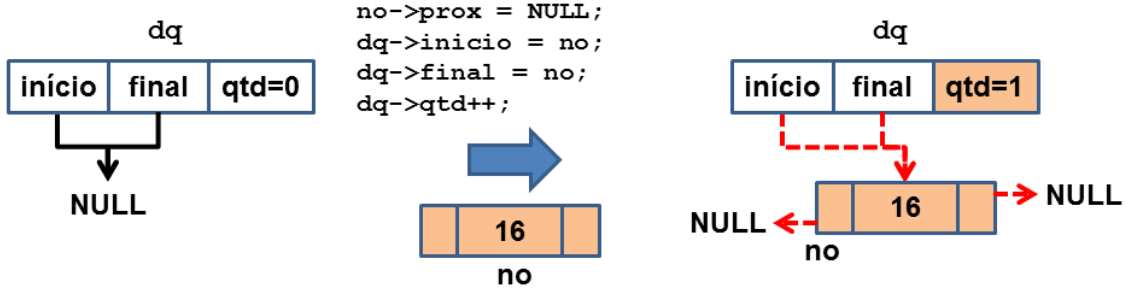
- Envolve alocar espaço para o novo elemento e ajustar alguns ponteiros
- Se o deque existe, temos que
  - alocar memória para o novo nó
  - copiar os dados
  - ajustar ponteiros
    - mudar início
    - anterior do nó é NULL
    - anterior do início é o nó
  - incrementar a quantidade

```
int insereInicio_Deque(Deque* dq, struct aluno al){
    if(dq == NULL)
        return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    no->prox = dq->inicio;
    no->ant = NULL;
    if(dq->inicio == NULL)
        dq->final = no;
    else//Deque não vazio: apontar para o anterior!
        dq->inicio->ant = no;
    dq->inicio = no;
    dq->qtd++;
    return 1;
}
```

Deque Dinâmico | Inserção no início

Insere em um deque vazio

```
no->ant = NULL;
no->prox = NULL;
dq->inicio = no;
dq->final = no;
dq->qtd++;
```



Insere no início:

```
no->prox = dq->inicio;
no->ant = NULL;
```

Se o deque está vazio:

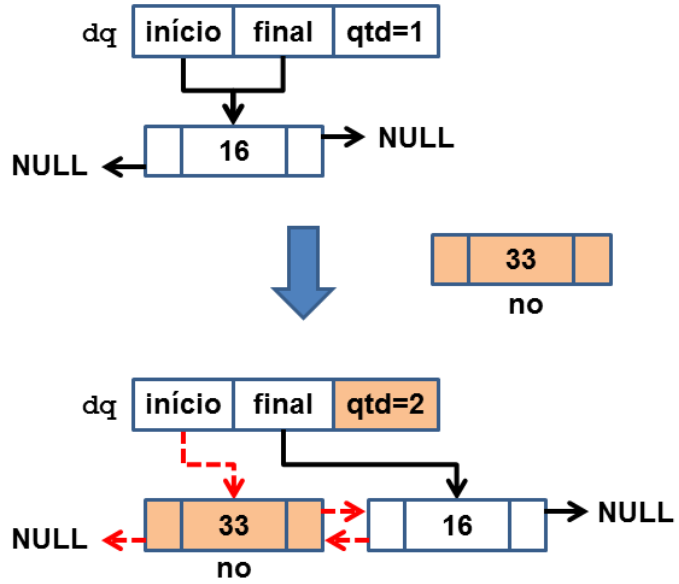
```
dq->final = no;
```

Se o deque NÃO está vazio:

```
dq->inicio->ant = no;
```

Por fim:

```
dq->inicio = no;
dq->qtd++;
```



Deque Dinâmico | Inserção no final

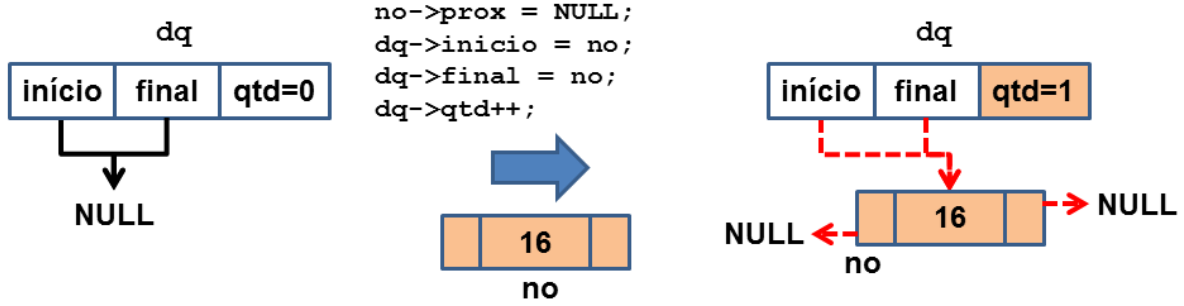
- Envolve alocar espaço para o novo elemento e ajustar alguns ponteiros
- Se o deque existe, temos que
  - alocar memória para o novo nó
  - copiar os dados
  - ajustar ponteiros
    - mudar final
    - próximo do nó é NULL
    - anterior do final é o nó
  - incrementar a quantidade

```
int insereFinal_Deque(Deque* dq, struct aluno al){
    if(dq == NULL)
        return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    no->prox = NULL;
    if(dq->final == NULL){//Deque vazio
        no->ant = NULL;
        dq->inicio = no;
    }
    else{
        no->ant = dq->final;
        dq->final->prox = no;
    }
    dq->final = no;
    dq->qtd++;
    return 1;
}
```

Deque Dinâmico | Inserção no final

Insere em um deque vazio

```
no->ant = NULL;
no->prox = NULL;
dq->inicio = no;
dq->final = no;
dq->qtd++;
```



Insere no final:

```
no->prox = NULL;
```

Se o deque está vazio:

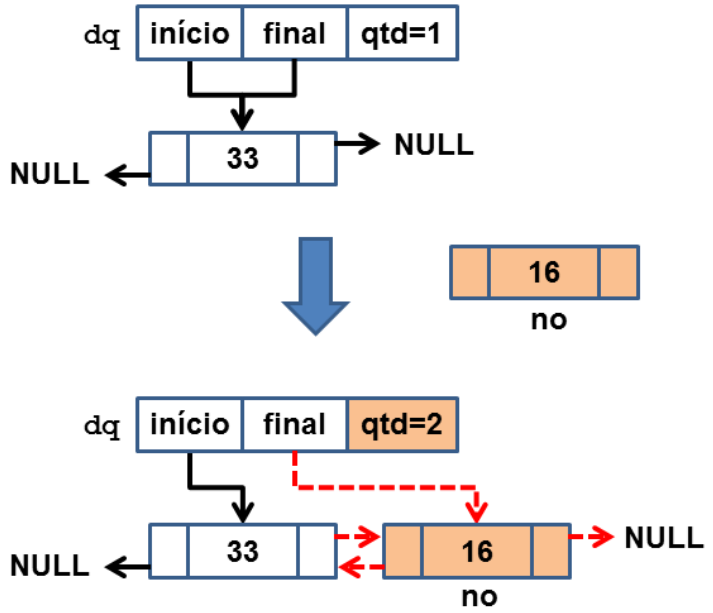
```
no->ant = NULL;
dq->inicio = no;
```

Se o deque NÃO está vazio:

```
no->ant = dq->final;
dq->final->prox = no;
```

Por fim:

```
dq->final = no;
dq->qtd++;
```



Deque Dinâmico | Remoção do início

- Envolve liberar a memória do elemento removido e ajustar alguns ponteiros

- Primeiro, verificamos se
  - o deque existe
  - o deque possui elementos

- E só depois
  - ajustar ponteiros
  - verificar se o deque ficou vazio
  - liberar a memória do nó
  - diminuir a quantidade

```
int removeInicio_Deque(Deque* dq){
    if(dq == NULL)
        return 0;
    if(dq->inicio == NULL)//Deque vazio
        return 0;
    Elem *no = dq->inicio;
    dq->inicio = dq->inicio->prox;
    if(dq->inicio == NULL)//Deque ficou vazio
        dq->final = NULL;
    else
        dq->inicio->ant = NULL;
    free(no);
    dq->qtd--;
    return 1;
}
```

Deque Dinâmico | Remoção do início

Remoção do início:

```
no = dq->inicio;
dq->inicio = dq->inicio->prox
```

Se o deque ficou vazio:

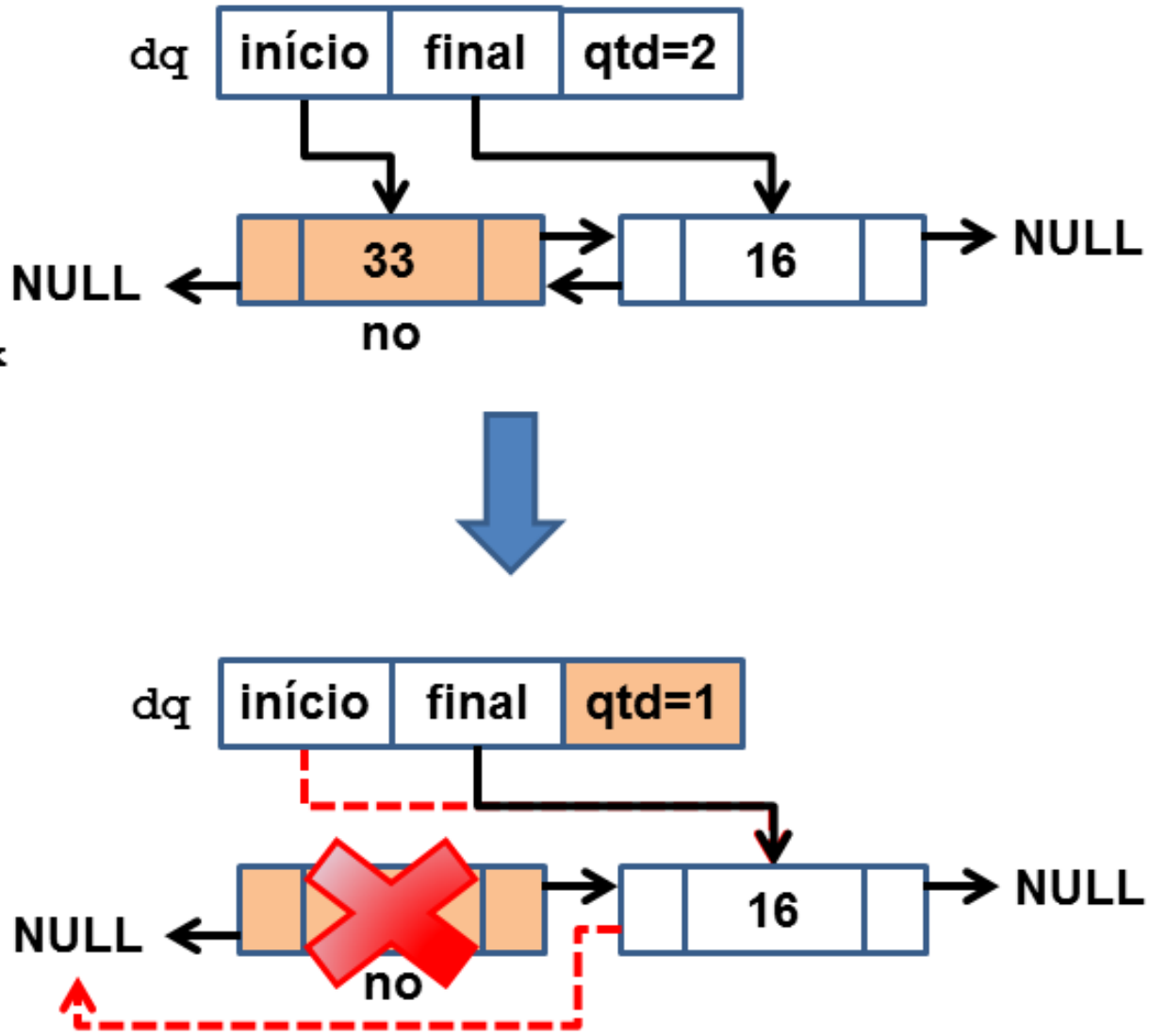
```
dq->final = NULL;
```

Se o deque NÃO ficou vazio:

```
dq->inicio->ant = NULL;
```

Por fim:

```
free(no);
dq->qtd--;
```



Deque Dinâmico | Remoção do final

- Envolve liberar a memória do elemento removido e ajustar alguns ponteiros
- Primeiro, verificamos se
  - o deque existe
  - o deque possui elementos
- E só depois
  - verificar se o deque ficará vazio
  - ajustar ponteiros
  - liberar a memória do nó
  - diminuir a quantidade

```
int removeFinal_Deque(Deque* dq){
    if(dq == NULL)
        return 0;
    if(dq->inicio == NULL)//Deque vazio
        return 0;
    Elem *no = dq->final;

    if(no == dq->inicio){//remover o primeiro?
        dq->inicio = NULL;
        dq->final = NULL;
    }else{
        no->ant->prox = NULL;
        dq->final = no->ant;
    }
    free(no);
    dq->qtd--;
    return 1;
}
```

Deque Dinâmico | Remoção do final

Remoção do final:

```
no = dq->final;
```

Se "no" é o primeiro:

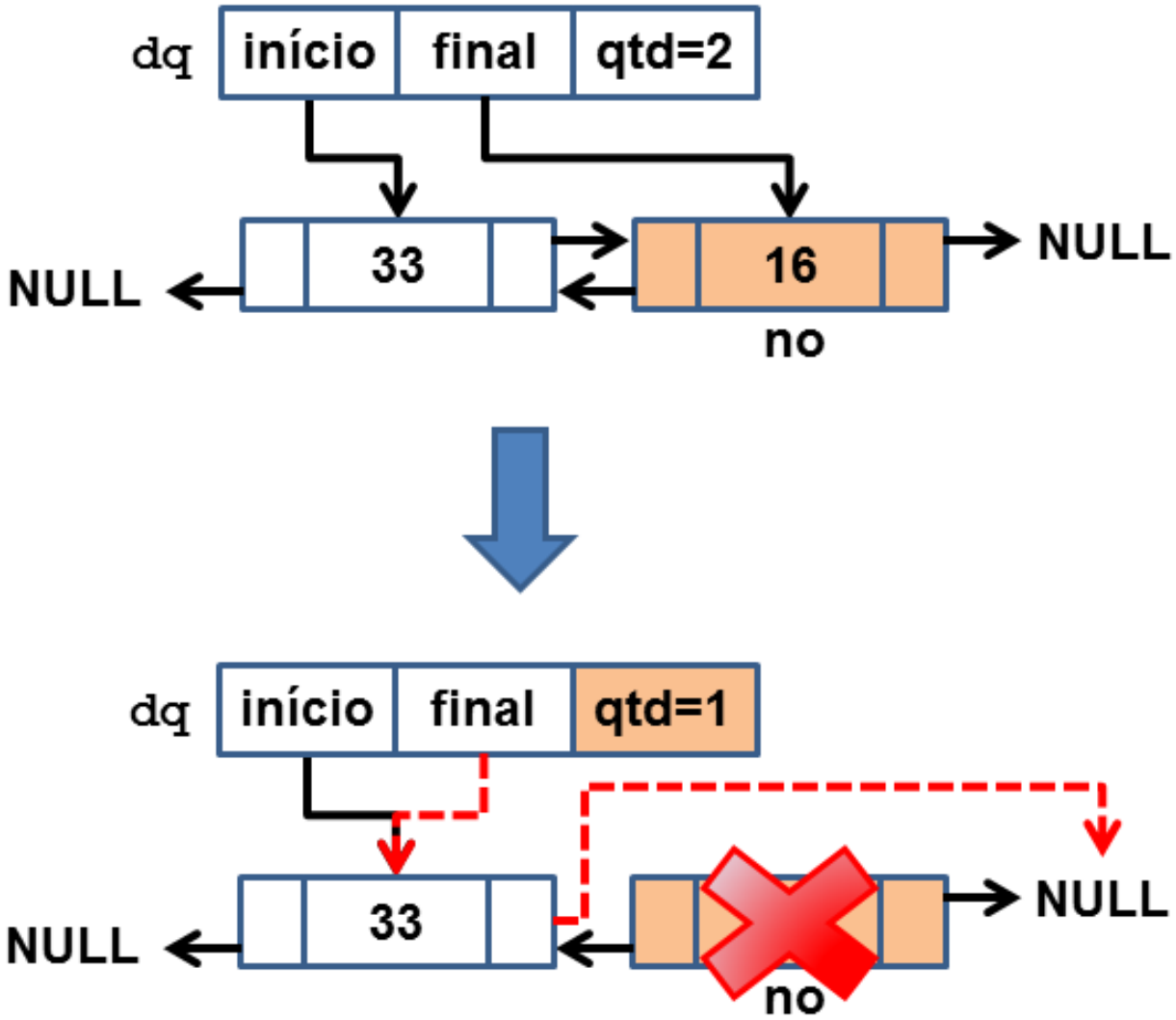
```
dq->inicio = NULL;
dq->final = NULL;
```

Se "no" NÃO é o primeiro:

```
no->ant->prox = NULL;
dq->final = no->ant;
```

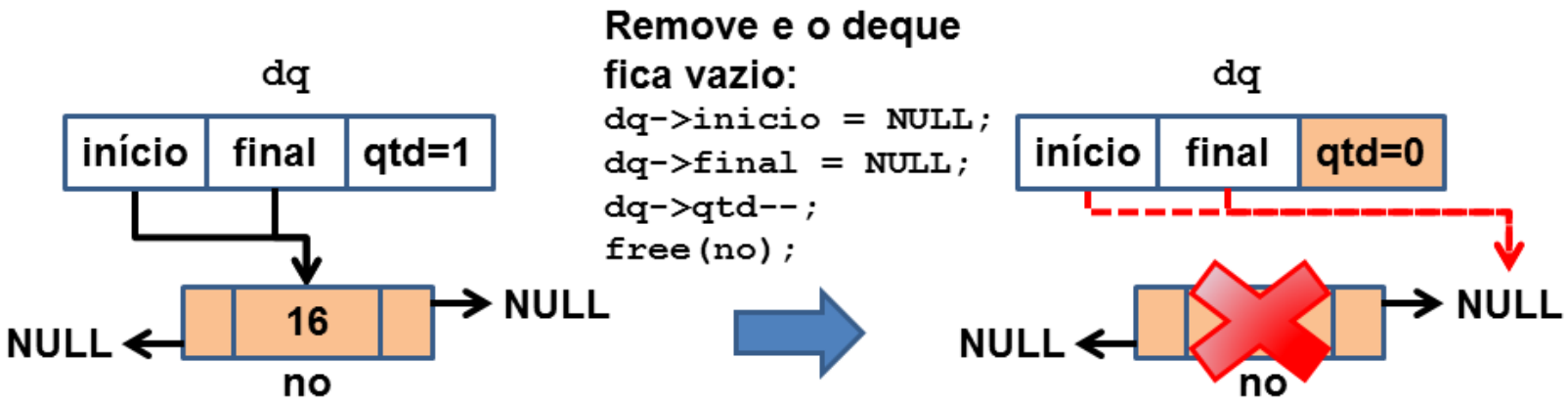
Por fim:

```
free(no);
dq->qtd--;
```



Deque Dinâmico | Remoção do final

- Remoção deixa o deque vazio

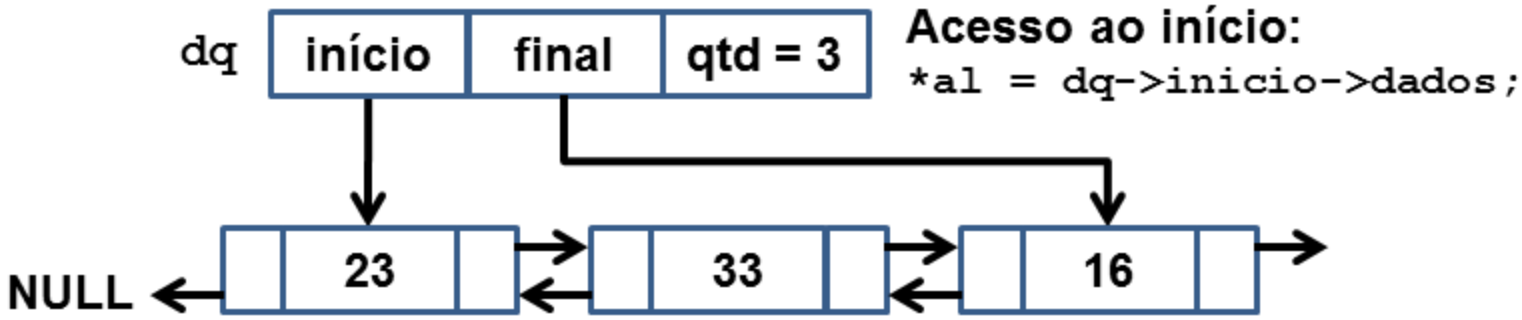


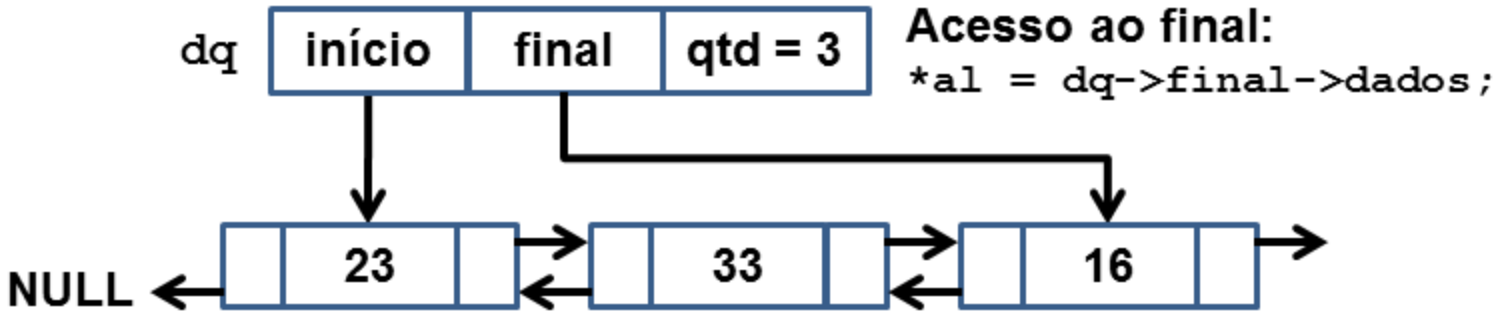
Deque Dinâmico | Acesso

- Podemos acessar ambas as extremidades do deque
  - Em ambos os casos acesso é imediato

```
int consultaInicio_Deque(Deque* dq, struct aluno *al){
    if(dq == NULL)
        return 0;
    if(dq->inicio == NULL)//Deque vazio
        return 0;
    *al = dq->inicio->dados;
    return 1;
}

int consultaFinal_Deque(Deque* dq, struct aluno *al){
    if(dq == NULL)
        return 0;
    if(dq->final == NULL)//Deque vazio
        return 0;
    *al = dq->final->dados;
    return 1;
}
```

Deque Dinâmico | Acesso





Material Complementar | Vídeo Aulas

- Aula 85 – Fila de Prioridades: Definição
  - https://youtu.be/cwi5U5jaBeI
- Aula 86 – Fila de Prioridades: Implementação
  - https://youtu.be/1llU1xKJqxM
- Aula 87 – Fila de Prioridades: Array Ordenado
  - https://youtu.be/j4XgBORyCok
- Aula 88 – Fila de Prioridades - Heap Binária
  - https://youtu.be/o138_fb85zk
- Aula 115 – Deque: Definição
  - https://youtu.be/0MxItm1z5xU

Material Complementar | Vídeo Aulas

- Aula 116 – Deque Estático
  - https://youtu.be/EqYAERvAnyQ
- Aula 117 – Deque Estático: Informações e Consulta
  - https://youtu.be/uP_8hjxNslg
- Aula 118 – Deque Estático: Inserção e Remoção
  - https://youtu.be/5Ah7Fg4vIkw
- Aula 119 - Deque Dinâmico
  - https://youtu.be/_n1nU_IE9QE
- Aula 120 - Deque Dinâmico - Informações e consulta
  - https://youtu.be/JEDIHRjVOUI
- Aula 121 - Deque Dinâmico: Inserção e Remoção
  - https://youtu.be/rqU0ZaRyGoM

Material Complementar | GitHub

- https://github.com/arbackes