

Fila de Prioridade – usando Heap

Filas de prioridades são estruturas de dados que gerenciam um conjunto de elementos, cada um com uma prioridade associada.

Dentre as operações previstas numa fila de prioridade estão:

- inserção de um elemento;
- exclusão do elemento de prioridade máxima;
- aumento de prioridade de um elemento;
- redução de prioridade de um elemento;
- consulta da prioridade de um elemento;
- consulta à quantidade de elementos (tamanho) da fila.

É desejável que todas estas operações sejam realizadas de maneira eficiente (**e neste EP a maior complexidade dessas operações será logarítmica**).

A eficiência dessas operações dependerá da maneira que a fila de prioridade foi implementada (e de sua estrutura subjacente).

Para este EP, vocês deverão implementar um conjunto de funções de gerenciamento de filas de prioridade utilizando principalmente dois conceitos: **heap máximo** e um **arranjo auxiliar de ponteiros para elementos**. Heaps máximos são árvores binárias que seguem uma regra de organização diferente das árvores binárias de busca (ou de pesquisa): **todo nó tem sempre sua chave (no caso, sua prioridade) maior ou igual a de seus filhos**. Diferentemente das implementações de árvores que vimos ao longo do semestre, os **heaps costumam ser implementados em arranjos** (no material complementar ao EP há material específico para detalhamento de heaps máximos).

A seguir, serão apresentadas as estruturas de dados envolvidas nesta implementação e como elas serão gerenciadas.

A estrutura básica será o **ELEMENTO**, que contém três campos: *id* (identificador inteiro do elemento), *prioridade* (número do tipo *float* com a prioridade do elemento), *posicao* (posição do elemento no arranjo correspondente ao heap).

```
typedef struct {  
    int id;  
    float prioridade;  
    int posicao;  
} ELEMENTO, * PONT;
```

id
prioridade
posicao

A estrutura **FILADEPRIORIDADE** possui quatro campos: *maxElementos* é um campo do tipo inteiro que representa a quantidade máxima de elementos permitidos na fila de prioridade atual (os *ids* válidos dos elementos valerão de 0 [zero] até *maxElementos*-1); *elementosNoHeap* é um campo do tipo inteiro que representa a quantidade de elementos efetivamente no heap; *heap* corresponde a um ponteiro para um **arranjo de ponteiros para elementos** do tipo *ELEMENTO*, este arranjo representará o heap máximo, seu tamanho será igual a *maxElementos* e o campo *elementosNoHeap* determinará quantos elementos válidos estão no heap em um dado momento; *arranjo* corresponde a um ponteiro para um **arranjo de ponteiros para elementos** do tipo *ELEMENTO* (este último arranjo é equivalente ao arranjo utilizado no EP1).

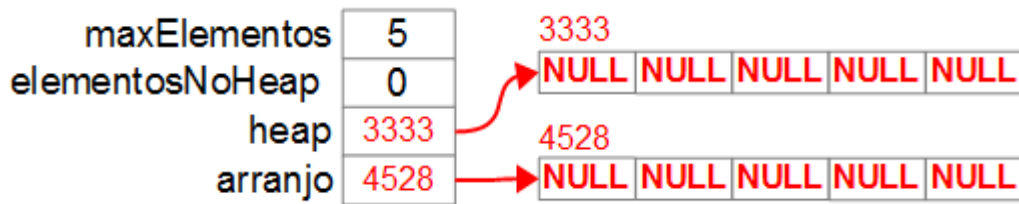
Na inicialização de uma fila de prioridades, tanto o arranjo apontado por *heap* quanto o apontado por *arranjo* são criados com todos seus valores valendo *NULL*. Já que os *ids* válidos variam de 0 a *maxElementos*-1, então há uma posição específica para guardar o endereço de cada *ELEMENTO* (quando ele for criado) no arranjo *arranjo*, permitindo acesso rápido a um elemento qualquer a partir de seu respectivo *id*.

typedef struct {	maxElementos	<input type="text"/>
int maxElementos;	elementosNoHeap	<input type="text"/>
int elementosNoHeap;	heap	<input type="text"/>
PONT* heap;	arranjo	<input type="text"/>
PONT* arranjo;		
} FILADEPRIORIDADE, * PFILA;		

A função *criarFila* é responsável por criar uma nova fila de prioridade que poderá ter até *maxElementos* e deve retornar o endereço dessa fila de prioridades. Observe que os dois arranjos de ponteiros para elementos já são criados e têm seus valores inicializados nessa função.

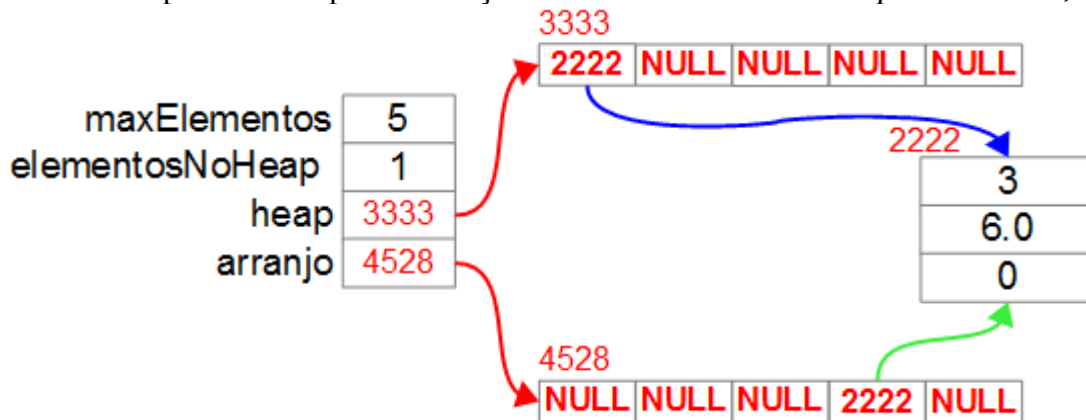
```
PFILA criarFila(int max){
    PFILA res = (PFILA) malloc(sizeof(FILADEPRIORIDADE));
    res->maxElementos = max;
    res->arranjo = (PONT*) malloc(sizeof(PONT)*max);
    res->heap = (PONT*) malloc(sizeof(PONT)*max);
    int i;
    for (i=0;i<max;i++) {
        res->arranjo[i] = NULL;
        res->heap[i] = NULL;
    }
    res->elementosNoHeap = 0;
    return res;
}
```

Exemplo de fila de prioridade recém criada com *maxElementos* = 5:

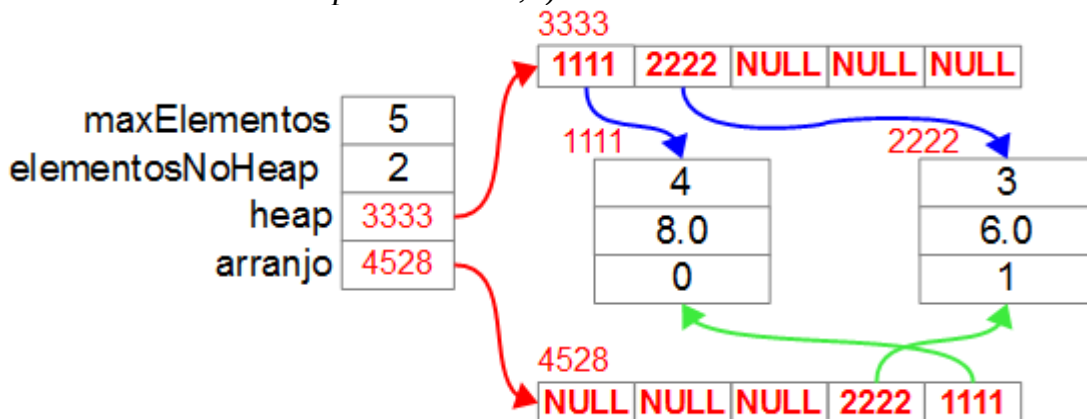


Ao se inserir um novo elemento na estrutura, este deverá ter seu endereço incluído no heap máximo e também deverá ter seu endereço armazenado na respectiva posição do arranjo (apontado pelo campo *arranjo*).

Exemplo de fila de prioridade após a inserção do elemento com *id*=3 e *prioridade*=6,0:



Exemplo de fila de prioridade após a inserção do elemento de *id*=4 e *prioridade*=8,0 (depois da inserção do elemento de *id*=3 e *prioridade*=6,0).



Ao se modificar a prioridade de um elemento da fila de prioridade, este deverá ser reposicionado dentro do heap máximo, caso necessário.

Notem que cada elemento possuirá uma única cópia em memória e será apontado/referenciado em dois lugares: em uma das posições do arranjo apontado por *heap* e por uma das posições do arranjo apontado por *arranjo*.

Funções que deverão ser implementadas no EP

int tamanho(PFILA f): função que recebe o endereço de uma fila de prioridade e retorna o número de elementos na fila (ou mais precisamente, o número de elementos válidos no heap máximo).

bool inserirElemento(PFILA f, int id, float prioridade): função que recebe o endereço de uma fila de prioridade, o identificador do novo elemento e o valor de sua prioridade.

Esta função deverá retornar *false* caso:

- o identificador seja inválido (menor que zero ou maior ou igual a *maxElementos*);
- o identificador seja válido, mas já houver um elemento com esse identificador na fila.

Caso contrário, a função deverá alocar memória para esse novo elemento, colocar o endereço dele no arranjo de elementos e colocar o endereço dele na posição correta do heap máximo (de acordo com sua prioridade), acertando todos os ponteiros necessários, o valor do campo *elementosNoHeap*, o campo *posicao* do novo elemento (e de qualquer elemento que mude de posição no heap durante esta inserção) e retornar *true*.

bool aumentarPrioridade(PFILA f, int id, float novaPrioridade): função que recebe o endereço de uma fila de prioridade, o identificador do elemento e o novo valor de sua prioridade.

Esta função deverá retornar *false* caso:

- o identificador seja inválido (menor que zero ou maior ou igual a *maxElementos*);
- o identificador seja válido, mas não haja um elemento com esse identificador na fila.
- o identificador seja válido, mas sua prioridade já seja maior ou igual à nova prioridade

passada como parâmetro da função.

Caso contrário, a função deverá atualizar a prioridade do elemento, reposicioná-lo (se necessário [lembre-se de atualizar também o valor do campo *posicao*, se necessário, tanto do elemento que teve sua prioridade aumentada mas também de qualquer outro que mudou de posição por causa desse aumento]) dentro da fila de prioridade (no heap máximo) e retornar *true*. Observação: esta função **não** deverá criar um novo elemento.

bool reduzirPrioridade(PFILA f, int id, float novaPrioridade): função que recebe o endereço de uma fila de prioridade, o identificador do elemento e o novo valor de sua prioridade.

Esta função deverá retornar *false* caso:

- o identificador seja inválido (menor que zero ou maior ou igual a *maxElementos*);
- o identificador seja válido, mas não haja um elemento com esse identificador na fila.
- o identificador seja válido, mas sua prioridade já seja menor ou igual à nova prioridade

passada como parâmetro da função.

Caso contrário, a função deverá atualizar a prioridade do elemento, reposicioná-lo (se necessário [lembre-se de atualizar também o valor do campo *posicao*, se necessário, tanto do elemento que teve sua prioridade reduzida mas também de qualquer outro que mudou de posição por causa dessa redução]) dentro da fila de prioridade (no heap máximo) e retornar *true*. Observação: esta função **não** deverá criar um novo elemento.

PONT removerElemento(PFILA f): função que recebe como parâmetro o endereço de uma fila de prioridade e deverá retornar *NULL* caso a fila esteja vazia. Caso contrário, deverá retirar o primeiro elemento do heap máximo, reorganizar o heap (acertando os elementos [isto é, o

campo *posicao* dos respectivos elementos] e ponteiros necessários), colocar o valor *NULL* na posição correspondente desse elemento no arranjo *arranjo*, acertar o valor do campo *elementosNoHeap* e retornar o endereço do respectivo elemento. A memória desse elemento não deverá ser apagada, pois o usuário pode querer usar esse elemento para alguma outra coisa.

bool consultarPrioridade(PFILA f, int id, float resposta)*: função que recebe o endereço de uma fila de prioridade, o identificador do elemento e um endereço para uma memória do tipo *float*.

Esta função deverá retornar *false* caso:

- o identificador seja inválido (menor que zero ou maior ou igual a *maxElementos*);
- o identificador seja válido, mas não haja um elemento com esse identificador na fila.

Caso contrário, a função deverá colocar na memória apontada pela variável *resposta* o valor da prioridade do respectivo elemento e retornar *true*.

Informações gerais:

Os EPs desta disciplina são trabalhos individuais que devem ser submetidos pelos alunos via sistema TIDIA (ae4.tidia-ae.usp.br/) até as 23:55h (com margem de tolerância de 60 minutos).

Você receberá três arquivos para este EP:

- *filaDePrioridade.h* que contém a definição das estruturas, os *includes* necessários e o cabeçalho/assinatura das funções. Você não deverá alterar esse arquivo.
- *filaDePrioridade.c* que conterá a implementação das funções solicitadas (e funções adicionais, caso julgue necessário). Este arquivo já contém um cabeçalho, o esqueleto geral das funções e alguns códigos implementados.
- *usaFilaDePrioridade.c* que contém alguns testes executados sobre as funções implementadas.

Você deverá submeter **apenas** o arquivo *filaDePrioridade.c*, porém renomeie este arquivo para *<seuNúmeroUSP>.c* (por exemplo, *12345678.c*) antes de submeter.

Não altere a assinatura de nenhuma das funções e não altere as funções originalmente implementadas (*exibirLog*, *criarFila*, etc).

Nenhuma das funções que você implementará deverá imprimir algo. Para *debugar* o programa você pode imprimir coisas, porém, na versão a ser entregue ao professor, suas funções não deverão imprimir nada (exceto pela função *exibirLog* que já imprime algumas informações).

Você poderá criar novas funções (auxiliares), mas não deve alterar o arquivo *filaDePrioridade.h*. Adicionalmente, saiba que seu código será testado com uma versão diferente do arquivo *usaFilaDePrioridade.c*. Suas funções serão testadas individualmente e em conjunto.

Todos os trabalhos passarão por um processo de verificação de plágios. **Em caso de plágio, todos os alunos envolvidos receberão nota zero.**