O Heap é a Minha Esperança

Micael Levi Lima Cavalcante - 21554923

[1]Instituto da Computação – Universidade Federal do Amazonas (UFAM)
Av. General Rodrigo Octávio, 6200 – Coroado I – Manaus – AM – Brasil

mllc@icomp.ufam.edu.br

1. Experimento

1.1. Tipos de ordenação

Visto que um tipo abstrato de dado *(TAD)* abstrai qualquer linguagem de programação, devemos ter em mente um modelo de dados (conjunto) encapsulado e uma gama de operações necessárias para a manipulação desses dados (procedimentos) privados.

Então, como o objetivo deste experimento é comparar dois TADs, vamos antes definir uma interface para o TAD: **Fila de Prioridade**.

Tabela 1. descrição do TAD: Fila de Prioridade

construir	cria e inicializa uma instância da fila
destruir	libera espaço alocado para uma instância da fila
dado	define a estrutura de dados que manipulará a fila
enfileirar(x)	insere o elemento x na fila (de acordo com a sua prioridade)
desenfileirar	remove o elemento de maior prioridade (primeiro ou último) da fila
vazia	retorna verdadeiro se a fila não possuir elementos

A única diferença entre os dois tipos de `ordenação` (completa e parcial) que serão utilizados para manipular a estrutura acima, está na forma de como os elementos da fila são inseridos e removidos. Então podemos separar a parte da ordenação, que define o tipo de ordenação realizada sobre a fila, em outro TAD. Assim surge o TAD: **Ordenação Fila**. Então já temos os dois TADs:

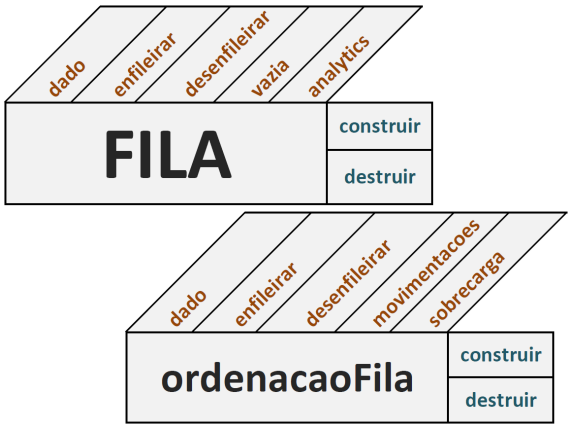

Figura 1. TADs: Fila de Prioridade & Ordenação Fila.

Tabela 2. descrição do TAD: Array Dinâmico Genérico

tamanho	retorna o tamanho do vetor
acessar(x)	retorna o elemento que está na posiçãoo x do vetor
atualizar(x)	altera o elemento que está na posição x do vetor
dado	estrutura que armazena o vetor e seu tamanho

Porém, antes de tudo, é preciso estabelecer que estrutura de dados representará a fila. Então criamos o TAD: **Array Dinâmico Genérico**. Sua interface pode ser definida como na tabela acima (2).

Dado a interface do TAD: **Ordenação Fila** (na figura 1), partimos para um nível de abstração mais baixo, o **nível de implementação**. Criamos dois códigos-fonte para definir os dois tipos de ordenação da fila. São eles que realmente criarão uma estrutura de dados para armazenar a fila e definirão a manipulação da mesma.

Começamos com a implementação da estrutura utilizada para cada tipo de ordenação:

```
/* === [ordenacaoCompleta.c] === */
typedef struct{
  TArrayDinamico *vetorFila;
  int posPrimeiro;
  int posUltimo;

}TDadoTAD;

/* === [ordencaoParcial.c] === */
#define PAI(i) (((i)-1)/2)

typedef struct{
  TArrayDinamico *vetorFila;
  int ocupacao;

}TDadoTAD;
```

O conceito utilizado na implementação da inserção da fila com ordenação completa foi o princípio da fila inversa. O elemento de maior prioridade será o último, assim, ao enfileirar algum elemento, inserimos no final e comparamos o mesmo com o anterior. Caso o anterior tenha maior prioridade, é preciso trocar os dois elementos (realizar o *swap*). A operação de enfileirar possuirá complexidade $O(n^2)$ no pior caso, i.e., possui um custo muito alto, por fazer uso do algoritmo de ordenação por inserção. Em contrapartida, a operação de desenfileirar o elemento de maior prioridade se torna trivial. Com isso podemos implementar (em linguagem C) da seguinte forma:

```
/* === [ordenacaoCompleta.c] === */
// ...
static short _enfileirar(TTAD* t, void* elemento){
  TDadoTAD *d = t->dado;
  if(!elemento || !d) return 0;

  int posInsercao = d->posUltimo + 1, i;
```

```
  TArrayDinamico *vet = d->vetorFila;

  d->posUltimo = posInsercao;

  if(!posInsercao){
    d->posPrimeiro = posInsercao;
    vet->atualizar(vet, posInsercao, elemento);
  }
  else{
    unsigned tam = vet->tamanho(vet);
    if(posInsercao >= tam){
      ajustarAD(vet, tam * 2);
      t->sobrecarga++;
    }

    vet->atualizar(vet, posInsercao, elemento);

              for(i=posInsercao; (i > 0)
      && COMPARAR_PRIORIDADES(vet->acessar(vet, i-1), elemento); i
          --){
        vet->atualizar(vet, i, vet->acessar(vet, i-1));
        vet->atualizar(vet, i-1, elemento);
    }

    t->movimentacoes_enfileirar += posInsercao - i;
  }

  return 1;
}

// ...
static void* _desenfileirar(TTAD* t){
  TDadoTAD *d = t->dado;
  TArrayDinamico *vet = d->vetorFila;

  if(d->posUltimo < 0) return NULL;
  void* ultimoElemento = vet->acessar(vet, d->posUltimo);
  d->posUltimo--;

  return ultimoElemento;
}
```

Por outro lado, para ordenação parcial, o algoritmo de inserção será mais eficiente pois não requer que a fila esteja completamente ordenada. O conceito adotado para a manter a ordenação parcial da fila, i.e., o elemento de maior prioridade será sempre o da primeira posição mas os demais não precisam estar ordenados, foi o de **Heap de Máxima**. O pior caso possuirá complexidade $O(\log n)$, isso significa que a fila estará mais otimizada. Uma versão da implementação (em C) desse tipo de ordenação pode ser:

```
/* === [ordenacaoParcial.c] === */

// ...
static short _enfileirar(TTAD* t, void* elemento){
  TDadoTAD *d = t->dado;
```

```
6   if(!elemento || !d) return 0;
7
8   int posInsercao = d->ocupacao, i;
9   int posAncestral = PAI(posInsercao);
10  void *elementoAncestral;
11
12  TArrayDinamico *vet = d->vetorFila;
13  unsigned tam = vet->tamanho(vet);
14
15  if(posInsercao >= tam){
16    ajustarAD(vet, tam * 2);
17    t->sobrecarga++;
18  }
19
20  vet->atualizar(vet, posInsercao, elemento);
21  d->ocupacao++;
22
23  for(i=posInsercao; (i > 0)
24    && COMPARAR_PRIORIDADES(elemento, vet->acessar(vet,
          posAncestral)); ){
25      vet->atualizar(vet, i, vet->acessar(vet, posAncestral));
26      vet->atualizar(vet, posAncestral, elemento);
27
28      t->movimentacoes_enfileirar++;
29      i = posAncestral;
30      posAncestral = PAI(i);
31  }
32
33  return 1;
34 }
35
36 // ...
37 static void* _desenfileirar(TTAD* t){
38  TDadoTAD *d = t->dado;
39  int posUltimo = d->ocupacao - 1;
40  void *raiz = NULL;
41  TArrayDinamico *vet = d->vetorFila;
42
43  if(posUltimo >= 0){
44      raiz = vet->acessar(vet, 0);
45      vet->atualizar(vet, 0, vet->acessar(vet, posUltimo));
46      vet->atualizar(vet, posUltimo, raiz);
47
48      t->movimentacoes_desenfileirar++;
49      posUltimo = (--d->ocupacao) - 1;
50      if(posUltimo > 0) ajustarHeap(t, 0, posUltimo);
51  }
52
53  return raiz;
54 }
```

Ao chamar a função `ajustarHeap` (linha 50 do código acima), assumimos que as sub-ávores `filhoEsquerdo(i)` e `filhoDireito(i)` já satisfazem a propriedade do Heap de Máxima mas o pai pode ser menor que seus filhos, então é preciso

restabelecer a ordenação. A implementação dessa função pode ser feita como mostra o código abaixo.

```
/* === [ordenacaoParcial.c] === */
// ...
void ajustarHeap(TTAD* t, int pai, int posUltimo){
  TDadoTAD *d = t->dado;
  TArrayDinamico *vet = d->vetorFila;
  void *aux;
  int esq, dir, posAtual;

  esq = 2*pai + 1;
  dir = esq + 1;
  posAtual = pai;

  if( (esq <= posUltimo)
    && COMPARAR_PRIORIDADES(vet->acessar(vet, esq), vet->acessar(
       vet, posAtual)) )
    posAtual = esq;
  if( (dir <= posUltimo)
    && COMPARAR_PRIORIDADES(vet->acessar(vet, dir), vet->acessar(
       vet, posAtual)) )
    posAtual = dir;

  if(posAtual != pai){
    aux = vet->acessar(vet, pai);
    vet->atualizar(vet, pai, vet->acessar(vet, posAtual));
    vet->atualizar(vet, posAtual, aux);

    t->movimentacoes_desenfileirar++;
    ajustarHeap(t, posAtual, posUltimo);
  }

}
```

Na figura a seguir temos a média do tempo de execução, em segundos, por instância pra cada tipo de ordenação. Foram utilizados cinco testes para cada instância e então calculada a média aritmética dos mesmos. A coluna "média" é computa o quão mais rápido é o tempo de execução da ordenação parcial em relação ao tempo da ordenação completa.

	TEMPO DE EXECUÇÃO (segundos):		
instâncias:	**ordenação completa**	**ordenação parcial**	**média:**
10	0,4490	0,0560	802%
100	0,0540	0,0680	79%
1.000	0,0700	0,0680	103%
10.000	0,6840	0,3860	177%
100.000	122,4350	0,3360	36439%
1.000.000	11486,3860	2,9790	385579%
10.000.000	-	31,6580	-

Figura 2. Relação entre as instâncias e o tempo de execução em cada tipo de ordenação.

Com isso concluímos que a ordenação parcial para solucionar o problema fornecido pode ser no mínimo 800 vezes mais rápida que a ordenação completa. E mais, para a maior instância testada ($10.000.000$) podemos estimar mais de 4 horas de processamento para enfim finalizar o programa com a ordenação completa, enquanto a parcial processa em menos de 32 segundos (em média).

A partir desta tabela (figura 2) plotamos o gráfico a seguir. Nele fica ainda mais evidente a diferença do tempo de execução entre os dois métodos analisados, o que já era esperado.

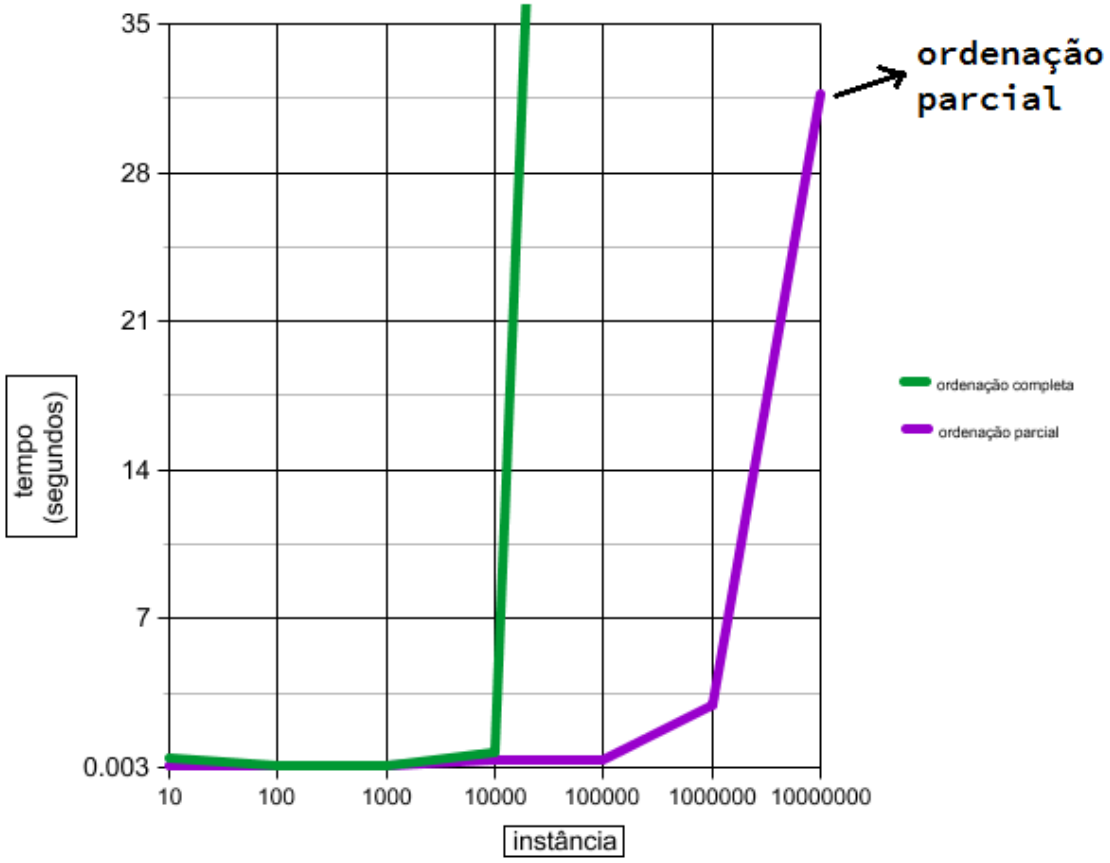

Figura 3. Gráfico do tempo de execução por instância.

1.2. A configuração experimental

Para a realização deste experimento, precisamos de antemão organizar os arquivos necessários (códigos-fonte e instâncias do problema) de acordo a hierarquia apresentada na figura abaixo (4):

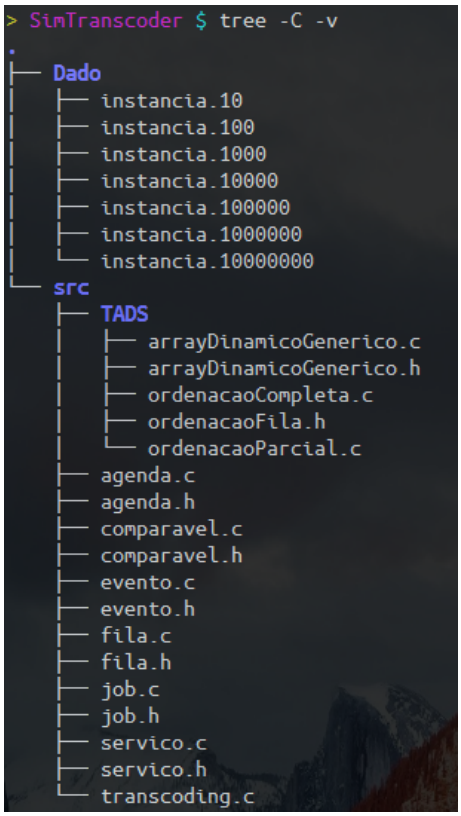

Figura 4. Árvore de diretório com os principais arquivos.

Precisamos ter essa hierarquia em mente para seguir.

1. Para compilar o programa na linha de comando, usamos:
   `$ gcc -o transcoder *.c TADS/!(X).c`
   Onde X será o nome código-fonte do tipo de ordenação que você **NÃO** quer compilar. Por exemplo, para gerar um programa cuja fila de prioridade terá ordenação completa, então $X = ordenacaoParcial$.

2. Com o executável **transcoder** gerado, podemos testar a instância 10 com o comando:
   `$ ./transcoder < ../Dado/instancia.10`

Os módulos dos dois TADs implementados em linguagem C terão a seguinte configuração de cabeçalho:

```c
/* === [ordenacaoCompleta.c] e [ordenacaoParcial.c] === */
#include "ordenacaoFila.h"
#include <stdlib.h>
```

Isso por que serão eles que implementarão as operações definidas no TAD: **Ordenação Fila** que por sua vez irá estabelecer a conexão com a implementação do TAD: **Fila de Prioridade**. Ou seja, os métodos e objetos com o mesmo nome e tipo (protótipo), exposto na figura 1 (página 1), serão conectados (ou *linkados*). Isso nos leva à seguinte relação de cabeçalhos (figura 5):

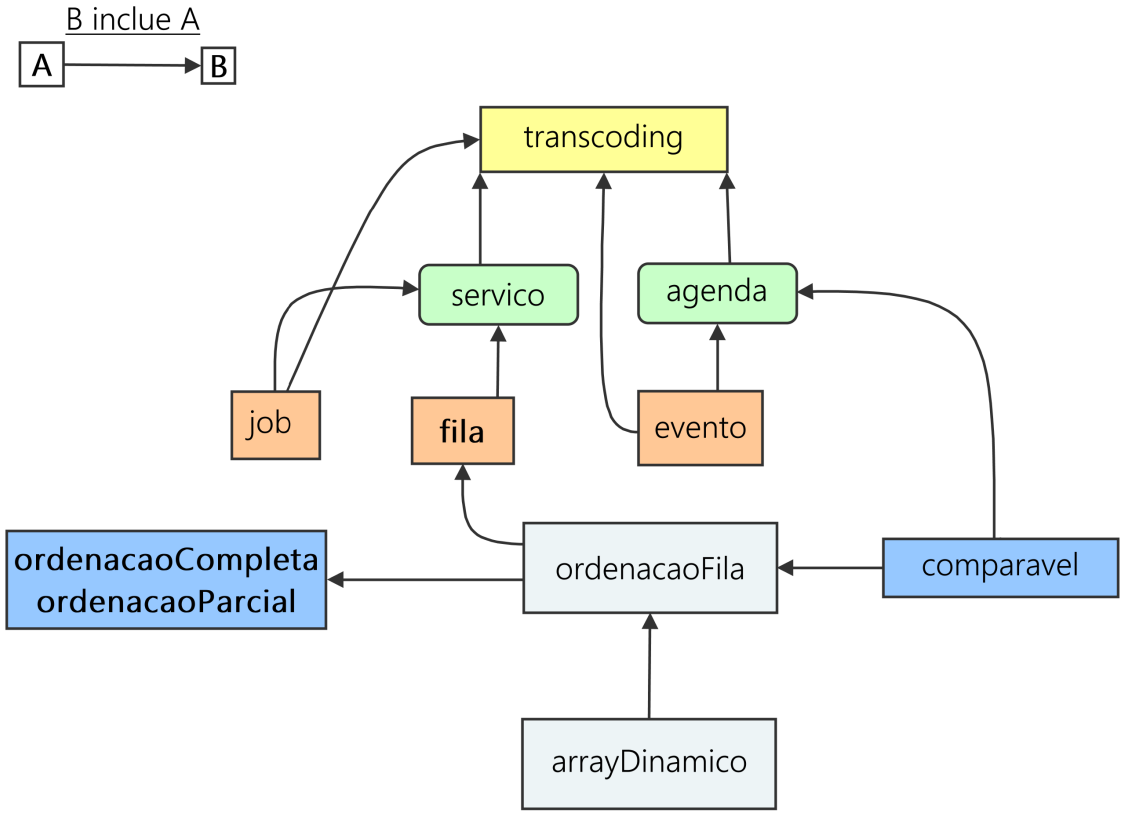

Figura 5. Conexão entre os módulos do programa.

Algumas observações a respeito da relação anterior:

- `transcoding.c` : código-fonte que contém a função principal.
- `fila.c` : código-fonte que possui a implementação do TAD: **Fila de Prioridade**.
- `ordenacaoFila.h` : cabeçalho do TAD: **Ordenação Fila**.
- `arrayDinamico.h` : cabeçalho do TAD: **Array Dinâmico**.
- `ordenacaoCompleta/Parcial.c` : faz uso do `arrayDinamico.h` que foi incluído no cabeçalho `ordenacaoFila.h`.

2. Resultados e interpretações

A seguir temos a tabela do total de movimentações realizadas em toda carga de trabalho processada por cada instância com determinado tipo de ordenação. A coluna "diferença" (na figura 6) computa a diferença entre as movimentações realizadas na ordenação parcial e as movimentações realizadas na ordenação completa. Não foi possível estimar o número de movimentações nas duas últimas instâncias testadas para a ordenação completa. Logo abaixo está o gráfico relaciona a esta tabela. Nele (figura 7) notamos a brutal diferença entre o número de movimentações. Cada movimentação contribui com o tempo de execução do programa.

	MOVIMENTAÇÕES:		
instâncias:	ordenação completa	ordenação parcial	diferença:
10	0	10	-10
100	744	300	444
1.000	192.058	7.363	184.695
10.000	19.302.127	99.764	19.202.363
100.000	1.996.492.145	1.268.375	1.995.223.770
1.000.000	-	15.245.461	-
10.000.000	-	179.474.307	-

Figura 6. Relação entre as instâncias e a quantidade de movimentações que cada tipo de ordenação realiza.

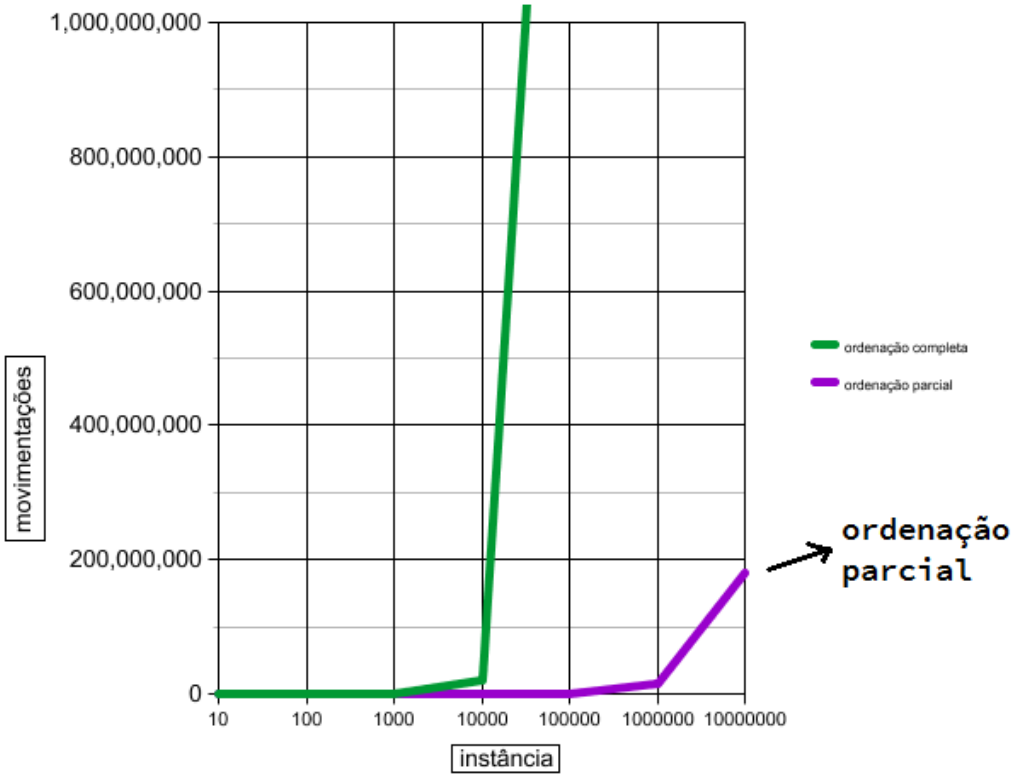

Figura 7. Gráfico da quantidade de movimentações totais realizadas por instância.

3. Conclusão

Como já era esperado para essa situação-problema, a ordenação parcial é muito mais eficiente que a completa devido à sua complexidade. E ela deve ser amplamente utilizada em qualquer tipo de fila de prioridade quem não exija a ordenação total dos seus elementos. Porém, se o objetivo for estabelecer uma ordem geral (que deve ser consultada por um sistema) de uma fila grande que demora para "andar" e a chegada de elementos é constante, então a ordenação completa utilizando o algoritmo de ordenação por inserção e o conceito de fila inversa, se sairá melhor pois o Heap deverá ordenar toda a estrutura várias vezes.