UNIVERSIDADE FEDERAL DE SANTA MARIA
ESTRUTURAS DE DADOS A
SISTEMAS DE INFORMAÇÃO

ISABELLA SAKIS
RHAUANI WEBER AITA FAZUL

RELATÓRIO DO TRABALHO SOBRE FILAS DE PRIORIDADE IMPLEMENTADA COMO HEAP BINÁRIO

SANTA MARIA, RS, BRASIL

Maio de 2016

Sumário

1. INTRODUÇÃO

O presente relatório foi escrito no intuito de relatar com detalhes o trabalho da cadeira de Estruturas de Dados, no qual foi feita à implementação de uma fila de prioridades com *heap* binário, que consiste em uma estrutura de dados onde cada elemento é inserido com uma chave que indica sua prioridade, onde a partir desta, operações como inserção e remoção serão realizadas.

Uma analogia interessante é a fila de espera em um banco, onde idosos e gestantes tem prioridade de atendimento sobre os demais. A prioridade de um elemento pode ser de qualquer tipo: valor, custo, distância, utilidade, etc. desde que seja ordenável.

Heap é uma estrutura baseada em árvores que satisfaz a propriedade das filas de prioridade. Geralmente é implementada com um *array*, porém nada impede de ser representada como uma lista encadeada (ligada). Nesse trabalho será utilizada a representação da fila prioridade como um *array*, utilizando heaps binários, ou seja, cada nó da árvore possui, no máximo, dois filhos.

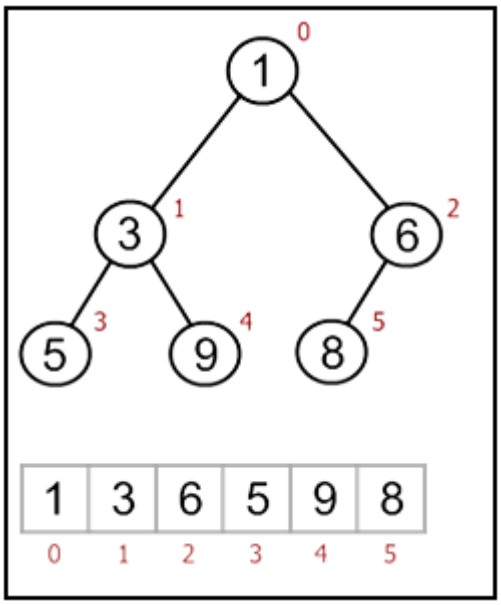

Figura 1 - Representação de um *Min heap* binário.
Fonte: *http://www.algolist.net/Data_structures/Binary_heap/Array-based_int_repr*

Na figura acima temos uma representação convencional da fila de prioridade utilizando *heap* e seguindo uma representação de árvore binária.

2. CONCEITOS

A seguir serão definidos alguns conceitos básicos relacionados à *heaps* binários representados como um vetor.

- **Funcionamento do *heap***

***Min heap* –** Cada nó é **menor** que seus filhos.
***Max heap* –** Cada nó é **maior** que seus filhos.

Todos os níveis da árvore, exceto o último, devem ser completos. Cada nó raiz, que não seja folha, tem prioridade maior ou igual à prioridade de seus filhos (sendo assim chamada *max heap*) ou prioridade menor ou igual à prioridade de seus filhos (sendo assim chamada *min heap*), ou seja, o valor de um nó é no máximo o valor de seu pai (*max heap*) ou vice-versa (*min heap*).

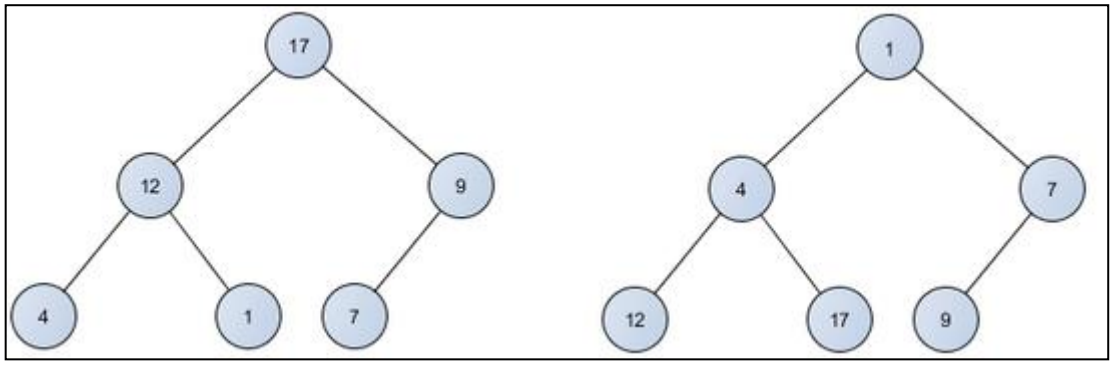

Figura 2 - Representação de uma *Max heap* à esquerda e de uma *Min heap* à direita.
Fonte: *http://www.algolist.net/*

- **Acesso aos índices**

Devido a facilidade de numeração usando *arrays* o acesso aos elementos antecessores e predecessores (pais e filhos) pode ser feito através das regras:

1. Índice do filho esquerdo do nó: **2 * índice do nó + 1**;
2. Índice do filho direito do nó: **2 * índice do nó + 2**;
3. Índice da raiz (pai) do nó: **(índice do nó – 1) / 2**.

3. METODOLOGIA

Para atingir os objetivos propostos neste trabalho, elaborou-se uma pesquisa sobre o assunto em questão, ou seja, fila de prioridades implementada com *heap* binário. Mediante aos conteúdos identificados a partir dessa pesquisa, foram conceituados os principais pontos a serem tratados. Na sequência, os algoritmos de inserção, remoção e consulta foram implementados e descritos passo a passo, assim como exemplos de sua implementação.

4. ALGORÍTIMOS

A seguir é descrito o funcionamento básico das funções de inserção, remoção e busca em filas prioridade. Como a implementação dessas funções necessitam da utilização tanto de *min heap* quanto de *max heap*, primeiramente é explicada a definição desses conceitos. O código fonte das implementações dessas funções em linguagem C se encontra no Apêndice A.

4.1 *MIN HEAP*

Para a implementação de uma *Min heap* a condição é de que a prioridade do elemento a ser inserido seja menor do que a de seus filhos, o que, por sua vez, implica que seus filhos tenham prioridade maior ou igual à de seus pais.

4.2 *MAX HEAP*

Ao contrário de um *Min heap* a condição a ser respeitada é a de que a prioridade do elemento a ser inserido seja maior do que a de seus filhos, o que implica que seus filhos tenham prioridade menor ou igual à de seus pais.

4.3 INSERÇÃO

Inserir um elemento no *heap*, comparado com uma árvore binária, requer um cuidado adicional para que as propriedades que definem o *heap* sejam obedecidas.

4.3.1 Sequência de passos do algoritmo geral de inserção

1. Inserir o elemento no final do *heap*, ou seja, na próxima posição livre do vetor;
2. Se a raiz existir, comparar o elemento com seu pai:

a) Se o elemento inserido tiver a mesma prioridade de seu pai ou respeitar as propriedades do *heap*, a inserção termina;

b) Se não, o elemento deve ser trocado com o pai e o passo anterior deve ser repetido até satisfazer as propriedades ou chegar à raiz.

3. A quantidade de elementos existentes no *heap* é incrementada.

4.4 REMOÇÃO

A remoção em um *heap* é feita a partir de sua raiz, ou seja, o elemento a ser removido será o que possui sua prioridade na primeira posição do vetor (índice 0). Após a remoção, as propriedades do *heap* devem ser mantidas.

4.4.1 Sequência de passos do algoritmo geral de remoção

1. Remover a raiz do *heap* (primeiro elemento do vetor);
2. Substituir a raiz pelo último elemento do último nível;
3. Comparar a nova raiz com seus filhos:

    a) Se o novo elemento respeitar as propriedades do *heap*, a remoção termina;

    b) Se não respeitar, trocar com um de seus filhos e retornar ao passo anterior.

4. A quantidade de elementos existentes no *heap* é decrementada.

4.5 CONSULTA

Existem diversos modos para fazer a busca de prioridade em um vetor, algoritmos mais complexos verificam se a prioridade do elemento buscado é maior ou menor da prioridade referente ao índice atual do vetor.

4.5.1 Algoritmo Geral

O algoritmo mais simples e mais utilizado consiste em uma busca simples de vetor, onde se percorre todo vetor até encontrar a prioridade requerida. Caso não seja encontrado, obviamente, o elemento não existe no *heap*.

5. EXEMPLOS

5.1 EXEMPLO DE INSERÇÃO

A Figura 3 representa um exemplo de inserção em uma fila de prioridade implementada como *heap binário*.

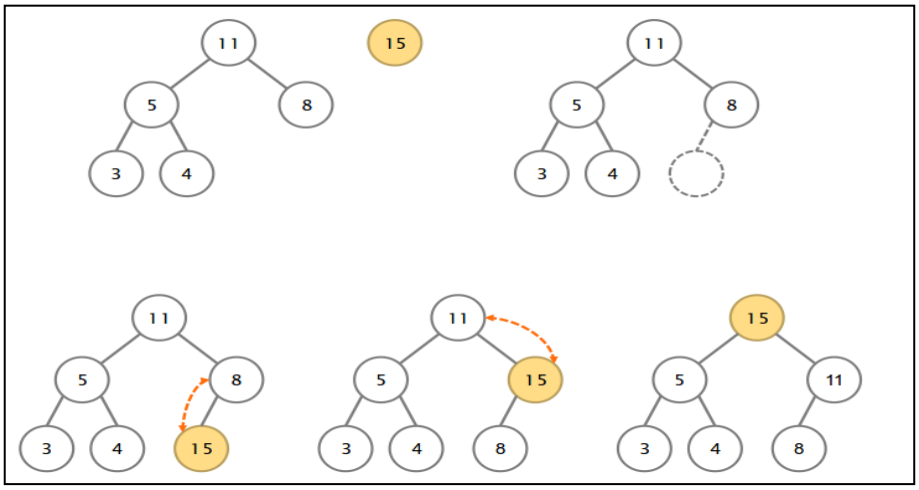

Figura 3 - Exemplo de inserção em uma fila de prioridade
Fonte: *http://webserver2.tecgraf.puc-rio.br/eda/slides/EDA_05_Heap.pdf*

Na figura 3 o elemento 15 é inserido na primeira posição livre da árvore, ou seja, última posição do vetor (passo 1 da sequência do algoritmo geral de inserção); em seguida, o valor é comparado com seu pai, como ele não satisfaz a propriedade de uma *Max heap*, pois 15 > 8 ele é trocado com seu pai (passo 2a da sequência do algoritmo geral de inserção); mais uma verificação é feita, e como 15 > 11, a troca de valores deve ser realizada novamente, a fim de satisfazer as prioridades do *heap*; por fim, como o elemento não possui raiz, a inserção termina.

5.2 EXEMPLO DE REMOÇÃO

A Figura 4 representa um exemplo de remoção em uma fila de prioridade implementada como *heap binário*.

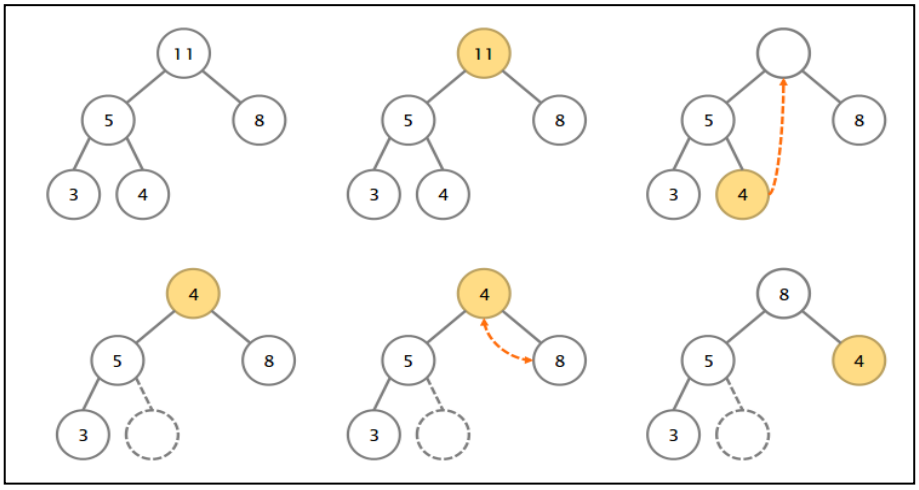

Figura 4 - Exemplo de remoção em uma fila de prioridade.
Fonte: *http://webserver2.tecgraf.puc-rio.br/eda/slides/EDA_05_Heap.pdf*

Na figura 4 o valor 4 – último elemento do vetor – é colocado no lugar em que se encontra o valor 11 – raiz do *heap*, ou seja, primeira posição do vetor - (passos 1 e 2 da sequência do algorítmo geral de remoção); em seguida a nova raiz é comparada com seus filhos, como ele não satisfaz a propriedade de uma *Max heap*, ele é trocado com seu filho (passo 3 da sequência do algorítmo geral de remoção); como não há mais filhos para comparação, a remoção termina.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDALA, Daniel D. **Filas de prioridade.** Disponível em: <http://www.facom.ufu.br/~abdala/DAS5102/TEO_HeapFilaDePrioridade.pdf>. Acesso em: 20, jun., 2016.

ALGOLIST. ***Algorithms and Data Structures, with implementations in Java and C++***. Disponível em: <http://www.algolist.net/>. Acesso em: 20, jun., 2016.

CARNEGIE, U. M. **Binary Heaps**. Disponível em: <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html>. Acesso em: 21,jun.,2016.

FEUP. **Algoritmos e Estruturas de Dados**. Disponível em: <http://paginas.fe.up.pt/~lpreis/prog2_06_07/AulasTeoricas/heap_vred.pdf >. Acesso em: 19, jun., 2016.

LOPES, J. C. **Heap.** Disponível em: <http://pt.slideshare.net/rodrigovmoraes/heap-8003774>**.** Acesso em: 21, jun., 2016.

POZZER, C. T. **Fila de Prioridade *(priority queue)*.** Disponível em: <http://www-usr.inf.ufsm.br/~pozzer/disciplinas/ed_5_FilaPrioridade.pdf>. Acesso em: 19, jun., 2016.

PUC-RIO. **Listas de prioridade e *Heaps*.** Disponível em: <http://webserver2.tecgraf.puc-rio.br/eda/slides/EDA_05_Heap.pdf>. Acesso em: 20, jun., 2016.

USFCA. **Algorithm Visualizations**. Disponível em: < https://www.cs.usfca.edu/~galles/visualization/Heap.html>. Acesso em: 19,jun.,2016.

VISUALGO. **Binary Heap**. Disponível em: <http://visualgo.net/heap>. Acesso em: 19,jun.,2019.

WALDEMAR, C.; RANGEL, J. L. **Apostila de Estrutura de Dados**. – Rio de Janeiro: Pontíficia Universidade Católica do Rio de Janeiro, 2002.

APÊNDICES

APÊNDICE A – CÓDIGOS-FONTE

- ***heap.h :***

```
#define MIN_HEAP 1
#define MAX_HEAP 2
#define RAIZ(x) (x - 1) / 2
#define FILHO_ESQ(x) (2 * x) + 1
#define FILHO_DIR(x) (2 * x) + 2

typedef struct heap {
	int* prioridades;
	int tam;   // tamanho maximo
	int pos;   // proxima pos livre
}Heap;

Heap* heap_cria(int max);
bool heap_insere(Heap* h, int prioridade, int tipo);
int heap_remove(Heap* h, int tipo);
void corrige_acima(Heap* h, int pos, int tipo);
void corrige_abaixo(Heap* h, int tipo);
void troca(int* v, int x, int y);
bool heap_busca(Heap* h, int busca);
int posicao(Heap* h, int valor);
void heap_imprime(Heap* h);
bool vazio(Heap *h);
void heap_libera(Heap* h);
```

- ***heap.c* :**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "heap.h"

Heap* heap_cria(int max) {
	Heap* h = (Heap*) malloc(sizeof(Heap));
	h->tam = max;
	h->pos = 0;
	h->prioridades = (int*) malloc(max * sizeof(int));

	return h;
}

bool heap_insere(Heap* h, int prioridade, int tipo) {
	if (h->pos < h->tam) { // tem lugar no vetor
		h->prioridades[h->pos] = prioridade;

		corrige_acima(h, h->pos, tipo);

		h->pos++;
	}
	else
		return false;
```

```
	return true;

}

int heap_remove(Heap* h, int tipo) {
	if (h->pos < 1) // nao tem elementos
		return -1;

	int topo = h->prioridades[0];
	h->prioridades[0] = h->prioridades[h->pos-1];
	h->pos--;

	corrige_abaixo(h, tipo);

	return topo;
}

void corrige_acima(Heap* h, int pos, int tipo) {
	while (pos > 0) {
		int raiz = RAIZ(pos);

		if (tipo == MIN_HEAP) {
			if (h->prioridades[pos] < h->prioridades[raiz])
				troca(h->prioridades, pos, raiz);
			else
				break;
		}
		else if (tipo == MAX_HEAP) {
			if (h->prioridades[pos] > h->prioridades[raiz])
				troca(h->prioridades, pos, raiz);
			else
				break;
		}

		pos = raiz;
	}
}

void corrige_abaixo(Heap* h, int tipo) {
	int raiz = 0;

	while (FILHO_ESQ(raiz) < h->pos) {
		int filho_esq = FILHO_ESQ(raiz);
		int filho_dir = FILHO_DIR(raiz);
		int filho;

		if (filho_dir >= h->pos)
			filho_dir = filho_esq;

		if (tipo == MIN_HEAP) {
			if (h->prioridades[filho_esq] < h->prioridades[filho_dir])
				filho = filho_esq;
			else
				filho = filho_dir;

			if (h->prioridades[raiz] > h->prioridades[filho])
				troca(h->prioridades, raiz, filho);
			else
				break;
		}
```

```
                else if (tipo == MAX_HEAP) {
                        if (h->prioridades[filho_esq] > h->prioridades[filho_dir])
                                filho = filho_esq;
                        else
                                filho = filho_dir;

                        if (h->prioridades[raiz] < h->prioridades[filho])
                                troca(h->prioridades, raiz, filho);
                        else
                                break;
                }

                raiz = filho;
        }
}

void troca(int* v, int x, int y) {
        int aux = v[x];
        v[x] = v[y];
        v[y] = aux;
}

bool heap_busca(Heap* h, int busca) {
        int i;

        for (i = 0; i < h->pos; i++)
                if (h->prioridades[i] == busca)
                        return true;

        return false;
}

int posicao(Heap* h, int valor){
        int i;

        for (i = 0; i < h->pos; i++)
                if (h->prioridades[i] == valor)
                        return i;

        return -1;
}

void heap_imprime(Heap* h) {
        if (vazio(h)) {
                printf("Heap vazio.\n");
                return;
        }

        printf("Vetor de prioridades:\n");

        int i, filho;
        for (i = 0; i < h->pos; i++) {
                printf("Raiz: %d\t", h->prioridades[i]);

                filho = FILHO_ESQ(i);
                if (filho < h->pos)
                        printf("  FilhoEsq: %d\t", h->prioridades[filho]);

                filho = FILHO_DIR(i);
                if (filho < h->pos)
```

```
                printf("  FilhoDir: %d", h->prioridades[filho]);

            printf("\n");
        }

        printf("\n");
}

void heap_libera(Heap* h) {
        free (h->prioridades);
        free (h);
}
```

- ***main.c :***

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "heap.h"

int menu(void);

int main(int argc, char **argv) {
        int tam, op, tipo, valor;
        bool ok;
        novo:

        printf("Digite o tamanho da heap: ");
        scanf("%d", &tam);

        Heap* heap = heap_cria(tam);

        if (heap != NULL)
            printf("Heap de %d elementos criada com sucesso.\n\n", tam);
        else
            exit(1);

        do {
            printf("\t1 - Min Heap\n\t2 - Max heap\n\tEscolha: ");
            scanf("%d", &tipo);

            if (tipo != 1  && tipo != 2) {
                printf("\nOpcao invalida. Escolha novamente.\n\n");
                ok = false;
            }
            else
                ok = true;

        }while (!ok);

        printf("\n");

        do {
            op = menu();
            switch (op) {
                case 1:   printf("Prioridade a ser inserida: ");
                          scanf("%d", &valor);

                          if (!heap_insere(heap, valor, tipo))
```

```
                                        printf("Insercao impossivel. Heap cheio.\n");
                        break;
                case 2: valor = heap_remove(heap, tipo);

                                if (valor == -1)
                                        printf("Remocao impossivel. Heap vazio.\n");
                                else
                                        printf("O elemento %d foi removido.\n", valor);
                        break;
                case 3: printf("Digite o elemento para consulta: ");
                                scanf("%d", &valor);

                                if (heap_busca(heap, valor))
                                        printf("Elemento %d encontrado na posicao %d do vetor de
prioridades.\n", valor, posicao(heap, valor));
                                else
                                        printf("Elemento nao encontrado.\n");
                        break;
                case 4: heap_imprime(heap);
                        break;
                case 5: heap_libera(heap);
                                goto novo;
                        break;
                case 6: printf("Fim.\n");
                        break;
                default: printf("Opcao invalida. Escolha novamente.\n");
        }
        printf("\n");

    }while (op != 6);

    heap_libera(heap);

    return 0;
}

int menu(void) {
    int op;
    printf("\t----------\n");
    printf("\tHEAP BINARIO\n\t1 - Inserir\n\t2 - Remover\n\t3 - Buscar\n");
    printf("\t4 - Imprimir\n\t5 - Novo heap\n\t6 - Sair\n\tEscolha: ");
    scanf("%d", &op);
    printf("\t----------\n\n");

    return op;
}
```