Relatório 06 - Heap Heap Hurray

SCC0220 - Laboratório de Introdução à Ciência da Computação II (2022)

Prof. Diego Furtado Silva

Departamento de Ciências da Computação (SCC)

Instituto de Ciências Matemáticas e de Computação (ICMC)

Universidade de São Paulo

Discentes:

- Felipi Yuri Santos (*11917292*)
- Vicenzo D'Arezzo Zilio (*13671790*)

Sumário

- I - Introdução
- II - Implementação
- III - Comparação dos tempos de execução

I - Introdução

Um tiktoker precisa de alguma ideia de assunto para trocar com Letícia, então ele decide comparar duas formas diferentes de implementar um algoritmo de ordenação, um dos melhores senão o melhor, porém um dos mais

trabalhosos de fazer funcionar, o heapSort().

A primeira forma de implementá-lo é de uma forma mais direta, quando temos um conjunto n de dados pré definido, colocamos todos esses dados numa árvore Heap através do método de maxHeapify() de uma vez só e precisamos ordená-lo após todos os elementos estarem inseridos na árvore binária.

A segunda forma é mais on-demand. A cada inserção de elemento na árvore Heap, fazemos o processo de BubbleUp para colocar sempre o elemento mínimo (o menor elemento) na raiz dessa árvore. A ordenação, entretanto, só ira ocorrer quando o método heapSort() for chamado, deixando-nos livres para colocar mais elementos mais tarde. Logo de cara conseguimos presumir que essa implementação on-demand será mais custosa por razões óbvias de ter muito mais operações envolvidas.

II - Implementações do HeapSort()

i. Primeira implementação: direta com maxHeapify()

```
//Verifica a descendência de um elemento, verificando e realizando as trocas necessárias.
void bubble_down(int* vetor, int pai, int tamanho){
    //encontra o filho da esquerda;
    int filho = pai*2 + 1;

    //condição de parada
    if(filho > tamanho) return;

   //verifica se será necessária a troca;
    if (vetor[filho] > vetor[pai] || (filho + 1 <= tamanho && vetor[filho+1] > vetor[pai])){

        //caso for, descobre com quem trocar;
        if (filho+1 <= tamanho && vetor[filho+1] > vetor[filho]){
            filho = filho+1;
        }

        //realiza a troca
        int aux = vetor[pai];
        vetor[pai] = vetor[filho];
        vetor[filho] = aux;

        //atualiza possíveis "netos" restantes;
        bubble_down(vetor, filho, tamanho);
    }
}

//Constroí uma heap a partir de um vetor de inteiros.
int * heapifyMax(int * entrada, int tamanho){
    int ultimoPai = tamanho/2 - 1;
    int i;
    for (i = ultimoPai; i >= 0; i--){
        bubble_down(entrada, i, tamanho);
    }
    return entrada;
}


//Recebe uma Heap e realiza uma ordenação dentro do vetor heap.
void heapSort(int* heap, int tamanho){
```

```
    //processo de ordenação
    while (tamanho > 0){//tamanho == 1 (vetor unitário)
        //maior elemento está sempre na raiz vetor[1]; colocamos o maior valor na última posição
        int maior = heap[0];
        heap[0] = heap[tamanho];
        heap[tamanho] = maior;

        tamanho--; //reduzimos o tamanho do Heap
        bubble_down(heap, 0, tamanho);//verificamos a consistência do Heap para o nó-raiz!
        //com a troca, o maior elemento não está na raiz!
    }

}
```

ii. Segunda implementação: on-demand

```
//cria o resgistro da heap e aloca seu vetor interno.
heap_t *heap_criar(int n){
    heap_t *h = (heap_t *) malloc(sizeof(heap_t));
    assert(h);
    h->vetor = (int *) calloc(n, sizeof(int));
    assert(h);
    h->capacidade = n;
    h->tamanho_heap = 0;
    return h;
}

//Função utilizada pelo bubbleUp, retorna o pai do índice k.
int consTutelar(int k){
    if (k % 2 == 0)
        return k / 2 - 1;
    else
        return (k - 1) / 2;
}


//Analisa a antecedência de um elemento, verificando e realizando as trocas necessárias.
void bubbleUp(int id, int *vetor){
    int pai = consTutelar(id);
    int aux;

    //condição de parada:
    if(pai < 0) return;

    //caso pai menor que filho, troca os elementos e chama a recursão para o pai.
    if(vetor[pai] < vetor[id]){
        aux = vetor[pai];
        vetor[pai] = vetor[id];
        vetor[id] = aux;
        bubbleUp(pai, vetor);
    }
}


//Insere um elemento no vetor da heap, mantendo sua lei.
bool heap_inserir(int key, heap_t * heap){
```

```
    //verifica se esta cheia
    if(heap->capacidade < heap->tamanho_heap)
        return false;

    int indice = heap->tamanho_heap;
    heap->tamanho_heap += 1;
    heap->vetor[indice] = key;

    //percorre a altura da árvore realizando as trocas caso necessário
    bubbleUp(indice, heap->vetor);
    return true;
}

//apaga o registro heap e seu vetor interno.
void heap_apagar (heap_t ** h){
    assert(*h);
    free((*h)->vetor);
    free(*h);
    *h = NULL;
}
```

III - Comparação dos tempos de execução

HeapSort com:	Caso 1 (ms)	Caso 2 (ms)	Caso 3 (ms)	Caso 4 (ms)	Caso 5 (ms)
maxHeapify()	1	2	12	10	20
bubbleUp()	1	3	23	32	20

Com essa comparação fica evidente que a versão com maxHeapify() seria mais eficiente pela quantidade menor de comparações, porém a versão on-demand é bem mais flexível e permite maior dinamicidade de trabalho, como supomos na introdução deste relatório.