ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES (EACH-USP)
Introdução à Análise de Algoritmos (ACH2002) – Prof. Karina Valdivia Delgado
Samira Silva Heringer (11876846), Sungwon Yoon (9822261)

Relatório: Análise dos códigos do Trabalho 3

1. Análise de Heap-Increase-Key

1.1 Pseudocódigo

```
HEAP-INCREASE-KEY(A, i)
1. enquanto (i > 1) e (A[INDEX-PARENT(A, i)]->prior < A[i]->prior):
2.          SWAP(&A[i], &A[INDEX-PARENT(A, i)])
3.          i = INDEX-PARENT(A, i)
```

```
INDEX-PARENT(i)
1. return i/2
```

```
SWAP(**a, **b)
1. *temp = *a
2. *a = *b
3. *b = *a
```

1.2 Descrição do algoritmo

A função **HEAP-INCREASE-KEY** ordena o novo elemento na árvore binária estática de acordo com as prioridades. São passados como parâmetros *int i* e *ELEMENTOS* A[]*. O vetor A é um vetor que guarda os ponteiros dos elementos e *i,* índice a ser ordenado. Como novo elemento sempre é inserido depois do último elemento do vetor, podemos dizer que *i* equivale à quantidade total *n* de elementos existentes nele.

A função **INDEX-PARENT** calcula o índice do pai. Uma vez que índices inválidos – i > 4001 ou i <= 1 – são filtrados previamente, apenas retorna o valor de i / 2.

A função **SWAP**, troca o conteúdo apontado por dois parâmetros passados pela referência, com uso de uma variável auxiliar.

1.3 Cálculo do consumo de tempo

Para cálculo de tempo, serão levados em consideração as operações de atribuição, operações de comparação e operações aritméticas.

O melhor caso ocorre quando a prioridade é mínima, ou seja, quando não há necessidade de movimentar a posição do elemento no vetor A. Assim, é executada apenas

uma vez a linha 1 – comparação i > 1; operação i/2 de INDEX-PARENT; e comparação de prioridades de dois elementos. Portanto, o consumo de tempo no melhor caso é T(n) = 3.

O pior caso ocorre quando a prioridade do novo elemento é máxima, ou seja, quando o elemento deve ser movimentado para a primeira posição do vetor. Nesse caso, o índice *i* começa de *n* e decresce até 1, sendo atualizado i/2 em i/2. Assim, as quatro atribuições da linha 2 e 3, o cálculo do índice do pai e uma comparação entre prioridades são executadas lg(n) vezes. A primeira comparação da linha 1 (i > 1) é executada lg(n) + 1 vezes, funcionando como condição de parada do laço. Portanto, o consumo de tempo no pior caso é T(n) = 7*lg(n) + 1.

Assim, podemos concluir que a complexidade de tempo do algoritmo HEAP-INCREASE-KEY é O(lg n).

2. Análise de Heap-Insert

2.1 Pseudocódigo

```
HEAP-INSERT(A, i, *novo)
1. A[i] = novo
2. HEAP-INCREASE-KEY(A, i)
```

2.2 Descrição do algoritmo

A função HEAP-INSERT recebe como parâmetro o endereço do novo elemento criado em uma outra função e chama a função HEAP-INCREASE-KEY, para posicionar corretamente esse elemento, como já descrito anteriormente.

2.3 Cálculo do consumo de tempo

Como a linha 1 é constante e já calculamos que a linha 2 tem tempo T(n) = 7*lg(n)+1 no pior caso, podemos concluir, portanto, que o consumo de tempo da função HEAP-INSERT é T(n) = 7*lg(n)+2 e a complexidade de tempo é O(lg n), no pior caso.

3. Principais dificuldades

Não encontrei dificuldades na implementação. Como a grande parte do código podia ser aproveitado do link disponibilizado, apenas comparações entre elementos foi mudado para comparações entre prioridades e algumas adaptações para struct foram feitas. Gastei maior parte do tempo corrigindo o código para ter saída do exemplo 1 igual ao do pdf, mas perguntando para monitora, fui informada de que era o exemplo 1 que estava errado.