6.1 Filas de prioridades

Nesta seção veremos uma das principais aplicações da estrutura de *heaps*, as *filas de prioridades*. Uma fila de prioridades é uma estrutura de dados onde cada elemento possui um valor que é a sua prioridade. Assim como os *heaps*, as filas de prioridades podem ser de máximo ou de mínimo. Uma fila de prioridades de máximo de um conjunto `S` possui as seguintes operações:

- $insert(S, x)$: insere o elemento x no conjunto S, o que equivale à operação $S = S \cup \{x\}$.
- $maximum(S)$: retorna o elemento de S que possui a maior prioridade (key).
- $extract\text{-}max(S)$: remove e retorna o elemento de S que possui a maior prioridade (key).
- $increase\text{-}key(S, x, k)$: incrementa o valor da prioridade (key) de x para o novo valor k, onde assumimos que $x \leq k$.

Alternativamente, uma fila de prioridades de mínimo possui as operações *insert*, *minimum*, *extract-min* e *decrease-key*.

Um *heap* pode ser usado para implementar uma fila de prioridades. Vejamos como implementar as operações de uma fila de prioridades de máximo. Se A é um *heap* de máximo, então $maximum(A)$ pode ser implementado em tempo constante, isto é, $\Theta(1)$:

Algorithm 8: maximum(A)

```
1 return A[1];
```

A operação $extract\text{-}max(A)$ é similar ao procedimento $Heapsort$:

Algorithm 9: extract-max(A)

```
1 if A.heap-size < 1 then
2 |   error "heap underflow";
3 end
4 max = A[1];
5 A[1] = A[A.heap-size];
6 A.heap-size = A.heap-size − 1;
7 Max-Heapify(A,1);
8 return max;
```

A operação *extract-max* realiza um número de operações constantes seguidas de um Max-Heapify que tem complexidade $O(\lg\ n)$. Logo, *extract-max* também tem complexidade $O(\lg\ n)$.

A operação *increase-key* é implementada como a seguir:

Algorithm 10: increase-key(A, i, key)

```
1 if key < A[i] then
2 |   error "new key is smaller than current key";
3 end
4 A[i] = key;
5 while i > 1 and A[Parent(i)] < A[i] do
6 |   exchange A[i] with A[Parent(i)];
7 |   i = Parent(i);
8 end
```

Exercício 6.1.1. *Prove a seguinte invariante de* loop *para a operação increase-key:*

> *Antes de cada iteração do laço* ***while*** *(linhas 5-8), o subvetor* $A[1..A.\textit{heap-size}]$ *satisfaz a propriedade de* heap *de máximo, exceto por uma possível violação:* $A[i]$ *pode ser maior do que* $A[Parent(i)]$.

O tempo de execução de increase-key em um *heap* com n elementos é $O(\lg\ n)$ porque o comprimento do caminho que vai da posição do elemento que terá sua prioridade alterada até a raiz do *heap* tem comprimento limitado por $(\lg\ n)$.

Por fim, a operação *insert* é dada como a seguir, e claramente tem complexidade $O(\lg\ n)$:

Algorithm 11: insert(A, key)

```
1 A.heap-size = A.heap-size + 1;
2 A[A.heap-size] = −∞;
3 increase-key(A, A.heap-size, key);
```