Ficha 4

Algoritmos e Complexidade

Estruturas lineares e Análise agregada

1 *Buffers*

Considere o seguinte *header file* para **buffers** de inteiros.

```
typedef struct buffer *Buffer;

Buffer init (void); // inicia e aloca espaço
int empty (Buffer); // testa se está vazio
int add (Buffer,int); // acrescenta elemento
int next (Buffer, *int); // proximo a sair
int remove (Buffer, *int); // remove proximo
```

1. Uma instanciação deste conceito de *buffer* são *stacks*. Neste caso, o próximo elemento a sair é o último que foi acrescentado (*Last In First Out*).

   Apresente duas implementações de *Stacks* em que todas as operações executem em tempo constante (i.e., independente do número de elementos que estão na *stack*).

   (a) Uma implementação baseada em listas ligadas

   (b) Uma implementação baseada em *arrays*. Assuma neste caso que existe definida uma constante `MaxS` que corresponde ao tamanho máximo da *stack*. Alternativamente, poderá ser passado um parâmetro extra na função de inicialização e que corresponde ao tamanho do vector a ser alocado na inicialização.

2. Uma outra instanciação do conceito de *buffer* são *queues*. Neste caso, o próximo elemento a sair é o primeiro que foi acrescentado (*First In First Out*).

   Apresente duas implementações de *Queues* em que todas as operações executem em tempo constante (i.e., independente do número de elementos que estão na *queue*).

   (a) Uma implementação baseada em listas ligadas

   (b) Uma implementação baseada em *arrays*. Assuma neste caso que existe definida uma constante `MaxQ` que corresponde ao tamanho máximo da *queue*. Alternativamente, poderá ser passado um parâmetro extra na função de inicialização e que corresponde ao tamanho do vector a ser alocado na inicialização.

3. Considere agora uma instanciação de *buffer* em que o próximo elemento a sair é o menor elemento que se encontra no *buffer*. E para este caso vamos considerar 3 implementações possíveis:

A os elementos do *buffer* são armazenados sequencialmente por ordem crescente;

B os elementos do *buffer* são armazenados por ordem de chegada;

Para cada uma das implementações sugeridas

(a) analise (informalmente) a complexidade das funções de inserção e remoção no melhor e pior casos (identifique esses casos).

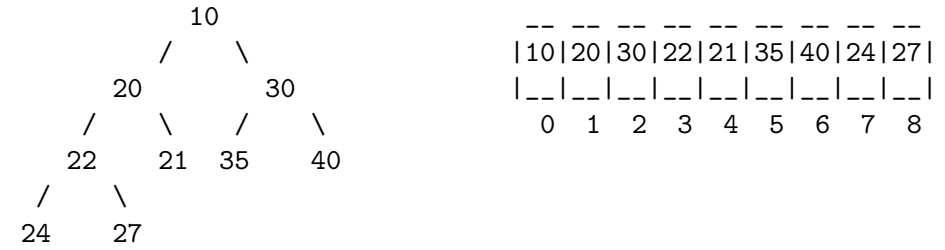
(b) Considere agora uma sequência de `N` instruções de inserção e remoção que, partindo do *buffer* vazio acabam com o *buffer* também vazio (e por isso mesmo a sequência tem de ter tantas remoções como inserções). Identifique a melhor e a pior destas sequências e calcule, para cada uma destas, o custo da sequência.

4. Uma *(min)-heap* é uma árvore binária que verifica duas propriedades:

   - *shape property*: a árvore é completa, ou quasi-completa.
   - *(min)-heap property*: o conteúdo de cada nó é menor ou igual que o conteúdo dos seus descendentes (não havendo, no entanto, qualquer relação de ordem entre os conteúdos das duas sub-árvores de um mesmo nó).

   As heaps têm assim uma implementação muito vantajosa em array, em que a árvore vai sendo disposta por níveis ao longo do array (da esquerda para a direita). O acesso ao nó pai e aos nós filhos é feito de forma directa por aritmética de índices.

   Exemplo de uma *min-heap* e sua representação em array:



   Considere o seguinte *header file* para **min-heaps** de inteiros.

```
#define PARENT(i)  (i-1)/2     // o indice do array começa em 0
#define LEFT(i)    2*i + 1
#define RIGHT(i)   2*i + 2

typedef int Elem;  // elementos da heap.

typedef struct {
 int   size;
 int   used;
 Elem  *heap;
} Heap;

int newHeap (int size);          // Cria uma nova Heap vazia com a capacidade size
int insertHeap(Heap *h, Elem x); // Insere um elemento na heap
void bubbleUp(Heap *h, int i);   // Função auxiliar de inserção: dada uma posição i
```

```
                                // da heap com um novo valor que possivelmente viola
                                // a propriedade da heap, propaga esse gradualmente
                                // para cima.
int extractMin(Heap *h, Elem *x);// Retira o mínimo da heap
void bubbleDown(Heap *h, int i); // Função auxiliar de remoção: dada uma posição i
                                // da heap com um valor que possivelmente viola
                                // a propriedade da heap, propaga esse gradualmente
                                // para baixo.
```

Apresente uma implementação das operações indicadas e analise o seu tempo de execução.

2 Tabelas de Hashing

1. Para implementar tabelas de hash usando o método de *open addressing* considere as seguintes declarações:

```
#define HASHSIZE    31      // número primo
#define EMPTY       ""
#define DELETED     "-"

typedef char KeyType[9];
typedef void *Info;

typedef struct entry {
  KeyType  key;
  Info info;
} Entry;

typedef Entry HashTable[HASHSIZE];
```

   (a) Implemente as seguintes funções

```
int Hash(KeyType);              // função de hash
void InitializeTable(HashTable); // inicializa a tabela de hash
void ClearTable(HashTable);     // limpa a tabela de hash
```

   (b) Use o método *linear probing* na implementação das seguintes funções.

```
// insere uma nova associação entre uma chave nova e a restante informação
void InsertTable_LP(HashTable, KeyType, Info);

// apaga o elemento de chave k da tabela
void DeleteTable_LP(HashTable, KeyType);

// procura na tabela o elemento de chave k, e retorna o índice da tabela
// aonde a chave se encontra (ou -1 caso k não exista)
int  RetrieveTable_LP(HashTable, KeyType);
```

   (c) Use agora o método *quadratic probing* na implementação das seguintes funções.

```
// função de hash
int Hash_QP(KeyType, int);
```

```
// insere uma nova associação entre uma chave nova e a restante informação
void InsertTable_QP(HashTable, KeyType, Info);

// apaga o elemento de chave k da tabela
void DeleteTable_QP(HashTable, KeyType);

// procura na tabela o elemento de chave k, e retorna o índice da tabela
// aonde a chave se encontra (ou -1 caso k não exista)
int  RetrieveTable_QP(HashTable, KeyType);
```

(d) Efectue a análise assimptótica do tempo de execução das funções que implementou.

2. Considere uma tabela de Hash implementada sobre um array de tamanho 7 para armazenar números inteiros. A função de hash utilizada é $h(x) = x\%7$ (em que % representa o resto da divisão inteira). O mecanismo de resolução de colisões utilizado é *open addressing* com *linear probing*.

   - Apresente a evolução desta estrutura de dados quando são inseridos os valores 1, 15, 14, 3, 9, 5 e 27, por esta ordem.
   - Descreva o processo de remoção de um elemento ensta estrutura de dados, exemplificando com a remoção do valor 1 depois das primeiras 4 inserções acima.

3. Para implementar tabelas de hash usando o método de *chaining* considere as seguintes declarações:

```
#define HASHSIZE    31    // número primo

typedef char KeyType[9];
typedef void *Info;

typedef struct entry {
  KeyType  key;
  Info info;
  struct entry *next;
} Entry;

typedef Entry *HashTable[HASHSIZE];
```

(a) Apresente uma implementação para as seguintes funções.

```
// função de hash
int Hash(KeyType);

// inicializa a tabela de hash
void InitializeTable(HashTable);

// limpa a tabela de hash
void ClearTable(HashTable);

// insere uma nova associação entre uma chave nova e a restante informação
void InsertTable(HashTable, KeyType, Info);

// apaga o elemento de chave k da tabela
```

```
void DeleteTable(HashTable, KeyType);

// procura na tabela o elemento de chave k, e retorna o apontador
// para a célula aonde a chave se encontra (ou NULL caso k não exista)
Entry *RetrieveTable(HashTable, KeyType);
```

(b) Efectue a análise assimptótica do tempo de execução das funções que implementou.

4. Pretende-se agora que faça a implementação de tabelas de hash dinâmicas cujo tamanho do array alocado vai depender do *factor de carga (nº de entradas / tamanho da tabela)*

   (a) Adapte as declarações das estruturas de dados para este fim.

   (b) Adapte as funções que definiu nas alíneas anteriores a esta nova implementação. Note que nas funções de inserção e de remoção

   - quando o factor de carga é superior ou igual a 75% (50% no caso usar o método *quadratic probing*) o tamanho da tabela é aumentado para o dobro;
   - quanto o factor de carga é menor ou igual a 25% o tamanho da tabela é reduzido a metade.

3 Análise Amortizada

1. Considere o seguinte algoritmo de incremento de um inteiro armazenado num array de bits.

```
void inc (int b[], int N) {
int i;

i = N-1;
while ((i >= 0) && (b[i] == 1)){
    b[i] = 0;
    i++;
   }
if (i>=0) b[i] = 1;
}
```

   Use (os três métodos de) análise amortizada para mostrar que o tempo médio de execução desta função é constante.

2. Relembre a definição de *heap* na seguinte definição de heaps dinâmicas. Os campos `size` e `used` serão usados para guardar o número máximo de elementos que a heap pode armazenar e o número de elementos que a heap tem armazenados.

```
typedef struct {
 int   size;
 int   used;
 int  *heap;
} Heap;
```

Considere que se encontram definidas as funções `bubbleUp` e `bubbleDown` de manuseamento de heaps que executam em tempo $O(\log(N))$ para uma heap de tamanho N.

A inserção numa heap vai então ser feita da seguinte forma:

- Se a heap não estiver cheia (i.e., `used` menor do que `size`) o novo elemento é acrescentado no fim e será feito `bubbleUp` para o colocar no local apropriado.
- Se a heap estiver cheia, é primeiro alocado espaço para uma heap com o dobro da capacidade, copiado o conteúdo da heap anterior para a nova, libertado o espaço da heap anterior e efectuado o procedimento anterior.

A remoção de um elemento (o menor) de uma heap será feita da seguinte forma:

- Começa-se por remover o elemento da posição 0, colocando lá o valor do último, e aplicando a função `bubbleDown`.
- Se após esta a heap tiver 75% da sua capacidade não ocupada, é alocado espaço para uma heap com metade da capacidade, copiado o conteúdo da heap anterior para a nova e libertado o espaço desta última.

(a) Apresente uma implementação das operações `insertHeap` e `extractMin`

```
Heap insertHeap(Heap *h, Int x);   // Insere um elemento na heap
Heap extractMin(Heap *h, Elem *x);  // Retira o mínimo da heap
```

(b) Mostre que para uma sequência de N inserções ou remoções, o custo amortizado de cada uma destas operações permanece igual a $O(\log(N))$

3. Uma definição alternativa de *Stacks* às clássicas representações em *array* ou em lista ligada, consiste no uso de arrays dinâmicos. Esta solução tem a simplicidade da implementação em *array*, aliada às vantagens de usar uma estrutura dinâmica.

```
typedef struct {
    int size, used;
    int *table;
} DynTable;
```

As operações de adição e remoção de um elemento (por exemplo nas *stacks*, as operações de *push* e *pop*) deverão testar a capacidade usada da tabela e, em certos casos, realocar os elementos da tabela:

- ao acrescentar um elemento a uma tabela cheia (`size == used`) deve-se começar por realocar os elementos da tabela para uma com o dobro da capacidade.
- Ao remover um elemento de uma tabela, se ela passar a estar apenas a 25% da sua capacidade, devem-se realocar os elementos para uma tabela com metade da capacidade.

Usando análise amortizada, mostre que esta solução tem custo amortizado constante das operações de inserção e remoção.

4. Uma implementação possível de uma fila de espera (*Queue*) utiliza duas *stacks* A e B, por exemplo:

```
typedef struct queue {
 Stack a;
 Stack b;
} Queue;
```

   - A inserção (`enqueue`) de elementos é sempre realizada na *stack* A;
   - para a remoção (`dequeue`), se a *stack* B não estiver vazia, é efectuado um *pop* nessa *stack*; caso contrário, para todos os elementos de A excepto o último, faz-se sucessivamente *pop* e *push* na *stack* B. Faz-se depois *pop* do último, que é devolvido como resultado.

   (a) Efectue a análise do tempo de execução no melhor e no pior caso das funções `enqueue` e `dequeue`, assumindo que todas as operações das *stacks* são realizadas em tempo constante.

   (b) Mostre que o custo amortizado de cada uma das operações de `enqueue` ou `dequeue` numa sequência de N operações é $\mathcal{O}(1)$. Faça isto das seguintes formas.

      i. Defina a sequência de N operações que, partindo de uma queue vazia, tem o maior custo. Calcule o custo médio de cada operação nessa sequência (análise agregada).

      ii. Apresente estimativas para o custo amortizado de cada operação de forma que para qualquer sequência de operações o somatório dos custos amortizados seja maior do que o custo real dessa sequência de operações (método contabilistico).

      iii. Defina uma função de potencial que permita concluir que o custo amortizado de cada operação é $\mathcal{O}(1)$. Baseado nesse potencial defina o custo amortizado de cada uma das operações de inserção e remoção de um elemento na *queue* (método do potencial).

5. Uma `quack` é uma estrutura que combina as funcionalidades de uma `queue` com as de uma `stack`. Pode ser vista como uma lista de elementos em que são possíveis três operações:

   - `push` que adiciona um elemento;
   - `pop` que remove o último elemento inserido;
   - `pull` que remove o elemento inserido há mais tempo.

   Apresente uma implementação de *quack*s usando 3 `stacks` garantindo que o custo amortizado de cada uma das três operações é $O(1)$, assumindo que todas as operações das `stacks` são realizadas em tempo constante.

   Justifique a sua implementação usando uma das 3 formas estudadas de análise amortizada.