

Ficha 3

Algoritmos e Complexidade

Estruturas de Dados

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 esta 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 2 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

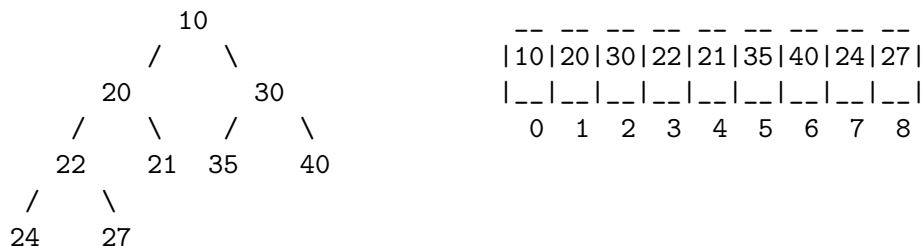
- analise (informalmente) a complexidade das funções de inserção e remoção no melhor e pior casos (identifique esses casos).
- Considere agora uma sequência de N instruções de inserção e remoção que, partindo do *buffer* vazio acabam com o *buffer* também vazio (e por isso mesmo a sequência tem de ter 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 índice 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  *values;
} Heap;

Heap *newHeap (int size);
int  insertHeap(Heap *h, Elem x);
void bubbleUp  (Elem h[], int i);
int  extractMin(Heap *h, Elem *x);
void bubbleDown(Elem h[], int N);

```

- (a) Suponha que tem uma *min-heap* com tamanho (**size**) 100, com 10 elementos (i.e., **used** tem o valor 10) e que as 10 primeiras posições do vector **values** têm os valores 4, 10, 21, 45, 13, 25, 22, 60, 100, 20. Diga qual o conteúdo desse vector após a inserção do número 6. Justifique a sua resposta desenhando as árvores correspondentes à *min-heap* antes e depois da referida inserção.
- (b) Defina uma função `int minHeapOK (Heap h)` que testa se a *min-heap* está correctamente construída (i.e., se todos os caminhos da raiz até uma folha são sequências crescentes). Certifique-se que a solução que apresentou tem um custo linear no tamanho da input.
- (c) Apresente uma implementação das operações indicadas e analise o seu tempo de execução.
- `newHeap` cria uma nova heap vazia com uma dada capacidade.
 - `insertHeap` insere um elemento na heap.
 - `bubbleUp` função auxiliar de inserção: dada uma posição da heap com um novo valor que possivelmente viola a propriedade da heap, propaga esse valor gradualmente para cima.
 - `extractMin` retira o mínimo da heap.
 - `bubbleDown` função auxiliar de remoção que dado o array em que uma *min-heap* está armazenada e em que a raiz (índice 0) possivelmente viola a propriedade da heap, propaga esse valor gradualmente para baixo.

2 Tabelas de Hash

- Considere uma tabela de *hash* (para implementar um conjunto de inteiros) com tratamento de colisões por *open addressing* (com *linear probing*) e em que a remoção de chaves é feita usando uma marca (de apagado).

Suponha que existem definidas as funções `add (int k)`, `remove (int k)`, e `exists (int k)`, sendo que esta última devolve verdadeiro/falso.

Suponha ainda que o tamanho da tabela é 7 e que a função de *hash* é $\text{hash}(x) = x \% 7$. Apresente a evolução da tabela quando, a partir de uma tabela inicialmente vazia, se executa a seguinte sequência de operações:

add 15; add 25; add 9; add 1; rem 9; add 38; rem 15; add 6; add 10; add 19;

- Considere uma tabela de *hash*, com *open addressing* e tratamento de colisões por *linear probing*. Considere ainda que, para implementar remoções, cada célula da tabela tem uma *flag* que pode tomar três valores possíveis: L, O ou A (Livre, Ocupada ou Apagada). Assumindo que as chaves são inteiros e que a função de *hash* usada é $\text{hash}(n) = n \% N$, considere o seguinte estado da tabela (para $N=11$).

0	1	2	3	4	5	6	7	8	9	10											
12	L	78	O	34	L	45	L	15	O	37	A	28	O	73	O	95	O	49	O	98	L

- (a) Partindo do princípio que a função `procura`, recebe o valor de uma chave e retorna a posição da tabela onde essa chave se encontra (e -1 caso a chave não se encontre na tabela), qual o resultado de fazer, no estado apresentado, `procura(12)` e `procura(37)`. Para cada um dos casos, indique quais as posições da tabela que são consultadas.
- (b) Se a próxima inserção for da chave 60, em que posição da tabela será armazenada? Justifique indicando quais as posições da tabela que são consultadas.
3. 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 nesta estrutura de dados, exemplificando com a remoção do valor 1 depois das primeiras 4 inserções acima.
4. 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 assintótica do tempo de execução das funções que implementou.

5. 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 assintótica do tempo de execução das funções que implementou.

6. 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;
 - quando o factor de carga é menor ou igual a 25% o tamanho da tabela é reduzido a metade.

3 Árvores binárias de procura

1. Considere a seguinte definição de um tipo para representar árvores binárias de procura (BST).

```
typedef struct btree {
    int value;
    struct btree *left, *right;
} Node, *BTree;
```

Apresente definições em C para resolver cada um dos problemas abaixo. Para cada caso apresente ainda relações de recorrência que traduzam o comportamento das funções em causa para dois casos extremos: (1) a árvore está perfeitamente desequilibrada (i.e., o número de nodos da árvore é igual à altura da árvore) e (2) a árvore está equilibrada (i.e., a altura da árvore corresponde ao logaritmo do número de nodos)

- (a) A função `int size (BTree)` calcula o número de nodos de uma BST.
 - (b) A função `int altura (BTree)` calcula a altura de uma árvore.
 - (c) A função `BTree add (Btree, int)` adiciona um elemento a uma árvore.
 - (d) A função `int search (BTree, int)` determina se um inteiro ocorre numa dada árvore.
 - (e) A função `int max (BTree)` determina o maior elemento de uma árvore (não vazia).
2. Uma árvore binária diz-se balanceada se a diferença de alturas entre as suas sub-árvores não for superior a 1. A seguinte função determina se uma árvore binária está ou não balanceada.

```
int balanceada (BTree a) {
    if (a == NULL) return 1;
    l = altura (a->left);
    r = altura (a->right);
    return ((abs (l-r) <= 1)      &&
            balanceada (a->left) &&
            balanceada (a->right));
}
```

- (a) Identifique o melhor e pior caso do tempo de execução desta função.
- (b) Para o pior caso apresente uma relação de recorrência que traduza o tempo de execução desta função em função do tamanho da árvore. Apresente uma solução dessa recorrência.
- (c) Uma alternativa para melhorar o comportamento da função acima consiste em usar uma função auxiliar que, além de determinar se uma dada árvore está balanceada, calcula a sua altura.

```
int balanceada (BTree a) {
    int p;
    return (balanceadaAux (a, &p));
}

int balanceadaAux (BTree a, int *p) {
    int l, r;
    ...
}
```

Complete a definição acima de forma a garantir que a função execute em tempo linear ao tamanho da árvore. Justifique a sua solução apresentando a recorrência que traduz o comportamento da função.

3. As árvores AVL (propostas por *G.M. Adelson-Velskii* e *E.M. Landis*) são uma variante das árvores binárias de procura em que em cada nodo se guarda a diferença de alturas entre as sub-árvores.

```
typedef struct avlTree {
    int value;
    int bal;
    struct avlTree *left, *right;
} AVLNode, *AVLTree;
```

Defina uma função `int alturaAVL (AVLTree)` que calcule a altura de uma árvore em tempo proporcional à altura da árvore (i.e., em tempo logaritmico relativamente ao tamanho da árvore no caso de uma árvore balanceada).

4. Defina as funções `AVLTree rotateLeft (AVLTree a)` e `AVLTree rotateRight (AVLTree a)` que fazem rotação simples à esquerda e esquerda na raiz de uma destas árvores.
5. Represente graficamente a evolução de uma árvore AVL quando é efectuada a seguinte sequência de inserções: 10, 20, 30, 70, 40, 50. Não se esqueça de indicar os factores de balanceamento de cada nó.
6. Suponha que, numa árvore AVL inicialmente vazia, foram inseridas as chaves 10, 15, 8, 9 e 20 (por esta ordem). Note que nenhuma das inserções provocou o desbalanceamento da árvore. Apresente uma sequência de inserções balanceadas na dita árvore que façam com que a chave que fica na raiz da árvore passe a ser 20.