Ficha 3

Programação Imperativa

1 Definição de Tipos

1. Considere o seguinte tipo para representar *stacks* de números inteiros.

```
#define MAX 100
typedef struct stack {
    int sp;
    int valores [MAX];
} STACK;
```

   Defina as seguintes funções sobre este tipo:

   (a) `void initStack (STACK *s)` que inicializa uma stack (passa a representar uma stack vazia)

   (b) `int isEmptyS (STACK *s)` que testa se uma stack é vazia

   (c) `int push (STACK *s, int x)` que acrescenta `x` ao topo de `s`; a função deve retornar `0` se a operação fôr feita com sucesso (i.e., se a stack ainda não estiver cheia) e `1` se a operação não fôr possível (i.e., se a stack estiver cheia).

   (d) `int pop (STACK *s, int *x)` que remove de uma stack o elemento que está no topo. A função deverá colocar no endereço `x` o elemento removido. A função deverá retornar `0` se a operação for possível (i.e. a stack não está vazia) e `1` em caso de erro (stack vazia).

   (e) `int top (STACK *s, int *x)` que coloca no endereço `x` o elemento que está no topo da stack (sem modificar a stack). A função deverá retornar `0` se a operação for possível (i.e. a stack não está vazia) e `1` em caso de erro (stack vazia).

2. Considere o seguinte tipo para representar *queues* de números inteiros.

```
#define MAX 100
typedef struct queue {
    int inicio, tamanho;
    int valores [MAX];
} QUEUE;
```

   Defina as seguintes funções sobre este tipo:

   (a) `void initQueue (QUEUE *q)` que inicializa uma queue (passa a representar uma queue vazia)

(b) `int isEmptyQ (QUEUE *q)` que testa se uma queue é vazia

(c) `int enqueue (QUEUE *q, int x)` que acrescenta `x` ao fim de `q`; a função deve retornar `0` se a operação fôr feita com sucesso (i.e., se a queue ainda não estiver cheia) e `1` se a operação não fôr possível (i.e., se a queue estiver cheia).

(d) `int dequeue (QUEUE *q, int *x)` que remove de uma queue o elemento que está no início. A função deverá colocar no endereço `x` o elemento removido. A função deverá retornar `0` se a operação for possível (i.e. a queue não está vazia) e `1` em caso de erro (queue vazia).

(e) `int front (QUEUE *q, int *x)` que coloca no endereço `x` o elemento que está no início da queue (sem modificar a queue). A função deverá retornar `0` se a operação for possível (i.e. a queue não está vazia) e `1` em caso de erro (queue vazia).

3. Nas representações de stacks e queues sugeridas nas alíneas anteriores o array de valores tem um tamanho fixo (definido pela constante `MAX`). Uma consequência dessa definição é o facto de as funções de inserção (`push` e `enqueue`) poderem não ser executadas por se ter excedido a capacidade das estruturas.

   Uma definição alternativa consiste em não ter um array com tamanho fixo e sempre que seja preciso mais espaço, realocar o array para um de tamanho superior (normalmente duplica-se o tamanho do array).

   Considere então as seguintes definições alternativas e adapte as funções definidas atrás para estas novas representações.

   Use as funções `malloc` e `free` cujo tipo está definido em `stdlib.h`.

   (a)
   ```
   typedef struct stack {
       int size; // guarda o tamanho do array valores
       int sp;
       int *valores;
   } STACK;
   ```

   (b)
   ```
   typedef struct queue {
       int size; // guarda o tamanho do array valores
       int inicio, tamanho;
       int *valores;
   } QUEUE;
   ```

4. Para gerir a informação sobre os alunos inscritos a uma dada disciplina, é necessário armazenar os seguintes dados:

   - Nome do aluno (string com no máximo 60 caracteres)
   - Número do aluno
   - Nota

   (a) Defina os tipos `Aluno` e `Turma`. Assuma que o número de alunos nunca ultrapassa 100, podendo por isso usar um array para armazenar a informação da turma.

   (b) Defina uma função `int acrescentaAluno (Turma t, Aluno a)` que acrescenta a informação de um dado aluno a uma turma. A função deverá retornar `0` se a operação for feita com sucesso.

(c) Defina uma função `int procura (Turma t, int numero)` que procura esse aluno na turma. A função deve retornar `-1` se a informação desse aluno não existir; caso exista deve retornar o índice onde essa informação se encontra.

(d) Defina uma função que determine quantos alunos obtiveram aproveitamento à disciplina (nota final maior ou igual a 10).

2 Listas Ligadas

1. Considere a seguinte definição de um tipo para representar listas ligadas de inteiros.

```
typedef struct slist *LInt;

typedef struct slist {
    int valor;
    LInt prox;
} Nodo;
```

(a) Apresente uma sequência de instruções que coloque na variável `a` do tipo `LInt`, uma lista com 3 elementos: 10, 5 e 15 (por esta ordem).

(b) Apresente definições (preferencialmente não recursivas) para:

i. `LInt cons (LInt l, int x)` que acrescenta um elemento no inicio da lista.

ii. `LInt tail (LInt l)` que remove o primeiro elemento de uma lista não vazia (libertando o correspondente espaço).

iii. `LInt init (LInt l)` que remove o último elemento de uma lista não vazia (libertando o correspondente espaço).

iv. `LInt snoc (LInt l, int x)` que acrescenta um elemento no fim da lista.

v. `LInt concat (LInt a, LInt b)` que acrescenta a lista `b` a `a`, retornando o início da lista resultante).

2. As duas últimas funções referidas na alínea anterior são muito pouco eficientes porque obrigam a percorrer uma lista apenas para nos posicionarmos no seu último elemento. Uma forma de melhorarmos a eficiência dessas operações consiste em guardar, para cada lista, dois endereços: o da primeira e o da última componentes.

```
typedef struct difl {
    LInt inicio, fim;
} DifList;
```

Redefina agora as duas operações da alínea anterior usando este novo tipo.

(a) `DifList snoc (DifList l, int x)`

(b) `DifList concat (DifList a, DifList b)`

3. Suponha que para resolver o problema descrito na secção 1 se optou por usar uma lista ligada em vez de um array.

   (a) Defina os novos tipos de dados para esta implementação.

   (b) Apresente definições das funções `acrescentaAluno`, `procura` e `aprovados` para esta nova implementação.

   Tenha o cuidado de rever os tipos destas funções nesta nova implementação.

4. Considere que os dados sobre os alunos e as notas dos minitestes vão ser lidos a partir do teclado com o seguinte formato:

   - na primeira linha será lido um inteiro `n` que representa o número de alunos inscritos.
   - de seguida aparece a informação (número e nome) de cada aluno inscrito (um por linha, correspondendo por isso a `n` linhas)
   - finalmente aparecem as notas dos alunos (uma por linha), em que cada linha contém o número do aluno e a classificação.

   Escreva um programa que lê a informação com este formato (do teclado) e escreve no ecran a informação sobre cada aluno (número, nome e nota. Na nota deve aparecer uma das seguintes

   - Um número de 10 a 20
   - `F` se não houver informação sobre a nota
   - `R` se a nota for menor do que `9.5`

5. Considere a seguinte definição para implementar listas duplamente ligadas de inteiros:

```
typedef struct node *DList;

typedef struct node {
    int value;
    DList prev, next;
} Node;
```

   Defina funções de processamento destas listas (nas duas primeiras alíneas assuma que se pretende manter as listas ordenadas por ordem crescente)

   (a) `DList addInt (DList l, int x)` que acrescenta um elemento à lista.

   (b) `DList exists (DList l, int x)` que determina se um elemento existe na lista; no caso de existir deve retornar o endereço da correspondente célula; caso contrário deve retornar `NULL`. Comece por definir duas funções `DList lookLeft (DList l, int x)` e `DList lookRight (DList l, int x)` que procuram um elemento para a direita ou para a esquerda.

   (c) `DList remove (DList l)` que remove um nodo da lista (libertando o correspondente espaço em memória).

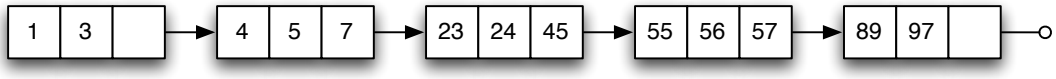
(d) `DList rewind (DList l)` que retorna o endereço do primeiro nodo da lista (`NULL` caso a lista seja vazia).

(e) `DList forward (DList l)` que retorna o endereço do último nodo da lista (`NULL` caso a lista seja vazia).

6. Uma forma de representar conjuntos de inteiros consiste em usar uma lista de blocos em que cada bloco consiste num array de inteiros **ordenado e sem repetições**. A lista também se encontra ordenada, no sentido em que o primeiro elemento de cada bloco é maior do que o último do bloco anterior. Por exemplo, se quisermos representar o conjunto

   `1, 3, 4, 5, 7, 23, 45, 55, 56, 57, 89, 97`

   e o tamanho de cada array for 3, podemos ter a seguinte lista:



   Considere então a seguinte definção:

```
#define N ...
typedef struct bloco {
    int quantos; // elementos ocupados
    int valores[N];
    struct bloco *prox;
} Bloco, *LBoco;
```

   (a) Apresente uma definição da função `int pertence (LBloco l, int x)`, que, dado um conjunto representado desta forma, e um inteiro, testa se esse inteiro pertence ao conjunto. A função deve retornar `1` caso o elemento pertença e `0` no outro caso.

   (b) Apresente uma definição da função `int quantos (LBloco l)` que, dado um conjunto representado desta forma, calcula o número de elementos do conjunto.

   (c) Apresente uma definição da função `int acrescenta (LBloco *l, int n)` que acrescenta um inteiro a um conjunto. A função deve devolver `0` em caso de sucesso.

   Assegure que, caso o conjunto tenha de ser representado em mais do que um bloco, nenhum bloco está menos do que 50% ocupado. Por exemplo, no exemplo acima, em que o tamanho do bloco é 3, nenhum bloco tem menos do que 2 posições ocupadas.

   (d) Apresente uma definição da função `int compacta (LBloco *l)` que reorganiza os números pelos blocos de forma a que todos os blocos (com possível excepção para o último, estão completamente preenchidos.

   A função deve devolver o número final de blocos e libertar todo o espaço não usado.