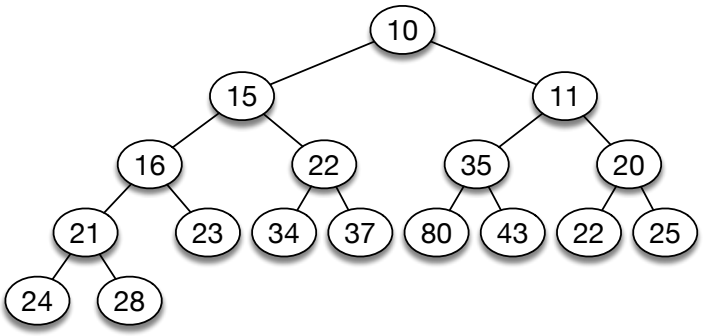
Ficha 3

Estruturas de Dados

Algoritmos e Complexidade
LEI / LCC / LEF

1 Min-heaps

Uma *min-heap* é uma árvore binária em que cada nodo é menor ou igual a todos os seus sucessores.
Por outro lado, uma árvore diz-se semi-completa se todos os níveis da árvore estão completos, com a possível excepção do último, que pode estar parcialmente preenchido (da esquerda para a direita).
As árvores semi-completas têm uma representação "económica" em array: os nodos são armazenados por nível, sempre da esquerda para a direita.
Por exemplo, a árvore (que é uma min-heap)



pode ser armazenada no array (de tamanho 17)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	15	11	16	22	35	20	21	23	34	37	80	43	22	25	24	28

1. Analisando o exemplo acima, determine expressões gerais que permitam calcular:
   (a) O índice onde se encontra a sub-árvore esquerda do nodo da posição `i`.
   (b) O índice onde se encontra a sub-árvore direita do nodo da posição `i`.
   (c) O índice onde se encontra o *pai* do nodo da posição `i`.
   (d) O índice onde se encontra a primeira folha, i.e., o primeiro nodo que não tem sucessores.

2. Defina a função `void bubbleUp (int i, int h[])` que (por sucessivas trocas com o *pai*) *puxa* o elemento que está na posição `i` da min-heap `h` até que satisfaça a propriedade das min-heaps.

   Identifique o pior caso desta função e calcule o número de comparações/trocas efectuadas nesse caso.

3. Defina a função `void bubbleDown (int i, int h[], int N)` que (por sucessivas trocas com um dos *filhos*) *empura* o elemento que está na posição `i` da min-heap `h` até que satisfaça a propriedade das min-heaps.

   Identifique o pior caso desta função e calcule o número de comparações/trocas efectuadas nesse caso.

4. Considere agora o problema de implementar uma fila com prioridades, i.e., uma fila em que o próximo elemento a retirar da fila é o menor que lá estiver.

   Uma possível implementação desta estrutura de dados consiste em usar uma min-heap.

```
#define Max 100
typedef struct pQueue {
    int valores [Max];
    int tamanho;
} PriorityQueue;
```

   Apresente as definições das operações habituais sobre este género de tipos (*buffers*).

   - `void empty (PriorityQueue *q)` que inicializa `q` com a fila vazia.
   - `int isEmpty (PriorityQueue *q)` que testa se está vazia.
   - `int add (int x, PriorityQueue *q)` que adiciona um elemento à fila (retornando `0` se a operação for possível).
   - `int remove (PriorityQueue *q, int *rem)` que remove o próximo elemento (devolvendo-o em `*rem`) e retornando `0` se a operação for possível.

5. A operação `void heapify (int v[], int N)` consiste em obter uma permutação do array que seja uma min-heap.

   Duas estratégias para implementar esta função são:

   top-down: Assumindo que as primeiras `p` posições do array constituem uma min-heap (de tamanho `p`) efectuar a invocação `bubbleUp (p, v, N)` de forma a obtermos uma min-heap de tamanho `p+1`.

   bottom-up: Para cada nodo da árvore, desde o mais profundo até à raiz, aplicar a função `bubbleDown`. Note-se que a invocação para as folhas é desnecessária, uma vez que não têm sucessores.

   Implemente a função `heapify` usando estas duas estratégias.

   Para cada uma delas, identifique a complexidade dessa função no caso em que o array original está ordenado por ordem decrescente.

6. Defina uma função `void ordenaHeap (int h[], int N)` que, usando a função `bubbleDown` definida acima, transforma a min-heap `h`, num array ordenado por ordem decrescente.

7. Considere o problema de ler uma sequência de N números e seleccionar os k maiores, com $k < N$, (tipicamente, k muito menor do que N).

   Uma solução possível consiste em começar por ler os k primeiros elementos e organizá-los numa min-heap. Para cada um dos $N - k$ seguintes, caso seja maior do que o menor dos números organizados, insere-se esse elemento na min-heap, removendo o menos dos que lá estão.

   Analise o custo desta solução (no pior caso) comparando-o com outra solução alternativa de, por exemplo, armazenar os k maiores números lidos numa lista ligada ordenada por ordem crescente.

2 Tabelas de Hash

Nos exercícios seguintes pretende-se usar uma tabela de Hash para implementar multi-conjuntos de strings. Para cada string deve ser guardado o número de vezes que ela ocorre no multi-conjunto. As operações em causa são por isso:

- inicialização de um multi-conjunto a vazio
- adição de um elemento a um multi-conjunto
- teste de pertença (saber qual a multplicidade de um elemento num multi-conjunto)
- remoção de uma ocorrência de um elemento de um multi-conjunto

Vamos por isso assumir a existência de uma função `unsigned hash (char *chave)`, como por exemplo a seguinte (`http://www.cse.yorku.ca/~oz/hash.html`)

```
unsigned hash(char *str){
   unsigned hash = 5381;
   int c;

   while (c = *str++)
      hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

   return hash;
}
```

2.1 Chaining

Vamos usar o seguinte tipo.

```
#define Size ...
typedef struct nodo {
    char *chave; int ocorr;
    struct nodo *prox;
} Nodo, *THash [Size];
```

Defina as funções usuais sobre multi-conjuntos:

1. `void initEmpty (THash t)` que inicializa um multi-conjunto a vazio
2. `void add (char *s, THash t)` que regista mais uma ocorrência de um elemento a um multi-conjunto
3. `int lookup (char *s, THash t)` que calcula a multiplicidade de um elemento num multi-conjunto
4. `int remove (char *s, THash t)` que remove uma ocorrência de um elemento de um multi-conjunto.

2.2 Open Addressing

Vamos usar o seguinte tipo.

```
#define Size ...
#define Free 0
#define Used 1
#define Del  2
typedef struct bucket {
    int status; // Free | Used | Del
    char *chave; int ocorr;
} THash [Size];
```

1. Comece por definir a função `int where (char *s, THash t)` que calcula o índice de `t` onde `s` está (ou devia estar) armazenada.
2. Defina as funções usuais sobre multi-conjuntos:
   (a) `void initEmpty (THash t)` que inicializa um multi-conjunto a vazio
   (b) `void add (char *s, THash t)` que regista mais uma ocorrência de um elemento a um multi-conjunto
   (c) `int lookup (char *s, THash t)` que calcula a multiplicidade de um elemento num multi-conjunto
   (d) `int remove (char *s, THash t)` que remove uma ocorrência de um elemento de um multi-conjunto.
3. Defina a função `int garb_collection (THash t)` que reconstrói a tabela `t` de forma a não haver chaves apagadas (`status==Del`).

   Analise a complexidade desta função.
4. Considere que se mantem um registo de quantas células da tabela estão apagadas, e que, sempre que este número atinge os 25% da capacidade do array, é feita uma `garb_collection`. Assumindo que o custo de uma remoção sem `garb_collection` é constante, determine o custo amortizado da remoção.

5. Uma forma de diagnosticar a *qualidade* da tabela de hash consiste em acrescentar, em cada célula (*bucket*), a informação do número de colisões que a inserção dessa chave teve que resolver.

   Modifique a definição da função de inserção apresentada acima de forma a armazenar também essa informação.

```
typedef struct bucket {
    int probC;
    int status; // Free | Used | Del
    char *chave; int ocorr;
} THash [Size];
```