Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi.
- Quando caricate il codice sul sistema assicuratevi che siano presenti tutte le direttive di `include` necessarie, comprese quelle per l'utilizzo delle primitive**.** Non dovete caricare l'implementazione delle primitive.
- **È importante scrivere il proprio main in Visual Studio per poter fare correttamente il debug delle funzioni realizzate!**

Esercizio 1 (punti 8)

Creare i file `newton.h` e `newton.c` che consentano di definire la seguente struttura:

```
typedef struct polinomio {
	int *coeffs;
	size_t size;
} polinomio;
```

e la funzione:

```
extern double Newton(const polinomio *p, const polinomio *d, double xn, double t,
                     int max_iter);
```

La `struct` consente di rappresentare una funzione polinomiale di grado n a coefficienti interi. I coefficienti del polinomio sono memorizzati a partire da quello di grado minore nel vettore `coeffs`. La dimensione del vettore `coeffs` è memorizzata nel campo `size` della `struct`. Dato ad esempio il polinomio $x^3 - 2x$, la `struct` che lo rappresenta conterrà `4` nel campo `size` e `coeffs` punterà ad un'area di memoria contenente i seguenti valori `0,-2,0,1`.

La funzione deve applicare ricorsivamente il metodo di Newton (anche noto come metodo delle tangenti) per calcolare in modo approssimato uno zero di una funzione polinomiale data. La formula ricorsiva del metodo di Newton è la seguente:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Dove x_n è l'approssimazione dello zero al passo n, x_{n+1} è l'approssimazione dello zero al passo $n+1$, $f(x_n)$ è il valore della funzione polinomiale in x_n e $f'(x_n)$ è il valore della derivata della funzione polinomiale in x_n.

La ricorsione deve terminare quando si verifica una delle seguenti condizioni:

1. l'approssimazione corrente della soluzione (x_{n+1}) è sufficientemente buona, ovvero dato un parametro di tolleranza di approssimazione `t`, si verifica la condizione

$$|x_{n+1} - x_n| \leq t$$

2. il numero di iterazioni effettuate è uguale al numero massimo di iterazioni ammesse `max_iter`. Se il parametro `max_iter` è minore o uguale a `0` vale solo la condizione di uscita 1.

La funzione prende in input i seguenti parametri:

- un `polinomio p` che rappresenta la funzione polinomiale di cui calcolare uno zero;
- un `polinomio d` che rappresenta la derivata di `p`;
- un valore iniziale `xn` da cui far partire il procedimento ricorsivo;
- il parametro di tolleranza `t`;
- il numero massimo di iterazioni `max_iter`.

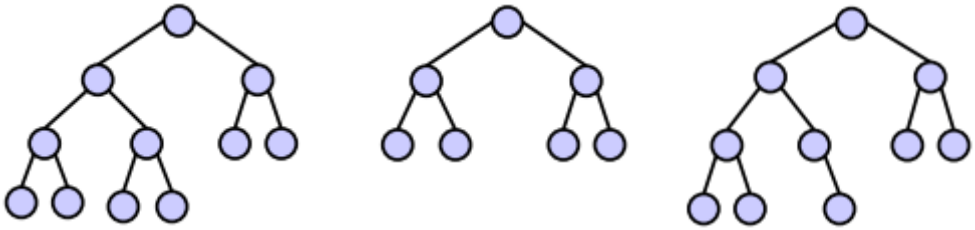
La funzione deve ritornare la soluzione individuata.

Si consiglia di utilizzare una funzione ausiliaria per la risoluzione dell'esercizio. Non saranno considerate valide soluzione che non fanno uso della ricorsione.

Esercizio 2 (punti 7)
Nel file `isheap.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool IsHeap(element *h);
```

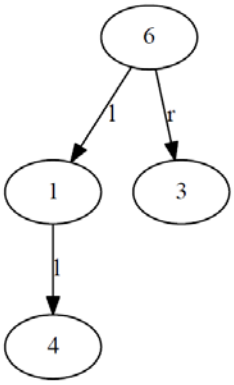
La funzione prende in input un albero binario di interi memorizzato in un array. L'albero di input è quasi completo, ovvero tutti i livelli dell'albero, ad eccezione eventualmente dell'ultimo, sono completi; nell'ultimo livello possono mancare alcune foglie consecutive a partire dall'ultima foglia a destra. Ad esempio, gli alberi binari A e B sono quasi completi, mentre C è un albero binario che non è quasi completo.



(A) (B) (C)

La funzione deve scorrere l'albero di input e ritornare `true` se questo è un (max)heap o è vuoto, `false` altrimenti.

Si ricorda che un array rappresentante un albero binario ha la seguente struttura: il primo elemento dell'array contiene il numero di nodi presenti nell'albero, il secondo elemento contiene la radice, il terzo e il quarto elemento contengono rispettivamente il figlio sinistro e destro della radice e così via. Ad esempio, l'array `4, 6, 1, 3, 4` rappresenta l'albero binario quasi completo seguente:



Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int element;
```

e le seguenti primitive:

```
int Left(int i);
int Right(int i);
int Parent(int i);
int Compare(const element *e1, const element *e2);
void Swap(element *e1, element *e2);
void MoveUpMaxHeap(element *h, int i);
void MoveDownMaxHeap(element *h, int i);
element *CreateHeap();
void FreeHeap(element *h);
element* HeapifyMaxHeap(element *v, size_t v_size);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `heap_int.h` e `heap_int.c`. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20191207_esame/heap_int/heap__int_8h.html

Esercizio 3 (punti 8)
Nel file `loadtree.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern tree LoadTree(const char *filename);
```

La funzione prende in input il nome di un file `filename` che contiene i dati di un albero binario di caratteri. I dati all'intero del file sono così strutturati:

- ogni carattere alfanumerico identifica un nodo dell'albero;
- le foglie sono precedute dal carattere `'.'`;
- ogni nodo è seguito dal figlio sinistro (che a sua volta può avere figli) e poi da quello destro (che a sua volta può avere figli);
- ogni nodo, ad eccezione delle foglie, ha sempre entrambi i figli.
- i `whitespace` (`' '`, `'\t'`, `'\r'`, `'\n'`, `'\v'`, `'\f'`) all'interno del file non hanno alcun significato.

Ad esempio il seguente file:

```
a
	.x
	b
		d
			.s
			.u
		.c
```

rappresenta l'albero riportato in Figura 1. Si noti che un file con il seguente contenuto:

```
a
.x
b
d
.s
.u
.c
```

rappresenta lo stesso albero poiché, come descritto nelle specifiche del file, i `whitespace` al suo interno non hanno alcun significato.

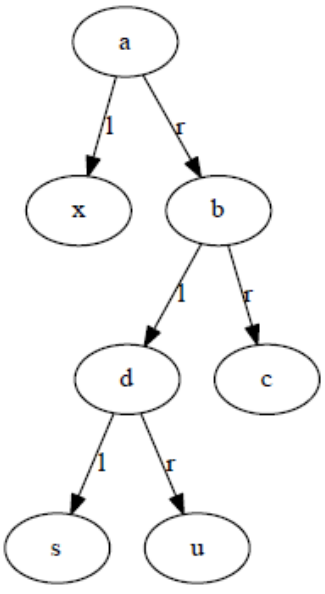


Figura 1

La funzione deve aprire il file in lettura, leggerne il contenuto e costruire l'albero corrispondente che deve quindi essere ritornato. Il file di input è sempre correttamente formato, ovvero rispetta sempre la sintassi precedentemente definita. Se non è possibile aprire il file o se il file è vuoto la funzione deve ritornare un albero vuoto (`NULL`).

Si consiglia di utilizzare una funzione ausiliaria per la risoluzione dell'esercizio.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef char element;
typedef struct tree_element {
       element value;
       struct tree_element *left, *right;
} node;
typedef node* tree;
```

e le seguenti primitive:

```
tree EmptyTree();
tree ConsTree(const element *e, tree l, tree r);
bool IsEmpty(tree t);
element *GetRoot(tree t);
tree Left(tree t);
tree Right(tree t);
bool IsLeaf(tree t);
tree InsertBinOrd(const element *e, tree t);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `tree_char.h` e `tree_char.c`. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20191207_esame/tree_char/tree__char_8h.html

Esercizio 4 (punti 10)
Nel file `concatena.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern list Concatena(list *ls, int ls_size);
```

La funzione prende in input un vettore di puntatori a liste `ls` e la sua dimensione `ls_size`. Ogni elemento del vettore `ls` può contenere l'indirizzo della testa di una lista o `NULL`. La funzione deve concatenare tutte le liste contenute in `ls` a partire dalla prima e ritornare la lista risultante, ovvero l'indirizzo della sua testa.

Se tutti i puntatori in `ls` sono `NULL` o se `ls_size` è `0`, la funzione deve ritornare una lista vuota.

Nel caso in cui si scelga di creare una nuova lista, contenente il risultato della concatenazione, al posto di concatenare le liste esistenti, la memoria dovrà essere opportunamente liberata. Ciò significa che la memoria complessiva allocata dovrà essere la stessa prima e dopo l'esecuzione della funzione.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int element;
typedef struct list_element {
  element value;
```

```
  struct list_element *next;
} item;
typedef item* list;
```

e le seguenti primitive:

```
list EmptyList();
list Cons(const element *e, list l);
bool IsEmpty(list l);
element Head(list l);
list Tail(list l);
element Copy(const element *e);
void FreeList(list l);
list InsertBack(list l, const element *e);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `list_int.h` e `list_int.c`. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20191207_esame/liste_int/list__int_8h.html