Exercise 5.4.3 Extend Exercise 5.4.2 using a *Binary Search* for this. ■

Exercise 5.4.4 Write a program which, based on a linked list data structure, reads the words in one at a time, inserting them into the **correct** part of the list so that the words are alphabetically sorted. The name of the file should be passed as `argv[1]`. How long does it take to build the list ?
How long does it take to do a linear search ? ■

5.5 Crush It!

Match-3 tile games have become one of the world's most popular games.

www `https://en.wikipedia.org/wiki/Tile-matching_video_game`

Such games use a rectangular grid (board) containing many tiles, of many different types (often colour, but here we will use letters). Where there are 3 or more tiles of the same type in a line (horizontally or vertically), they are removed. Once all removals that are possible have occurred, the tiles above fall down to fill in the gaps. In our version, a large number of tiles are available above (and hidden from the player) which cannot be matched until they have fallen down into the playing area.

In our version of the game:

- Matchable tiles are in the range $(A \ldots Z)$, although it's is common for only a small number (e.g. $A \ldots D$) to be used.
- The width of the board is always five tiles.
- The 'playing' height of the board is six. Other tiles can be above this, but won't be matched until they have dropped down into one of the bottom six rows.
- The maximum number of rows the board ever needs to hold is 20.
- Matching (that is finding a horizonal or vertical line of the same tile) is done in 'parallel' - if a tile is shared between two matches (e.g. the middle tile in a 3×3 '+' pattern) both of these matches are removed.
- Given the limited size of board in which matches can be made, the longest match that can be made is of five tiles horizontally (the width of the board) and six tiles vertically the 'playing' height of the board.

An example of this is shown here:

·	·	·	·	·
B	B	B	D	B
C	D	A	A	C
D	A	A	B	D
A	A	B	C	A
A	B	C	D	A
B	C	A	A	A

·	·	·	·	·
·	·	·	D	B
C	D	A	A	C
D	A	A	B	D
A	A	B	C	·
A	B	C	D	·
B	C	·	·	·

·	·	·	·	·
·	·	·	·	·
C	D	·	D	·
D	A	A	A	·
A	A	A	B	B
A	B	B	C	C
B	C	C	D	D

(Left) Game board in it's initial state. Orange squares show were matches can be made. (Middle) Three matches are made - one for the 3 horizonal 'A' tiles, one for the three vertical 'A' tiles and one for the three horizontal 'B' tiles. (Right) Game board final state after tiles are dropped down. Not there are now more matches that can be made.

Exercise 5.5.1 Here we will write some (but not all) of the functionality necessary for a match-3 tile game. Skeleton code may be found in :

www `https://github.com/csnwc/Exercises-In-C`

then navigate into `Code/Week5/CrushIt`.

Complete the files **crushit.c** and **mydefs.h** which, along with my files *crushit.h* and *driver.c*, implements some important functionality necessary for a game of this type.
My file *crushit.h* contains the function definitions which you'll have to implement in your **crushit.c** file. My file *driver.c* contains the `main()` function to act as a driver to run the code. Your file will contain many other functions as well as those specified, so you'll wish to test them as normal using a `test()` function.
If all of these files are in the same directory, you can compile them using the `Makefile` given. The functions you need to complete include:

`initialise()` - this takes a pointer to the board state, and a string. The string can be a filename, but if this filename can't be opened, it is assumed to be a list of tiles to fill the board with, from the top down. Such a string must contain complete rows of tiles, with no partial rows.

`match()` - this takes a pointer to the board state, and removes all matches of 3 or more tiles in a vertical or horizontal line. Removed tiles are replaces with the '.' (empty tile) character.

`dropblocks()` - this takes a pointer to the board state, and makes any blocks that are above an empty tile fall down until all holes are filled in if it is possible to do so.

`tostring()` - this takes a pointer to the board state, and a string and copies whole rows of the board into the string from the top downwards. The whole board isn't copied since most of the characters at the top are unused (hole) tiles. Therefore, we begin copying at the first row on which a non-hole tile appears.
Hints:

- Do not begin by writing the file handling functionality - this cannot be tested, so use the string initialising option instead.
- To begin with use your own, simpler driver file - mine makes sense once everything is working, but may seem complex to begin with.
- Your `crushit.c` file should contain many other sub-functions which are used by the major ones specified. You can put anything in this file, provided it still compiles as specified.
- Do not alter or resubmit *crushit.h*, *Makefile* or *driver.c* - my original versions (or even slightly different ones) will be used to compile the *crushit.c* file that you adapt.

■