COMP10002 Workshop Week 10

First Hour	• Understanding linked lists & BST • Discuss Exercises 8 and 9 in the lec07 lecture slides.
Lab Time	Two main streams: - Working on quiz3 - Working on string/malloc exercises
LMS requirements	• Find another nursery rhyme that you like, and insert its words into a binary search tree • Discuss Exercises 8 and 9 in the lec07 lecture slides. • Then look at Exercises 4, 5, 6, and 7 in lec07.pdf, and implement and test solutions to at least two of them.
quiz 3	Quiz 3 will cover Chapters 1 to 8, plus Section 10.1: all of lec06.pdf and the first half of lec07.pdf through to video lec07-d, including malloc() and realloc() but not linked lists or binary search trees.xt
Next Week	Number representation Assignment 2

BST, insert and search in BST, BST vs arrays

Find another nursery rhyme that you like, and insert its words into a binary search tree

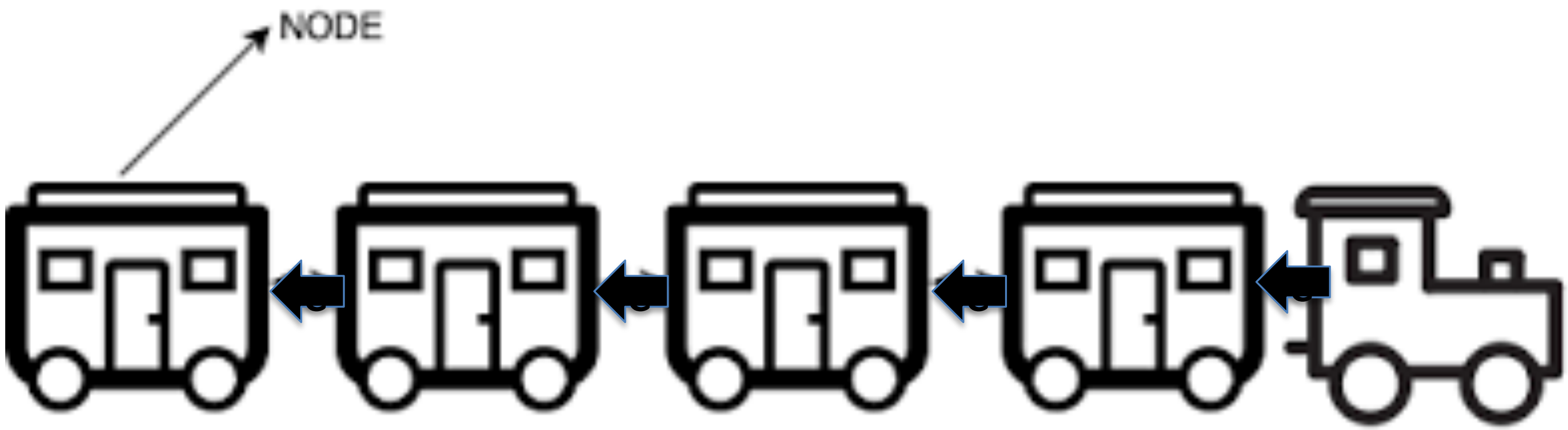
12345 Once I caught A fish Alive

One, two, three, four, five,
Once I caught a fish alive,
Six, seven, eight, nine, ten,
Then I let go again.

Why did you let it go?
Because it bit my finger so.
Which finger did it bite?
This little finger on the right

The focus of this picture is …

... a linked list. Linked lists: insert/delete at 2 ends



Exercise 9 (in the context of linked lists)

Suppose that insertions and extractions are required at both head and foot. **How can delete foot() be implemented efficiently?** (Hint, can a second pointer be added to each node?)

Exercise 8: Implementing Stacks Using Arrays

Stacks and queues can also be implemented using an array of type data t, and static variables. Give functions for make empty `stack()` and `push()` and `pop()` in this representation.

Programming:

- Test it by adding 10 integers:

  `0 1 2 … 9`

  to a stack, then print them (in reverse order).
- In the same program, add ten 2D points

  `(0, 0) (1,10) (2, 20) … (9,90)`

  then print them in reverse order

Note: no class on Friday next week (week 11)

Class Replacement

Exercises 4-7 from lec07.pdf

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string s and returns a pointer to it.

Exercise 5

Write a function `char **string_set_dupe(char **S)` that creates a copy of the set of string pointers `S`, assumed to have the structure of the set of strings in `argv` (including a sentinel pointer of `NULL`), and returns a pointer to the copy.

Exercise 6

Write a function) `void string_set_free(char **S)` that returns all of the memory associated with the duplicated string set S.

Exercise 7

Test all three of your functions by writing scaffolding that duplicates the argument `argv`, then prints the duplicate out, then frees the space.
(What happens if you call `string_set_free(argv)`? Why?)

Exercises 4-7 from lec07.pdf

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string s and returns a pointer to it.

Exercise 5

Write a function `char **stri`
creates a copy of the set of string
structure of the set of strings in `a`
`NULL`), and returns a pointer to t

Exercise 6

Write a function) `void strin`
returns all of the memory associa

Exercise 7

Test all three of your functions by
argument `argv`, then prints the dup
(What happens if you call `string_set_free(argv)`? Why?)

I am organising 3 rooms:

- Send me a short message "G" if you want to join a group and do some programming on exercises 4-7
- Send "O" if you want to join a group for other programming exercises/discussions.
- Send nothing if you want to stay in the main room for class discussions on
  - string matching algorithms
  - other quiz3 problems, including programming questions

Exercise 4 (from last workshop)

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

```
char *string_dupe(char *s) {
   char *t= malloc( (strlen(s)+1) * sizeof(char) );
   strcpy(t,s);
   return t;
}
... main ... {
  char *s= string_dupe(“Tada!”);
  ...
  free(s);
  return 0;
}
```

String matching: the task

Input:

- a text T[] such as "abab yxy a**ababc**b"
- a pattern P[] such as "ababc"

Ouput

- first position where P appears in T, or NOTFOUND

Output for the example:

- 10
- how many (pattern) shifts?
- how many comparisons?)

String matching: KMP & BMH

Both algorithms:

- start with aligning `P` with the start of `T`
- repeatedly shift `P` to the right as far as possible by comparing `P` with `T` character-by-character

But

- `KMP` compare pattern (with text) from *left to right*, why?
- `BMH` compare pattern (with text) from *right to left*, why?

String matching: KMP & BMH

Both algorithms:

- start with aligning `P` with the start of `T`
- repeatedly shift `P` to the right as far as possible by comparing `P` with `T` character-by-character

But

- `KMP` compare pattern (with text) from left to right, why?

  because `MP` *are in alphabetic order* 😀
- `BMH` compare pattern (with text) from right to left, why?

  but `MH` *are in reverse order* 😀

How to run KMP *manually*

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

5 comp until mismatch;

When mismatch, examine **only** the *left part of P* to decide the shift, but note that this left part is also the *currently matched part* between T and P:

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	X

In the algorithm: here unmatched position i= 4, F[i]= 2, and we need to shift P by i-F[i] = 4-2= 2 positions to the right.

Equivalently, *without* building F:

- shift P to the right step-by-step and stop when the prefix of the blue matches with the suffix of the green parts.

How to run KMP *manually*

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

prefix ”ab” matches suffix “ab”

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

		a	b	a	b	c

when having prefix-suffix match: shift P to align that match

Then repeat the process from the *current position* in T,
ie. from comparing y with its peer a in this case

How to run KMP *manually*

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

5 comp until mismatch; shift until "**ab**" aligned with "**ab**"
if having prefix-suffix match: align to that match

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

compare **y** with **a** : no prefix-suffix match
no prefix-suffix match: align the start of the pattern (**a**) with **y**

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

shift 1 for mismatch at 0-th char of pattern P
Note: also shift 1 for mismatch at 1-th position

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

now start with comparing **a** with **a**
(in the algorithm: from `0= max(F[i],0)`)

from now we will have 4 shift 1 for "**yb a**" because mistmatch happens at the 0-th or 1-st char of P (but note: 2 comp when mismatch as 1-st)

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a

a	b	a	b	c

FOUND (last alignment)

TOTAL: 9 alignments, 19 comparisons

How to run BMH *manually*

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a
a	b	a	b	c											

1 comp until mismatch
no matter where mismatch happens, the shift is totally decided by *the rightmost examined char* of T **y**

Shift P the whole length because y does not appear in the pattern P (in the algorithm: `S[x]= length(P)` if `x` is not in `P`

Note: In BMH, compare pattern (with text) from right to left.

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a
					a	b	a	b	c						

mismatch; **a** is in pattern, shift until **a** aligned with the first **a** in P (from right)

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a
							a	b	a	b	c				
								a	b	a	b	c			

shift until **b** aligned with the first **b**

shift until **a** aligned with the first **a**

a	b	a	b	y	a	y	b		a	a	b	a	b	c	a
										a	b	a	b	c	

Other problems with quiz, including programming?