COMP20003 Workshop Week 3

Linked Lists, Arrays vs Linked Lists

1. Dictionary ADT and some supporting DS
2. Linked Lists
   - a bit on Complexity
   - arrays: The Goods & The Bads
   - Linked Lists: What, Why, How

1. Q&A on `W3.4` and its `Makefile`
2. `gdb`
3. Assignment 1: Q&A

Dictionary ADT and some concrete DS for it

ADT (interface: what?)

ADT: Dictionary

- contains data records
- operations:

create, insert, search, delete, free

ID	Name
1007	Lord Voldermort
1001	Harry Potter
1010	Albus Dumbledore
• • •	

a datum

a key

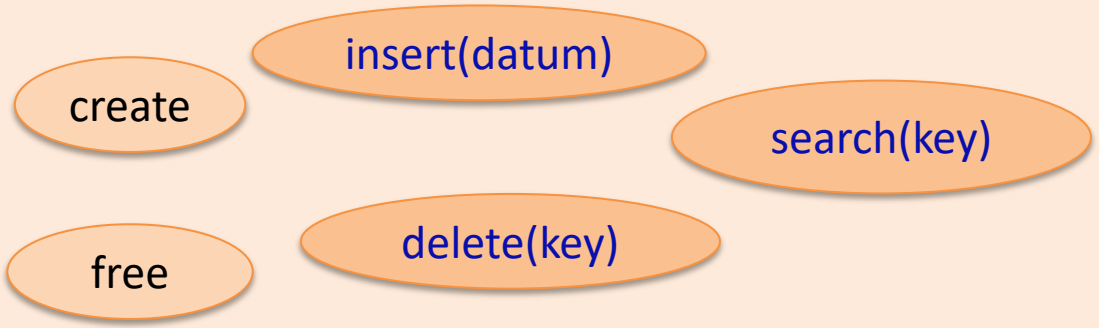
DS (implementation: how?)

- Static Array
- Dynamic Array
- Sorted Dynamic Array
- Linked List
- • • •

Dictionary as an ADT and some concrete DS

ADT: Dictionary

- contains data records, each record has a key
- operations:



ID	Name
1007	Lord Voldermort
1001	Harry Potter
1010	Albus Dumbledore
	• • •

Array (supposing size is OK)

insert(datum):
O(?) θ(?)
delete (key):
O(?) θ(?)
search(key):
O(?) θ(?)

Sorted Array

insert(datum):
O(?) θ(?)
delete (key):
O(?) θ(?)
search(key):
O(?) θ(?)

Linked List

insert(datum):
O(?) θ(?)
delete (key):
O(?) θ(?)
search(key):
O(?) θ(?)

- There is no single "best data structure" for all cases, each DS has its advantages and disadvantages
- But how to compare them? Use Asymptotic Complexity Analysis!

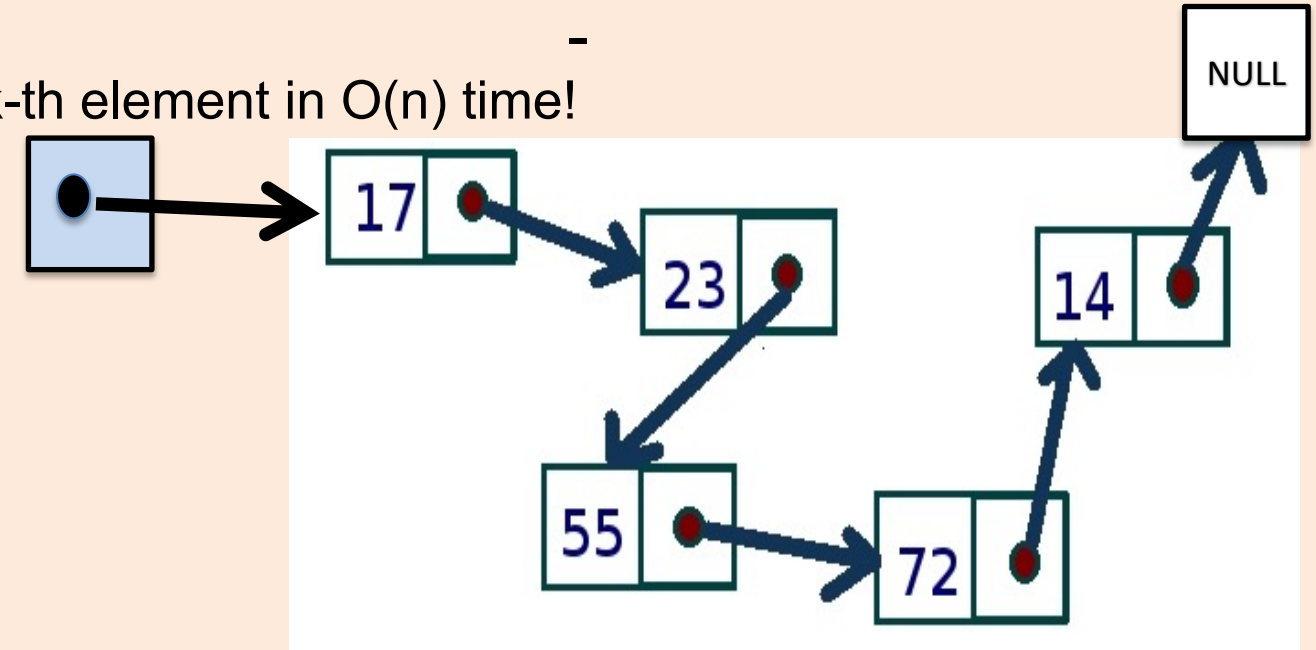
Arrays vs Linked Lists: random access and sequential access to k-th element

ARRAY

A (`A= malloc(10 * sizeof(*A);`)

LINKED LIST

Access k-th element in O(n) time!



So, arrays are better in accessing k-th element.
How about:

- insert a new element at the start?
- delete the first element?

Define Linked List: Method 1 (less popular)

Method 1: Linked list == pointer to the first node of the chain.

```
typedef struct node *list_t;
struct node {
   data_t data;
   struct node *next;
};
list_t L= createList();
...
```

✓ Simple and well abstracted!
✓ Efficient insert to the start.
✗ Inefficient insert to the end (hey, it's simple in arrays!)

Do
W3.3
Now

NOTES

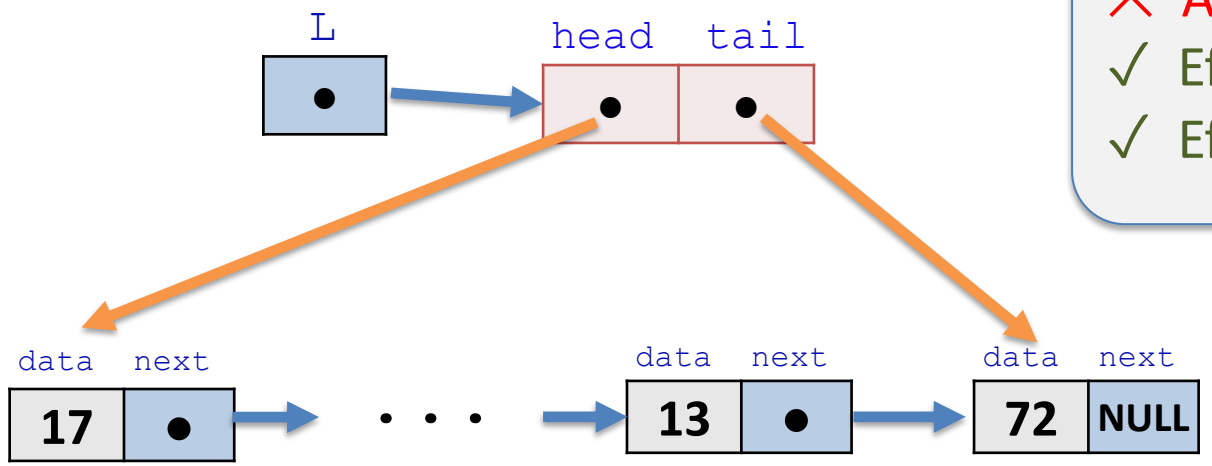
- `typedef` gives us a new data type `list_t` which is *equivalent* to `struct node *`
- To use or not to use `typedef` is the matter of personal preference

Define Linked List: Method 2 (popular)

Method 2: Linked list is a pair of pointers.

```c
typedef struct node node_t;
struct node {
    data_t data;
    struct node *next;
};
```

```c
typedef struct list list_t;
typedef struct {
    node_t *head;
    node_t *tail;
};
list_t *L= createList();...
```



✗ A bit more complicated.
✓ Efficient insert to the start.
✓ Efficient insert to the end.

Linked lists: Basic Operations

Basic operations:

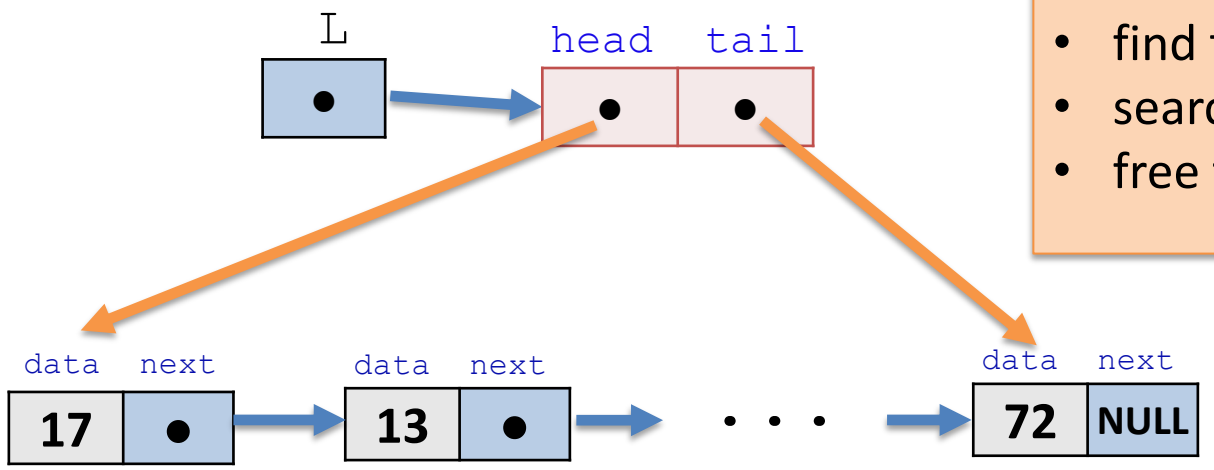
- `create`: create an empty list
- `prepend`: insert a data to the start of a list
- `append`: append a data to the end of a list
- `search`: search for a data in a list
- `deleteHead`: remove & return the first element
- `deleteTail`: remove & return the last element
- `free`: delete a list, free the memory

Q1: What should be considered when implementing insertion or deletion ?

- `prepend/append`: ?
- `deleteHead/deleteTail`: ?

Q2: We want a linked list of integers. How To:

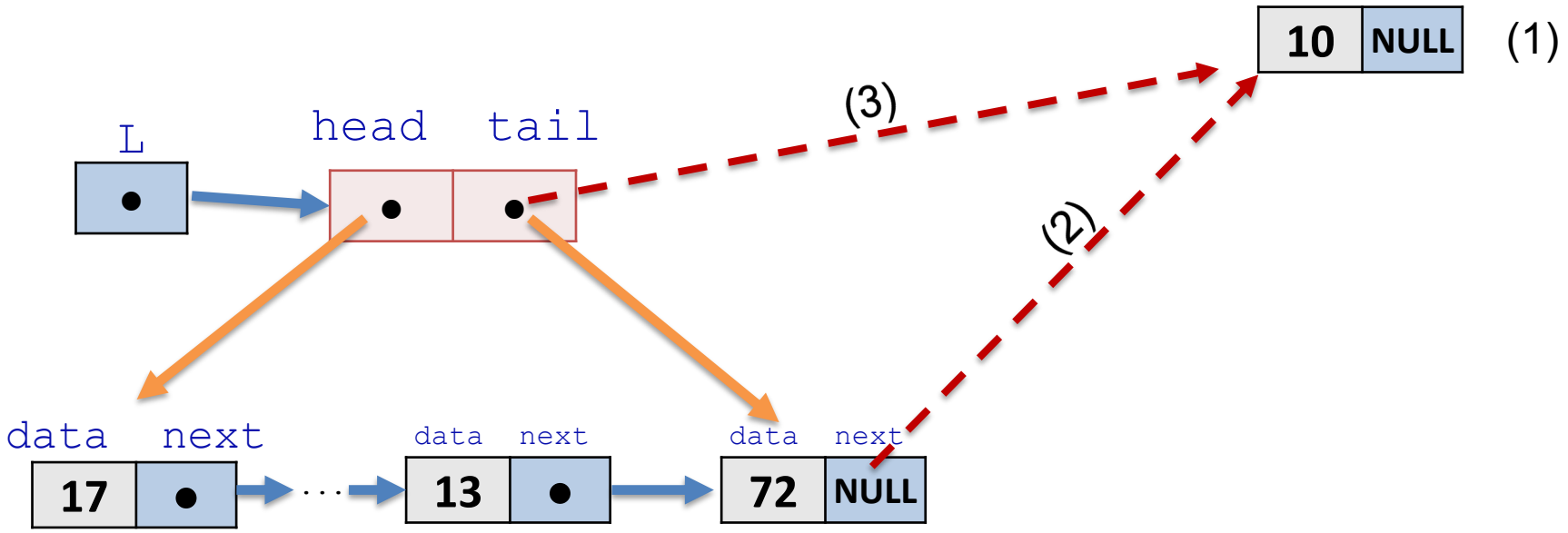
- build the linked list from scratch?
- find the sum of data in the list?
- search for a value in the list?
- free the list?



Linked List Example: `listAppend` : append node with data 10 – *is this correct?*

1. create new node and set data	`node_t *new= malloc(sizeof(*new));`
	`new->data= 10;  // here, data_t is int`
	`new->next= NULL;`

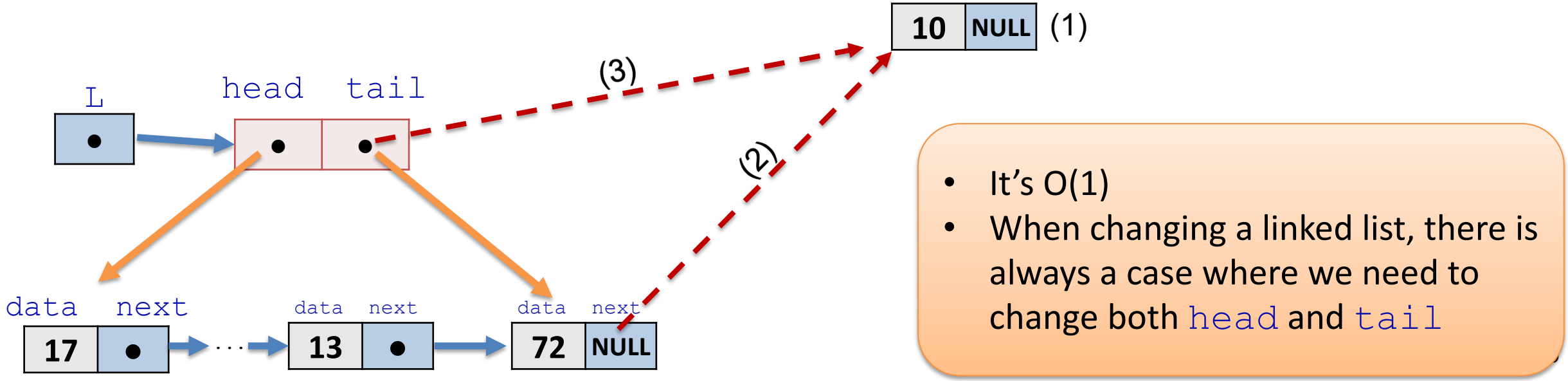
2. Link the node to the chain	`L->tail->next= new;`
3. Repair `tail`	`L->tail= new;`



Linked List Example: `listAppend` : append node with data 10

1. create new node and set data
2. Link the node to the chain
3. Repair `tail`
4. Repair `head` and `tail` if needed

```
node_t *new= malloc(sizeof(*new));
assert(new);
new->data= 10;    // here, data_t is int
new->next= NULL;
if (L->tail) {
   L->tail->next= new;
   L->tail= new;
} else
   L->head= L->tail= new;
```



- It's O(1)
- When changing a linked list, there is always a case where we need to change both `head` and `tail`

Peer Activity on `deleteHead` [5 min]

Discuss with classmates & answer W3.10 now.

Assignment 1: Q&A (15 minutes)
Remember to watch the video in the section `General`

THE TASK
Get the Green Tick for all items In `Assignment1`

MARKING RUBRIC

- Correctness: 4 marks
- Correct `malloc`/`realloc`/`free`: 2 marks
- Space-efficiency: 1 mark
- Programing style, modularity, checking `malloc`/`realloc` and `fopen`: 2 marks
- Self Evaluation: 1 mark (**hurdle**)
- Submission Certificate: 0 mark (**hurdle**)

It's a
GROUP ASSIGNMENT

- Build your group of 2
- Discuss section "Teamwork Recommendation" with your team

Assignment 1: you must store data efficiently

Space-efficiency: 1 mark

Linked list: each node has a pointer to `data_t`

Array: must be dynamic array of pointers to `data_t`

Data field inside structure of `data_t`:

- integer values should be stored as `int`
- floating point values should be stored as `double`
- do not store numeric values as strings: that would waste memory
- a string value should be stored as a string with the least possible amount of bytes (using `strdup`, for example)

Assignment 1:
Correct `malloc`/`realloc`/`free`: 2 marks

Remember:

- Each execution of `malloc` needs one execution of `free`
- Practically, no free is needed for `realloc`
- You need to make sure that you have a clean `valgrind` report

Assignment 1:

Programing style, modularity, checking `malloc`/`realloc` and `fopen`: 2 marks

Modularity

- You have good examples in **`qStud`** for style and modularity, including `Makefile`
- Try to design your modules and build your `Makefile` from scratch first, then compare your design with the similar `qStud (`as in `W3.4`, or `W3.5`, or ...)

Assignment 1:

Programing style, modularity, checking `malloc`/`realloc` and `fopen`: 2 marks

`malloc`/`realloc`/`fopen`:

- After every `malloc`/`realloc`/`fopen` and even `stdup` you need to use `assert` to check.

Good Practice:

Write a function, say, `my_malloc`

```
void *my_malloc(size_t n) {
    void *p= malloc(n);
    assert (p);
    return p;
}
```

then use it instead of `malloc`.

- The same is applied to `realloc`, `fopen`, `strdup`.
- It's convenient to place all these helping functions in a module, say, `utils` (with `utils.h` and `utils.c`).

Assignment 1: Notes on input *for our well-formatted data*

- `sscanf` and `fscanf` are equivalent in functionality, but for `sscanf` you need to read the whole line into a buffer before hand and is a bit more complicated to process.
- if you read a whole line into a string buffer, you can:
  - consider to use `strsep` – it can handle empty strings,
  - or process the buffer character-by-character

NOTES: *for a string field*

- scanf, fscanf, sscanf, and strsep cannot automatically handle the whole string if it is given in quotes like

  `"Parkville, VIC 3053"`
- You'd better to have a function that reads a string for a general case (and perhaps returns a `char *)`

Assignment 1: using external codes

You can use all the codes you have in lectures and workshops, but you *need to **add references***. Remember, *it's better for your learning to develop your code from scratch, without consulting the workshop materials*.

If you use external code, in any way, remember to add proper references!

Lab Time is Fun Time

1. Q&A on W3.4 and its `Makefile`
2. GDB: follow the videos in W3.6, W3.7
3. Team Work: Do this week's exercises with your teammates
4. And FINISH assignment 1 with your team

Assignment 1

- Effective Team Collaboration is important.
- Use the Discussion Forum wisely!
- Ideally, finish your assignment by 5PM of you-know-which Thursday.

Notes:

- `valgrind` is excellent for finding potential problems
- `gdb` is great for tracing your code
- but don't rely too much on `gdb` debugging: *having a clear algorithm saves you days of debugging*