Spaced-Repetition Algorithm through a Linked-List

We're given the below 2 data-sets:

id	name	user_id
1	*'French'*	*1*

id	lang_id	original	translation	next
1	*1*	*'entraine toi'*	*'practice'*	*2*
2	*1*	*'bonjour'*	*'hello'*	*3*
3	*1*	*'maison'*	*'house'*	*4*
4	*1*	*'développeur'*	*'developer'*	*5*
5	*1*	*'traduire'*	*'translate'*	*6*
6	*1*	*'incroyable'*	*'amazing'*	*7*
7	*1*	*'chien'*	*'dog'*	*8*
8	*1*	*'chat'*	*'cat'*	*null*

Our first concern is to transform this data for insertion into a Linked-List:

Your boilerplate contains the `getLanguageWords` method, which returns the above data as an `Array` of `Objects` (*with each `Object` containing a row of data*).

Using an instance from the Linked-List covered in curriculum, we would:

```
1 const ll = new LinkedList();
```

We can also add more properties as needed to this instance, as they'll be useful later on (*remeber that boilerplate contains middleware which identifies the user, as well as their respective* `lang_id`):

```
ll.id = request.language.id;
ll.name = request.language.name;
ll.total_score = request.language.total_score;
```

To get the Linked-List's `head`, we would need the following SQL syntax:

```
SELECT
  word.id,
  word.language_id,
  word.original,
  word.translation,
  word.next,
  word.memory_value,
  word.correct_count,
  word.incorrect_count,
  language.total_score

  FROM word

  LEFT JOIN language

  ON language.head = word.id WHERE language.id =
  -- the lang_id
;
```

But we know that the middleware is already placing the language table on the user's `request`:

```
request.language.head;
```

After creating an instance from the Linked-List `class` and added any necessary properties, we need to `find` and insert the `head,` but first we need all words from the words table:

```
let words = LangService.getLanguageWords(request.language.id);

// then we match the head's id to the word's id:

let word = words.find(val => val.id === request.language.head);
```

Once that is found, we can insert it:

```
ll.insertFirst({
  id: word.id,
  original: word.original,
  translation: word.translation,
  memory_value: word.memory_value,
  correct_count: word.correct_count,
  incorrect_count: word.incorrect_count,
});
```

After the `head` is inserted, we'll continue matching the `next` with the word's `id`:

```
while (word.next) {
  word = words.find(w => w.id === word.next)
```

```
3
4   ll.insertLast({
5     id: word.id,
6     original: word.original,
7     translation: word.translation,
8     memory_value: word.memory_value,
9     correct_count: word.correct_count,
10    incorrect_count: word.incorrect_count,
11  })
12 };
```

At this point, we can start preparing our logic to deal with the `guess` coming in the `post` route:

```
1  const node = ll.head;
2  const answer = node.value.translation;
3  let isCorrect;
4
5  // if the incoming guess is correct:
6  if(req.body.guess === answer) {
7    isCorrect = true;
8    ll.head.value.mem_val *= 2;  // multiply by 2
9    ll.head.value.correct += 1;  // increment the word's correct count
10   ll.total_score += 1;         // increment the total score
11 }
12 // and if incorrect:
13 else {
14   isCorrect = false;
15   ll.head.value.mem_val = 1;   // reset back to 1
16   ll.head.value.incorrect += 1; // increment the word's incorrect count
17 }
```

Now that we sorted the main part of the logic, according to Spaced Repetition concept, the final step is to move that node down in the List, by the amount of memory value. to avoid convolution it's better to add a method in our class to take care of this specific task:

```
1  moveHeadBy(level) {
2    let head = this.head;             // first we create a copy of head
3    this.head = this.head.next;       // then detach it from the list
4    this.insertAt(level, head.value)  // use insertAt to move it downwards
5  }
```

After adding the above method to our LinkedList class, now we can call it from the instance to move the `head` downwards by the amount of memory value:

```
1  ll.moveHeadBy(ll.head.value.memory_value);
```

That's all the logic needed. The last part is to persist this Linked-List structure, by saving its nodes' values in database. We would need to pass this manipulated Linked-List instance (i.e.: `ll`) to the database service, so it takes care of saving it. Remember that we need to update two different tables. The `language` table and the `word` table. To help us to iterate over the list and to update each `node` in database, we might need to add this "*helper*" method within our Linked-List `class`:

```
1  forEach(cb) {               // when we call this method, we would also pass the callback
   instruction to update in db!
2    let node = this.head;     // start by creating a copy of the node at the top of our list
3    const arr = [];           // create an empty Array, so we can push each db update to it
```

```
  while (node) {            // iterate over all nodes
    arr.push(cb(node))
    node = node.next
  }
  return arr;               // return an Array with each update callback within
}
```

The last part would be to update the database's respective tables:

```
// since we need to update two tables in parallel,
// as well as updating several rows in the word table,
// through knex.transaction, the transacting method can
// be chained to any query and passed the Object we need to update

knex.transaction(trQry =>
  Promise.all([
    db('language')
      .transacting(trQry)
      .where('id', ll.id)
      .update({
        total_score: ll.total_score,
        head: ll.head.value.id,
      }),

    ...ll.forEach(node =>       // note the spread operator here, being used with the LL forEach inner method - without we'll lose a row
      db('word')
        .transacting(trQry)
        .where('id', node.value.id)
        .update({
          memory_value: node.value.memory_value,
          correct_count: node.value.correct_count,
          incorrect_count: node.value.incorrect_count,
          next: node.next ? node.next.value.id : null,
      })
    )
  ])
)
```

As you can see, you cannot copy and paste the code in your assignment, because it won't work. You need to adapt it to where it belongs!