COMP10002 Workshop Week 10

dynamic data structures

- BST insert: 5-minute warming up
- malloc/assert/free, Discuss Ex. lec07.4
- stack, dynamic arrays, of lec07.8
- linked lists, exercise lec07.9

- Exercises 4, 5, 6, and 7 in lec07.pdf: implement and test solutions to at least two of them.
- do Assignment 2

LMS requirements

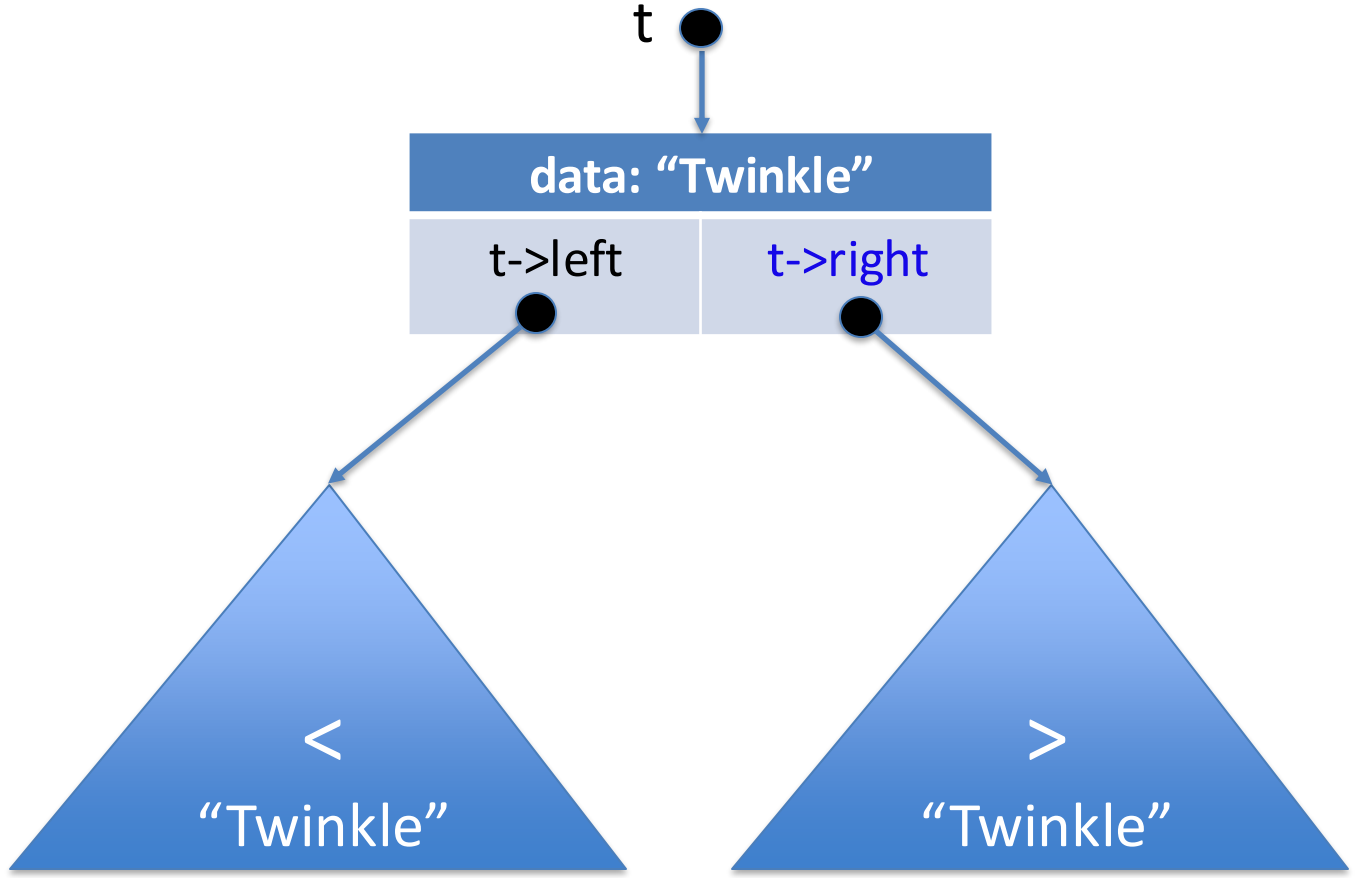
- Find another nursery rhyme that you like, and insert its words into a binary search tree
- Discuss Exercises 8 and 9 in the lec07 lecture slides.
- Then look at Exercises 4, 5, 6, and 7 in the lec07 lecture slides, and implement and test solutions to at least two of them.

Warm-Up: Binary Search Tree

```
// simplified version of  BST for the RHS
typedef struct node bst_t;

struct node {
  char *data;    // ptr to stored structure
  bst_t *left;   // left subtree of node
  bst_t *right;  // right subtree of node
};

// insert in BST order
bst_t *insert(bst_t *tree, char *value);


. . .
bst_t *t= NULL; // empty tree
t= insert(t, “Twinkle”);
t= insert(t, “twinkle”);
…
```

Insert the words into an initially empty BST:

Twinkle, twinkle, little star,
How I wonder what you are!
…



Dynamic Memory vs Automatic (aka Static) Memory

```
int *build_array() {
      int i, n= 10;
      double x= 2.5;
      int *new_arr;

      new_arr= malloc(40); /*1*/
      assert(new_arr != NULL);
      for (i=0; i<n; i++) {
      new_arr[i]= i;
      }              /*2*/
  return new_arr;
}
```

Automatic **Dynamic**

start of
i, n, x, new_arr

/*1*/ -----40 bytes (10 int)----

start of mem block ---------- 8000
(supposed address)

/*2*/ 0 1 2 3 4 5 6 7 8 9

end of
i, n, x, new_arr

```
... main ...
      int *A= build_array(); /*3/
      printf("%d\n", A[9]);
  ...
      free(A);/*4*/
      printf("%p\n", A);
      printf("%d\n", A[9]); // wrong!
```

/*3*/ 0 1 2 3 4 5 6 7 8 9

end of mem block at addr 8000 ---------- /*4*/ not available

Dynamic Memory Basic: malloc – assert – free

Dynamic memory is allocated during the runtime on programmer's request

- we request/allocate a memory chunk using malloc
- a call to malloc might fail, and the functions return NULL in this case ➔ use assert to check
- when no longer needed we must de-allocate the memory chunk using free.

✗	✗	✗	✓

int *p;
*p= 10;
...
<end program>
	int *p;
	p= malloc(sizeof(int));
	*p= 10;
	...
	<end program>
		int *p;
		p= malloc(sizeof(int));
		assert(p != NULL);
		*p= 10;
		...
		<end program>
			int *p;
			p= malloc(sizeof(int));
			assert(p != NULL);
			*p= 10;
			...
			free(p);
			<end program>

Memory before malloc	Memory after malloc, and just after *p= 10
p <undefined value>	p 10 free(p) will de-allocate this

Pointers and dynamic memory: related tools in a nutshell

Operation	How?
Pointer declaration for <type>	data_t *ptr; // data_t or any data type
Allocation, method 1	ptr= (data_t *) malloc(n * sizeof(data_t)); assert(ptr);
Allocation, method 2 (using automatic casting)	ptr= malloc(n * sizeof(*ptr)); assert(ptr);
Re-Allocation (auto copy and free the previous memory)	ptr= realloc(ptr, new_n * sizeof(*ptr)); assert(ptr);
Deallocation	free(ptr); // rule: one malloc – one free
Notes: sizeof(int)==sizeof(float)==4 sizeof(double)==8 sizeof(char)==1 sizeof(any-pointer) == 8	

Why Dynamic Memory?

Reasons include:

1. To keep data alive beyond the local scope of a function (i.e., data persists after the function returns). Example: lec07.Ex4
2. To create arrays of any size that can also grow or shrink as needed. Example: Dynamic Arrays
3. To create flexible data structures like linked lists or trees, where size and connections can change. Example: Linked Lists

lec07.Ex4: How

Exercise 4

Write a function

char *string_dupe(char *s)

that creates a copy of the string `s` and returns a pointer to it (same as function strdup)

```
char *string_dupe(char *s) {
   //  Notes: do not use library function strdup
   //         but you can use strcpy
   ... t;     //  first declare t,    HOW?

   //  then make target t be identical to source  s,          HOW?
 ...


 return t;
}
```

```
// example of using string_dupe
... main(...) {
        char *string= “example”;
        char *s= string_dupe(string);
        printf(“duplicated s= %s\n”, s);
  ...
```

lec07.Ex4: Is this solution correct?

Exercise 4

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

```
char *string_dupe(char *s) {
   // first declare t
   char t[MAXLEN+1]; // supposing MAXLEN is well defined

   // then make target t be identical to source s
   strcpy(t,s);


   return t;
}
```

```
// example of using string_dupe
... main(...) {
    char *string= "a hoax!";
    char *s= string_dupe(string);
    printf("duplicated s= %s\n", s);
    ...
```

Check your answer: lec07.Ex4 char *string_dupe(char *s)

```
char *string_dupe(char *s) {
      // char t[MAXLEN]; → wrong because t will be automatically deleted at the end of this function
      char *t= malloc( (strlen(s)+1) * sizeof(char) );
            //  memory for t dynamically created, it survives until it is free-ed!
      strcpy(t,s);
      return t;
}

... main ... {
  char *s= “Ta daa!”, *dup;
  dup= string_dupe(s);
  ...
  free(dup);        // the memory malloc-ed above for t is free-ed here
     return 0;
}
```

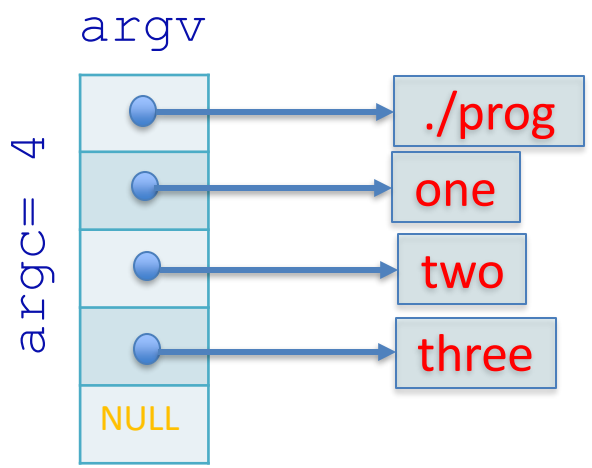
argc, argv and Exercise 5 – HOWTO Discussion

Review: example of argc, argv

./prog compiled from

int main(int argc, char *argv[])...

```
[user@sahara ~]$ e two three
```



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.

```
char **string_set_dupe(char **S) {
  ... duplicated;  // HOW TO declare duplicated?

    //  then make it be identical to S,          HOW?


    return  duplicated;
}

//   example of using in main():
char **dupl_argv= string_set_dupe(argv);
// now dupl_argv is identical to argv[]
```

“Static Arrays” versus Dynamic Arrays

Dynamic array has flexibility: it can be re-sized when needed, it can “survive” outside a function
Note 1: static arrays are pointer constant, dynamic arrays are pointer variables
Note 2: for dynamic arrays, tuple (`A, size, n`) is conveniently packed into a struct

Static	Dynamic

`#define SIZE 100`
`int B[SIZE];`
 	
`int n= 0;`
`while (scanf(“%d”,&x)==1) {`
`  if (n==SIZE) {`
`  // problem:`
`  // B[] runs out of space`
`  break;`
`  }`
`  B[n++]= x;`
`}`
`…`
	`int size= 8; // 8 or some small value`
	`int *A;  // will make A an array with capacity size`
	`A= malloc(size*sizeof(*A));`
	`assert(A);`
	`int n= 0;`
	`while (scanf(“%d”,&x)==1) {`
	`  if (n==size) {`
	`  ???`
	
	`  }`
	`  A[n++]= x;`
	`}`
	`…`
	`free(A);  // only when A no longer needed`

“Static Arrays” versus Dynamic Arrays

Dynamic array has flexibility: it can be re-sized when needed.
Note 1: static arrays are pointer constant, dynamic arrays are pointer variables
Note 2: for dynamic arrays, tuple (`A, size, n`) is conveniently packed into a struct

Static	Dynamic

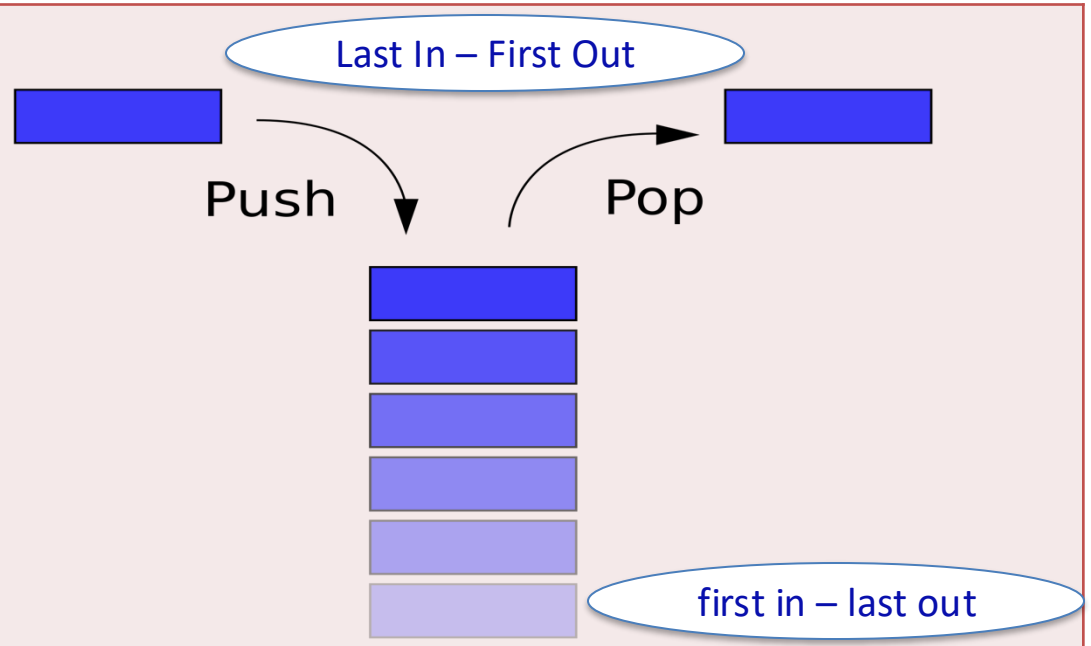
`#define SIZE 100`
`int B[SIZE];`
 	
`int n= 0;`
`while (scanf(“%d”,&x)==1) {`
`  if (n==SIZE) {`
`  // problem:`
`  // a[] runs out of space`
`  break;`
`  }`
`  B[n++]= x;`
`}`
`…`
	`int size= 8; // 8 or some small value`
	`int *A;`
	`A= malloc(size*sizeof(*A));`
	`assert(A);`
	`int n= 0;`
	`while (scanf(“%d”,&x)==1) {`
	`  if (n==size) {`
	`  size *= 2;`
	`  A= realloc(A, size*sizeof(*A));`
	`  assert(A);`
	`  }`
	`  A[n++]= x;`
	`}`
	`…`
	`free(A);  // only when A no longer needed`

lec07.Ex8: 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.

Stack (LIFO)

http://www.123rf.com/stock-photo/tyre.html



https://simple.wikipedia.org/wiki/Stack_(data_structure)

Stack Operations

`push(x)`: add element `x` into (the top of) stack

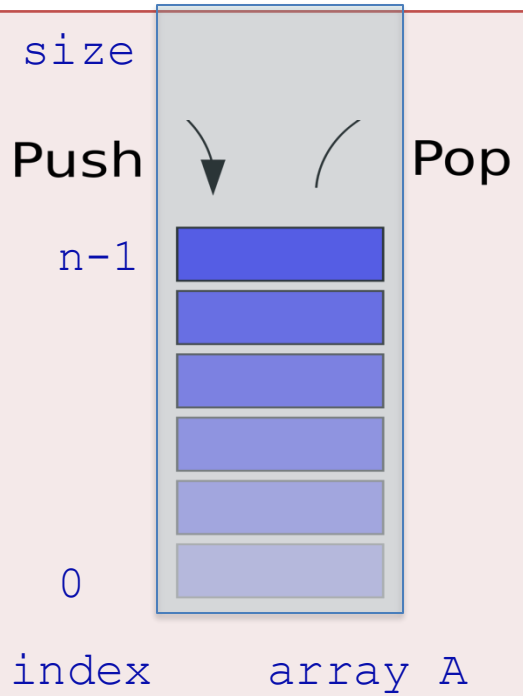
`pop()`: remove an element from (the top of) stack

`create()`: create a new, empty stack

`isEmpty()`: check if stack is empty, or

Stack implementation using array

http://www.123rf.com/stock-photo/tyre.html



https://simple.wikipedia.org/wiki/Stack_(data_structure)

Stack Operations

push(x) : add element x into (the top of) stack

```
A[n++]= x;
// supposing no problem with array size
```

pop() : remove an element from (the top of) stack

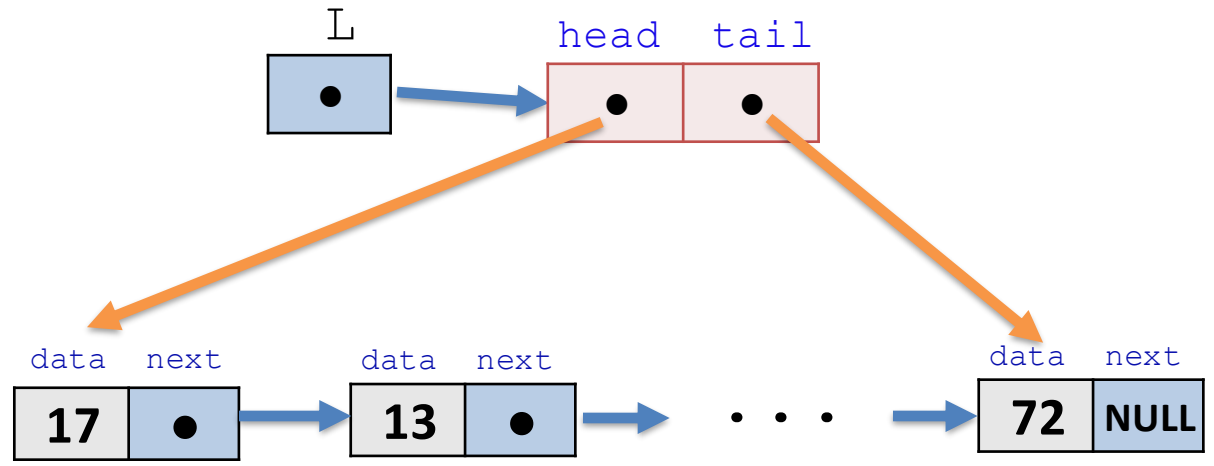
```
return A[--n];
```

push(x) is problematic: *needs a dynamic array*

Linked List: as in `listops` (Ex10.x1)

```
typedef struct node node_t;
struct node {
        data_t data;
        node_t *next;
};
typedef struct {
        node_t *head;
        node_t *foot;
} list_t;

list_t *make_empty_list(void);
int is_empty_list(list_t *list);
void free_list(list_t *list);
list_t *insert_at_head(list_t *list, data_t value);
list_t *insert_at_foot(list_t *list, data_t value);
...

int main(...) {
  list_t *L= make_empty_list();
  ...
  free_list(L);
}
```

need to maintain both head and foot when processing



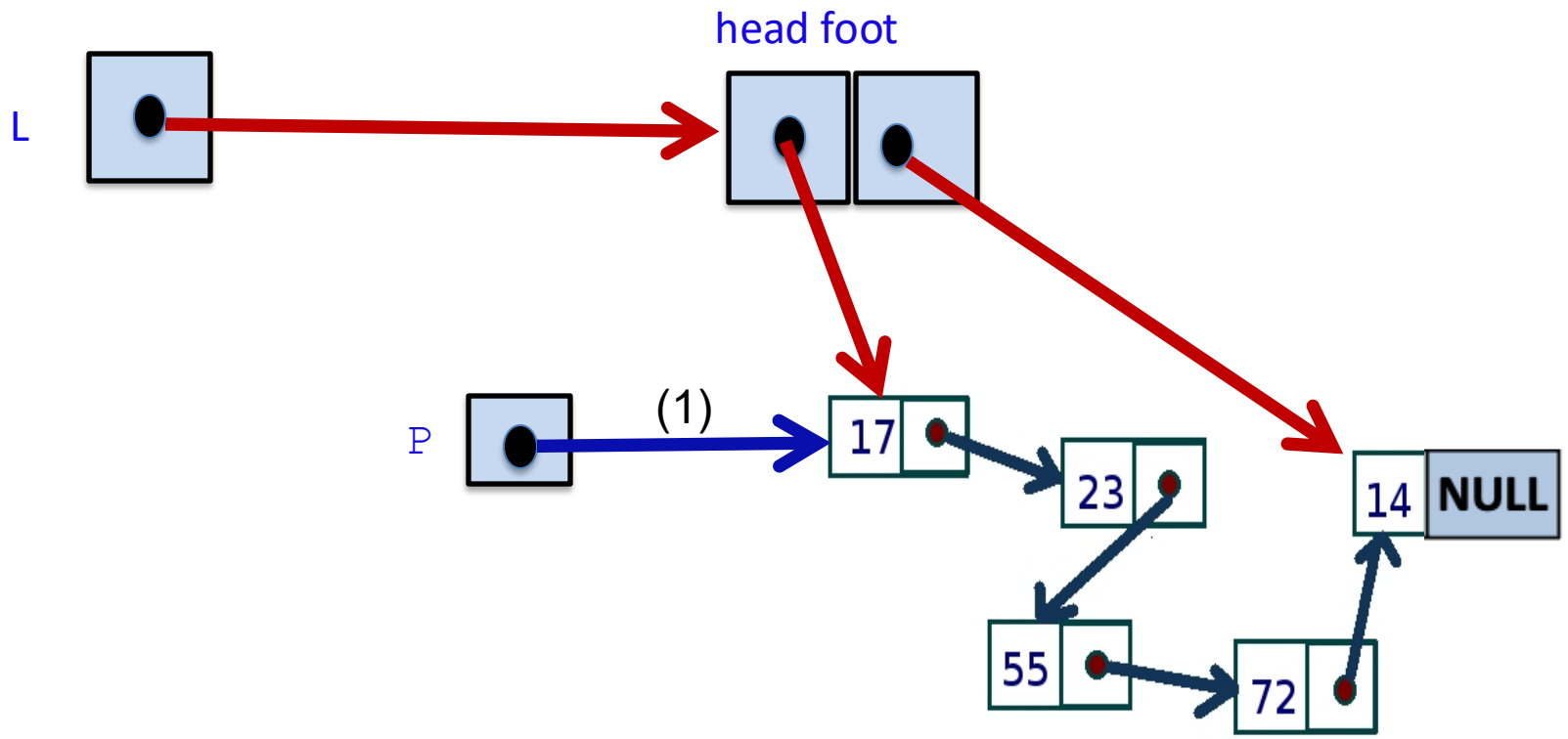
Quiick Questions: We have a sequence of integers such as 17, 13, ..., 72. How to:

- *build a linked list for these integers in the input order?*
- *build a linked list for these integers in the reversed input order?*
- *build a linked list for these integers in the increasing order?*

Linked List Example: search for data_t x

1. start with first element
2. loop while exists
• 2a check
• 2b move to the next element
 	
here P is NULL
	`node_t *P= L->head;`
	`while (P != NULL ) {`
	`  if (P->data == x) return P;`
	`  P= P->next;`
	`}`
	`return NULL;`

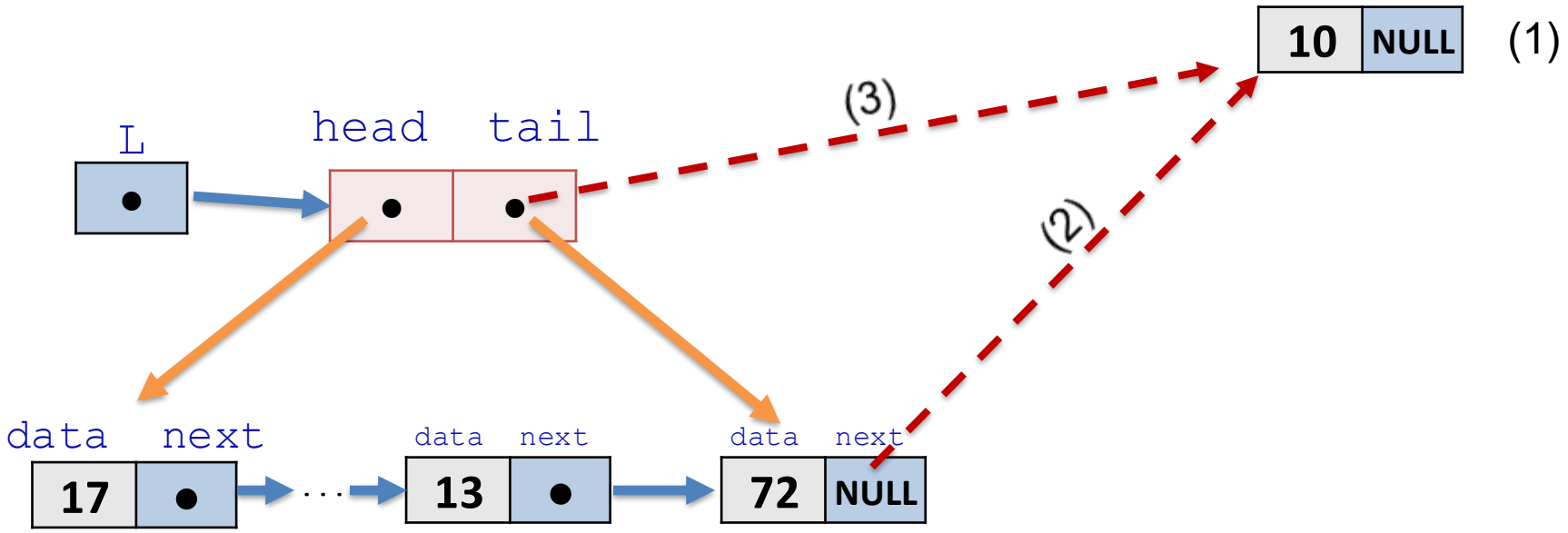
Class Exercise: print out all data of L



Example `insert_at_foot` : insert node with `data_t 10` to `list_t *L` – *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;`

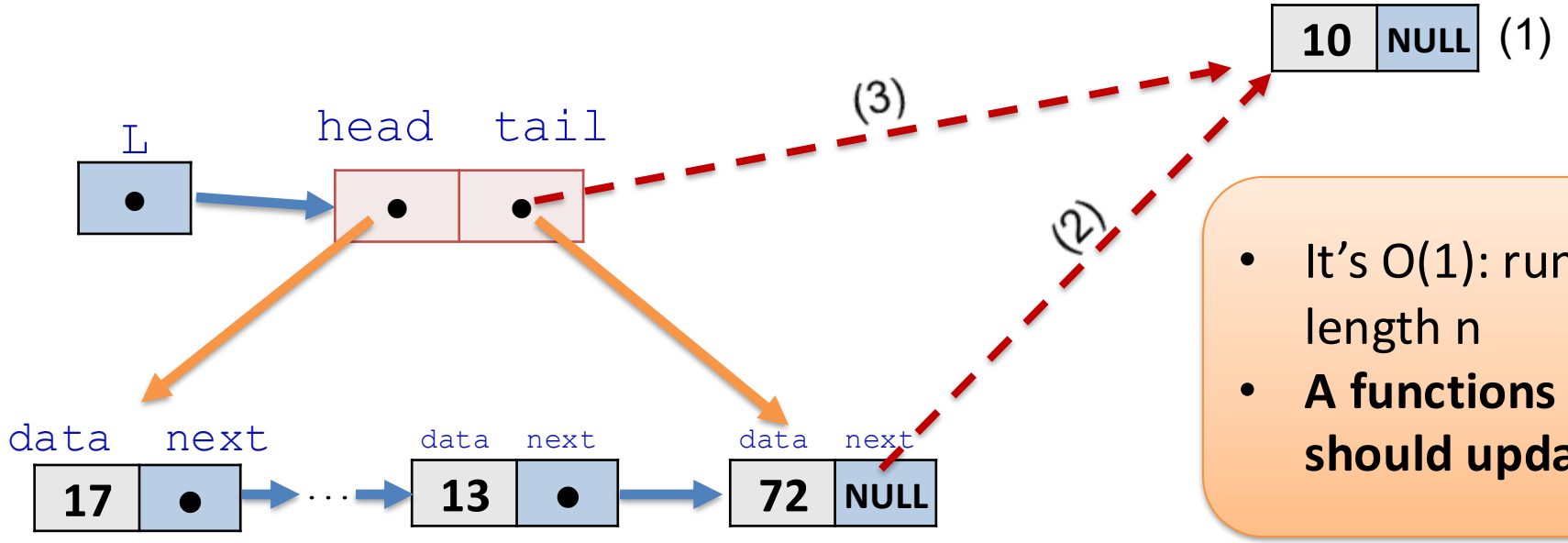
2. Link the node to the chain	`L->tail->next= new;`
3. Repair `tail`	`L->tail= new;`



insert_at_foot : insert node with data_t 10 to list_t *L – a correct version

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): running time not depending on length n
- **A functions that changes a linked list should update both `head` and `tail`**

Discussion: Ex10.x2 Linked Lists

Know how to use listops.c and listops.h

Task 2: add to the library listops function

list

list_t *insert_in_order(list_t *list, data_t value, int (*cmp)(void*, void*))

that inserts a value into a list and keeps the items in the list in the ascending order after the insertion.

Discuss:

- Where to insert?
- How to insert?
- Special cases?

Note

- list->tail is only for the convenience of insert_at_foot
- If tail is not needed list_t can be better declared as:

```
typedef node_t list_t
```

FEIT Small Class Surveys

The surveys were sent to you from "no-reply@unimelb.edu.au" with the subject "**FEIT Subject Surveys**".

Please complete them if you haven’t done so.

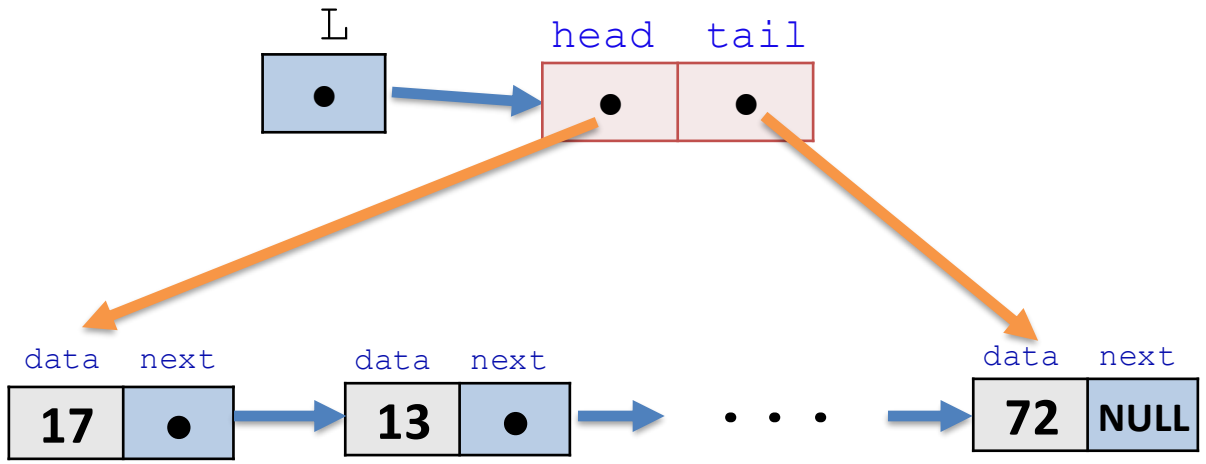
Arrays vs Linked Lists: random access and sequential access to k-th element

ARRAY

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

LINKED LIST

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



Comparing arrays and linked lists. Which one is more efficient for:

- accessing the k-th element?
- searching for a value
- inserting a new element following a certain element?
- delete the element following a certain element?

Lab Time: Finish Ed Ex10.x1, 10.x2

Exercise 4: Write a function

```
char *string_dupe(char *s)
```

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?)

Exercise 8: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.Test your code: using stack, input at most 1000 integers and print them in reverse order.

Exercise 9: Suppose that insertions and extractions are required at both head and foot. Add a second pointer to each node (to make a doubly-linked list) and then rework the previous functions.

Assignment 2

- Understand spec and how to approach
- Decide: linked lists or dynamic arrays
- Questions?
- Aim to finish one more stage today!
- Have a good plan for finishing the assignment!

New items in the marking rubric:

- avoidance of typedefs, -0.5;
- avoidance of structs, -1.0;
- avoidance of pointers to structs, -0.5;
- memory management issue (minor), -0.5;
- memory management issue (major), -1.0;