COMP10002 Workshop Week 11

First Hour	• BST insert: 5-minute warming up • malloc/assert/free, Ex. lec07.4, lec07.5 • linked lists exercise lec07.9 • (time permitted) stack, exercise lec07.8
Lab Time	**MAIN ROOM** • Dynamic Array and its possible application in A2 • A2: Q&A **BREAK-OUT ROOMS** • Work on A2, and/or • Exercises 5, 6, 7 in lec07
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. • Make sure that you know what is expected of you in regard to Assignment 2, and make a solid start
Next Week	Assignment 2 due **6PM Fri 15 OCT**

Warm-Up: 12345 Once I caught A fish Alive

Insert the words into an initially empty BST:

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

`malloc/free/assert`: What, Why, When?

malloc= WHAT? Exercise: Write code fragment that

- use just a pointer `p` (ie. *not using another variable*), make `*p` be `10` and print out `*p`

```
int *p;
...
*p= 10;
printf("*p= %d\n", *p);
```

Memo: library functions malloc() and free()

```
void  *malloc(size_t n);
```

(dynamically) allocate a chunk of n bytes, and return the address of that chunk. It returns `NULL` if failed. Example:

```
int *a;
a= (int *) malloc( sizeof(int) );
a= malloc( sizeof(int) );

a= malloc( sizeof(*a ) );
```

--------- Need to check if `malloc` sucessful ----------

```
assert(a != NULL);   // OR
assert(a);
```

Note: memory created by malloc is not auto-freed!

"`assert(a)`" is equivalent to:

```
if (!a) {
   exit(EXIT_FAILURE);
}
```

```
void  free(void *p);
```

free the memory chunk that p points to. Example:

```
free(a);
a= NULL;
```

Exercise

Write a code fragment that build an array of 10 strings using scanf. Example input:

```
string0 string1
two
three   four
5 6 7 8 string9
```

then print these strings. Suppose that the max length of strings is 99. Change the code to minimize the wasted memory of strings A[i].

```
code:
#define SIZE 10
#define MAX 99
char A[SIZE][MAX+1];
int n=0;

for (; n<10; n++) {
  scanf(" %s", A[n]);
}
...
```

lec07.E4: Is the 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.

```
#define MAXLEN 100
char *string_dupe(char *s) {
    char t[MAXLEN];
    strcpy(t,s);
    return t;
}

… main(...) {
char *s= string_dupe("A hoax!");
```

Is the 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.

```
char *string_dupe(char *s) {
   char t[MAXLEN];
   strcpy(t,s);
   return t;
}
```

➔ Incorrect: the memory for t disappears at the end of string_dupe and cannot be used in the calling function like in:

```
char *s= string_dupe("A hoax!");
```

Make the code 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.

```
#define MAXLEN 100
char *string_dupe(char *s) {
    char t[MAXLEN];
    strcpy(t,s);
    return t;
}

… main(...) {
char *s= string_dupe(“A hoax!”);
```

Check your solution: Ex 4

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;
}
```

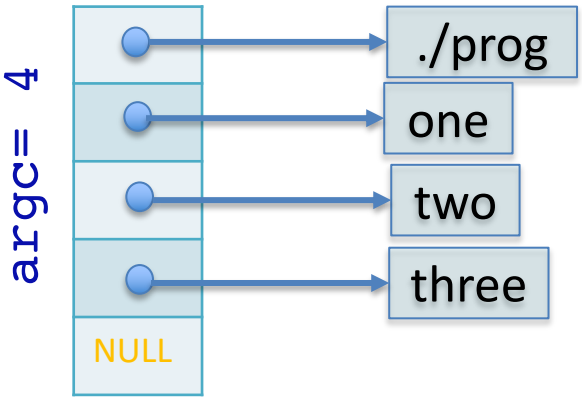
argc, argv and Exercise 5

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.

// ./prog one two three

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




```
char **string_set_dupe(char **S) {


}
```

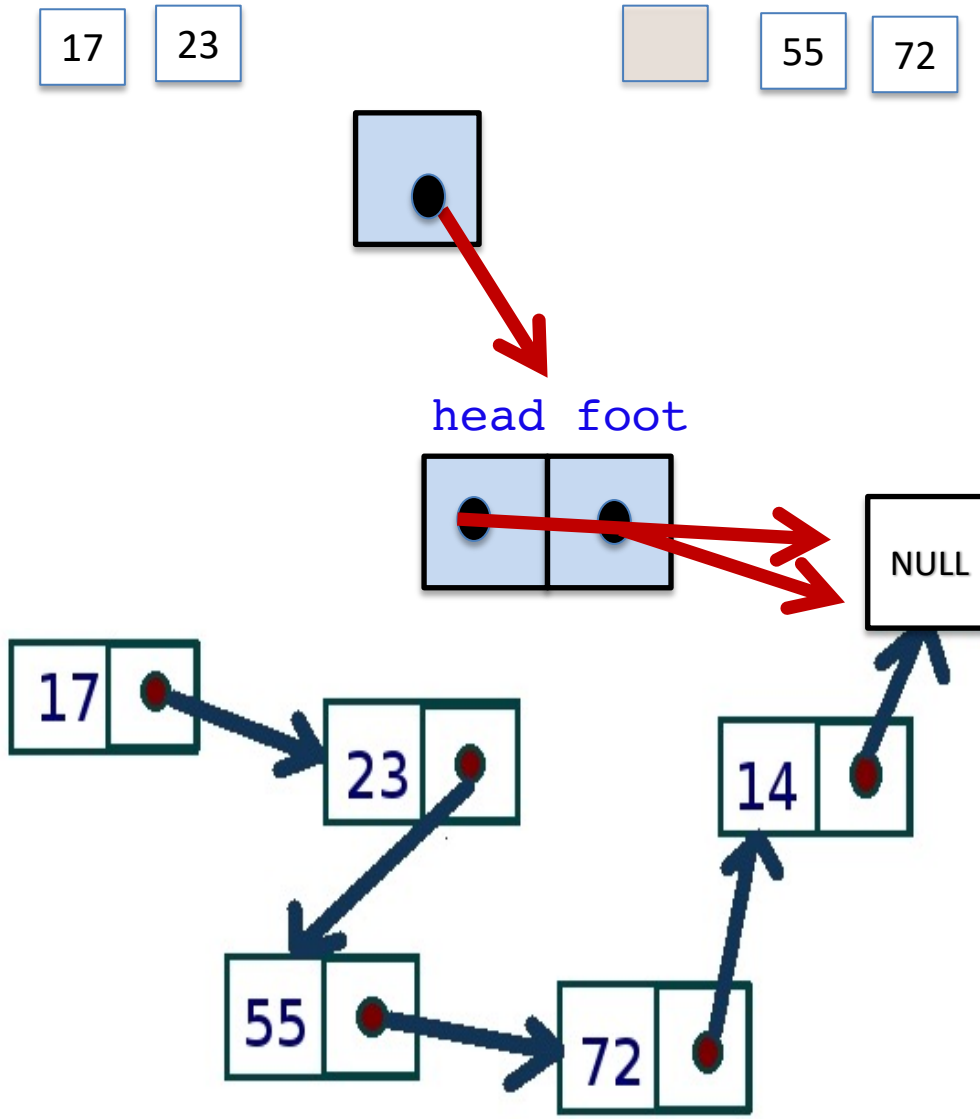
Linke Lists (using Alistair's `listops.c`)

Define Linked List: as in Alistair's `listops.c`

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

int main(…) {
  list_t *L= make_empty_list();
  L= insert_at_foot(L, 55);
  L= insert_at_foot(L, 72);
  L= insert_at_foot(L, 14);

  L= insert_at_head(L, 23);
  L= insert_at_head(L, 17);
  ...
  free_list(L);
}
```

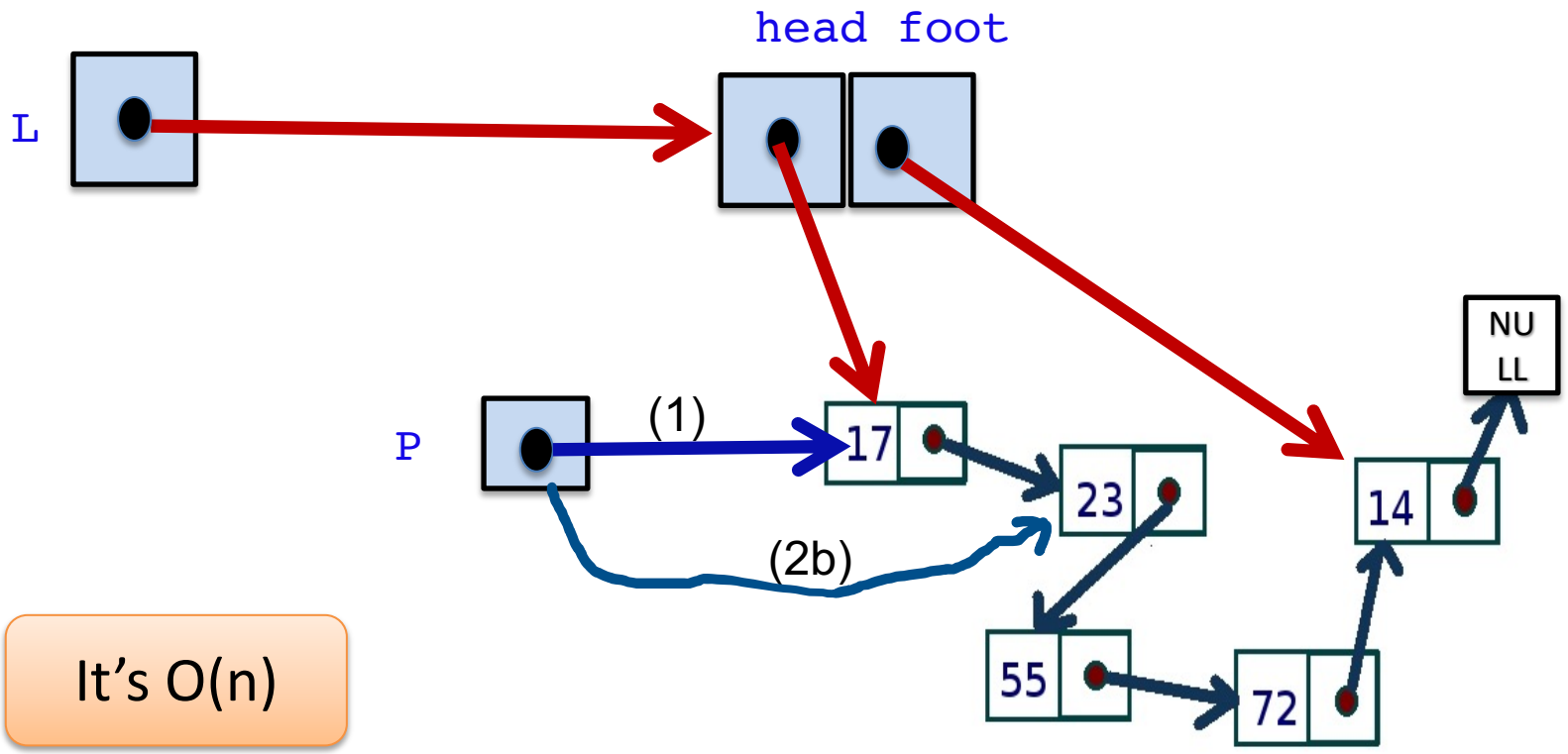


Linked List Example: search for data x

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) {  // do smt with P->data
  if (P->data == x) return P;
  P= P->next;
}
return NULL;
```

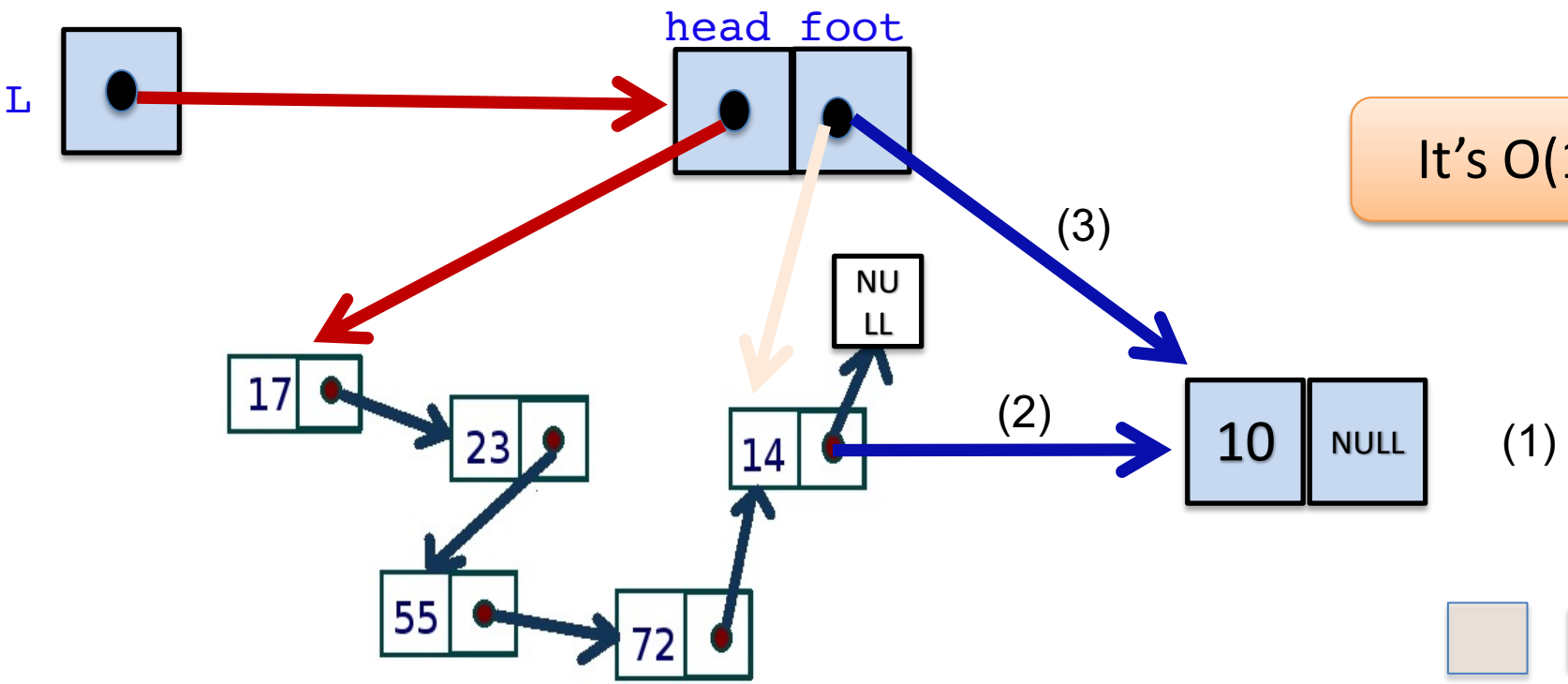


It's O(n)

Linked List Example: insert_at_foot : Insert node with data 10 at foot

1. create new node and set data
2. Link the node to the chain
3. Repair the foot
4. if the list is empty before inserting

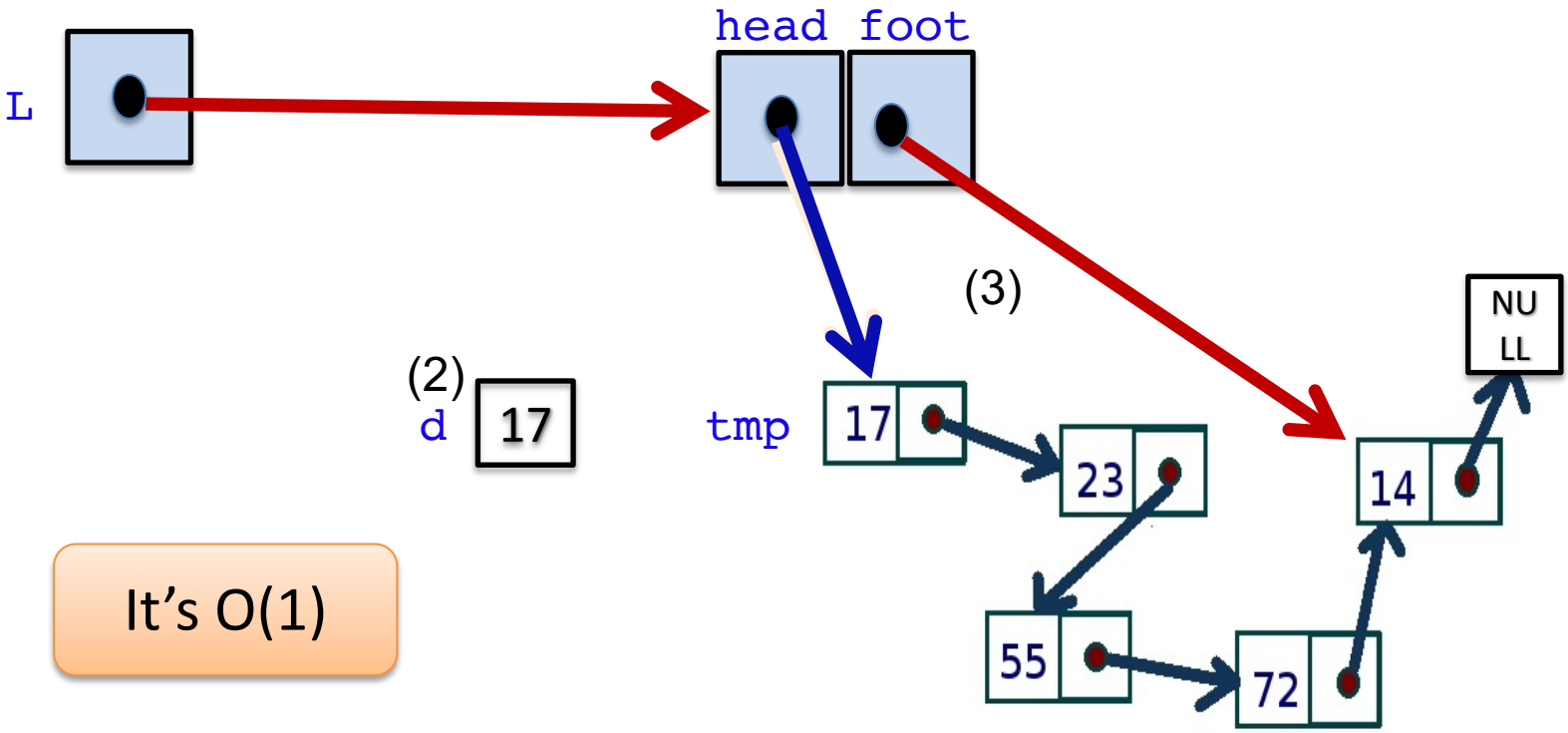
```
node_t *new= mymalloc(zizeof(*new));
new->data= 10;    // here, data_t is int
new->next= NULL;
if (L->foot) {
  L->foot->next = new;
  L->foot= new;
} else
  L->foot= L->head= new;
```



It's O(1)

Linked List Example: remove_head

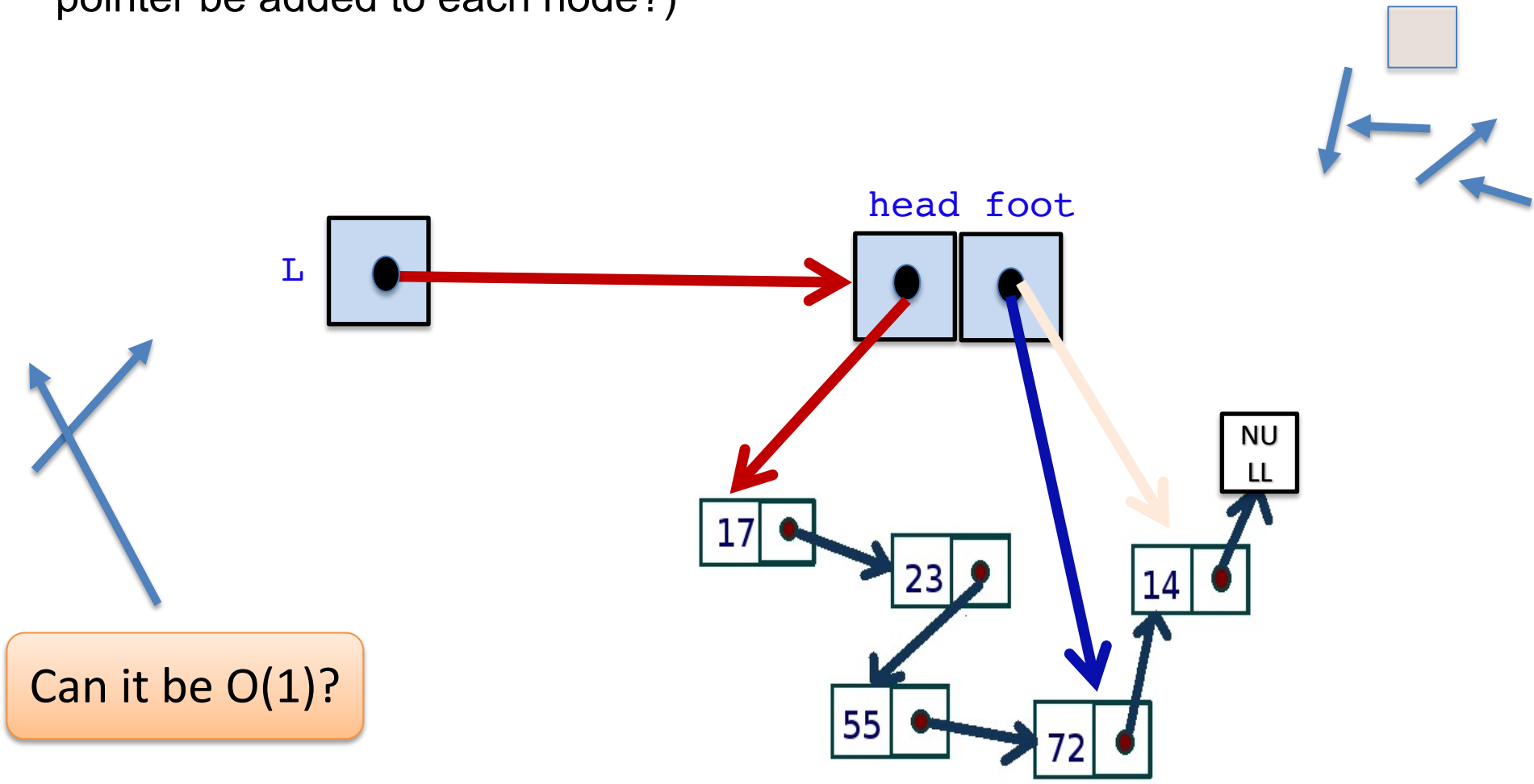
1. safeguard	`assert(L->head);`
	`node_t *tmp= L->head;`
2. keep the data	`data_t d= tmp->data;`
3. Repair the head	`L->head= L->head->next;`
	`free(tmp);`
4. Repair the foot if needed	`if (!L->head) L->foot= NULL;`
5. Return data	`return d;`



It's O(1)

Exercise 9 (in the context of linked lists): remove_foot

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?)



Can it be O(1)?

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:

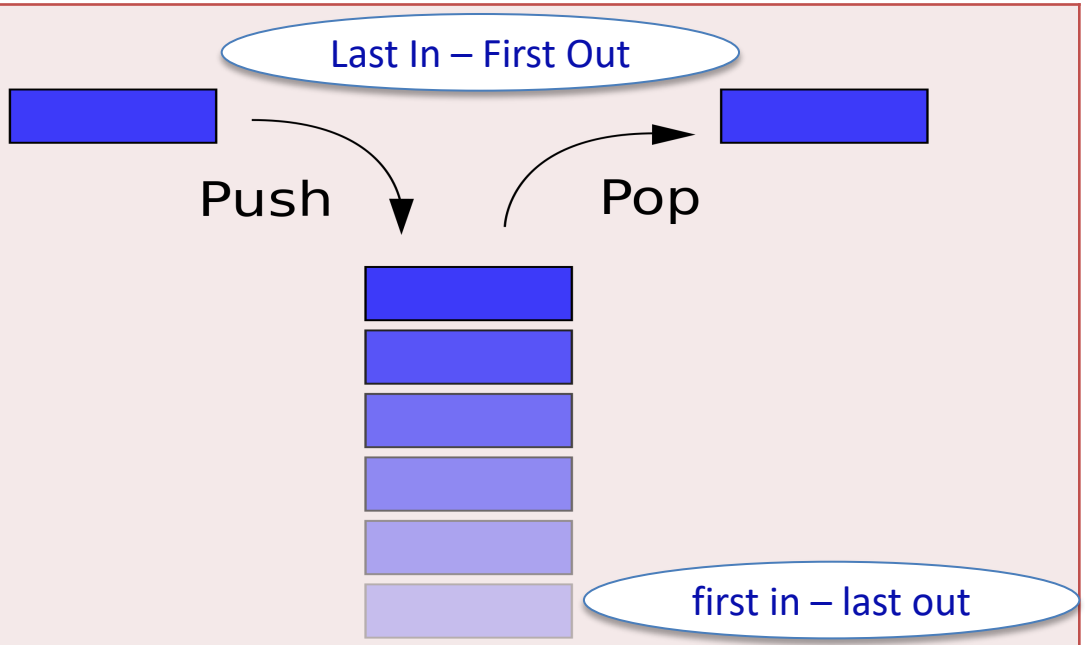
- Test it by adding 10 integers:

  `0 1 2 … 9`

  to a stack, then print them (in reverse order).

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

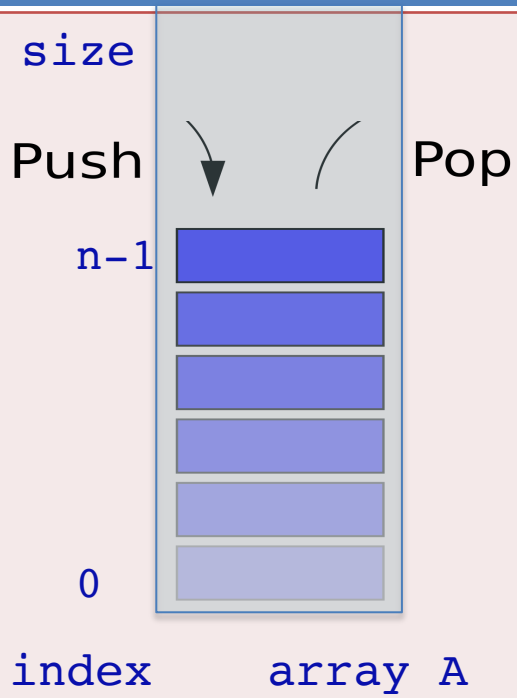
`pop()`: remove an element from (the top of) stack

`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;

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

return A[--n];

Lab Time: Exercises 4-8 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?)

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.

Lab Time

Break-out Room will be opened shortly. Your choice to join one of the:

MAIN ROOM

- Dynamic Array and its application to A2
- A2: Q&A

BREAK-OUT ROOMS

- Work on A2, and/or
- Finish Exercise 9 in lec07
- Finish Exercises 4, 5, 6, 7 in lec07

Memo: Tools for resizing (memory re-allocation)

```
p= realloc( p, n);
```

resizes the memory allocated to the old `p` by performing 4 actions:

- allocate a new chunk of `n` bytes,
- copy the chunk pointed by the old `p` to that new chunk,
- free the memory chunk pointed to by the old `p`, and
- returns the address of that new chunk.

Note: similar to `malloc`, `realloc` might fail.

Example of use:

```
int *p;
p= malloc(10 * sizeof(*p));
assert(p);
...
p= realloc(p, 20 *sizeof(*p));
assert(p);
...
free(p);
```

Arrays - Static & Dynamic

Dynamic array has flexibility: it can be re-sized when needed.

Static	Dynamic

`#define SIZE 100`
`int a[SIZE];`
 	
`int n= 0;`
`while (scanf("%d",&x)==1) {`
`  if (n==SIZE) {`
`  // problem:`
`  // a[] runs out of space`
`  break;`
`  }`
`  a[n++]= x;`
`}`
`...`
	`int size= 8; // 8 or some small value`
	`int *b;`
	`b= malloc(size*sizeof(*b));`
	`assert(b);`
	`int n= 0;`
	`while (scanf("%d",&x)==1) {`
	`  if (n==size) {`
	`  size *= 2;`
	`  b= realloc(b, size*sizeof(*b));`
	`  assert(b);`
	`  }`
	`  b[n++]= x;`
	`}`
	`...`
	`free(b);`

Dynamic Array Example: Keeping Track of check boards

A2: Q&A

Additional Slides (for self-review)

Arrays: Static Arrays and Dynamic Arrays

Static Arrays. Scenario 1: known an upper bound of size

A task: input an array of int, output the array's elements in ascending order.

```
#define SIZE 1000

int main(int argc, char *argv[]) {
  int A[SIZE], n=0;

  for (n=0; n<MAX && scanf("%d", A+n)==1; n++);

  sort(A, n);
  print_array(A,n);
  return 0;
}
```

What if `SIZE` is unknown in advance? (ie. not available at compiler time?)

Automatic memory allocation (by compilers)

```
#define SIZE 1000

int main(int argc, char *argv[]) {
  int A[SIZE], n=0; // mem auto-allocated by compiler
  for (n=0; n<SIZE && scanf("%d", A+n)==1;
n++);
... // sorting & something else ...
  // A and n are  auto-released (auto-freed) by compiler
}
```

Memo 1: An array is controlled by three named objects:

- array name **A**
- current number of elements **n**
- capacity **SIZE**

Memo 2: memory for a variable is auto-allocated at the start of its scope, and auto-released at the end of its scope.

Dynamic Arrays. Scenario 2: If SIZE in unknown, and n is known at runtime:

```
int main(int argc, char *argv[]) {
   int *A, n=0; // n and pointer A auto-allocated by compiler

   scanf("%d", &n);

   A= <a memory chunk for n integers>
   // ie we demand (the system to allocate) a chunk of n integer cells
   // and we store the address of the chunk into A
   // → A would be equivalent to an array of n integers

   for (i=0; i<n && scanf("%d", A+i)==1; i++);
   …
   // Make sure that the memory chunk pointed to by A is freed

   // A and n are auto-freed by compiler
}
```

Scenario 2: SIZE in unknown, n is known at runtime:

```
int main(int argc, char *argv[]) {
   int *A, n=0; // n and pointer A auto-allocated by compiler
   scanf("%d", &n);



   A= malloc( n*sizeof(*A) );
   // that is, A= a memory chunk for n integers
   //       that the system allocated for the function call malloc

   assert(A); // terminates if malloc failed

   for (i=0; i<n && scanf("%d", A+i)==1; i++);
   …
   free (A); // Free the memory chunk associated with A
   // A and n are auto-freed by compiler
}
```

In C99 we can also use:
int A[n];

Scenario 2: SIZE in unknown, n is known at runtime:

```
int main(int argc, char *argv[]) {
  int *A, n=0; // n and pointer A auto-allocated by compiler
  scanf("%d", &n);

  A= malloc( n*sizeof(*A) );
  // that is, A= a memory chunk for n integers
  //       that the system ***allocated*** for the function call malloc

  assert(A); // terminates if malloc failed
  for (i=0; i<n && scanf("%d", A+i)==1; i++);
  …
  free (A); // Free the memory chunk associated with A
  // A and n are auto-freed by compiler
}
```

"assert(A)" is equivalent to:

```
if (!A) {
   exit(EXIT_FAILURE);
}
```

Scenario 3: what if n is not available at the loop start?

```
#define INIT_SIZE 8
   int size=INIT_SIZE              // estimated capacity
   // declares an array as a triple  <A, n, size>
   int *A=NULL, n=0;
   A= malloc( size * sizeof(*A) );
   assert(A);

   for (i=0; scanf("%d", &x)==1; i++) {
      ???
      A[i]= x;  // error if i==size
   }
   ...
```

n is not available in advance?

```
#define INIT_SIZE 8

  // declares an array as a triple <A, n, size>
  int *A=NULL, size=INIT_SIZE, n=0;
  A= malloc( size * sizeof(*A) );
  assert(A);

  for (i=0; scanf("%d", &x)==1; i++) {
    if (i==size) {
      // solution: free(A) then re-malloc A with a bigger memory chunk
    }
    A[i]= x;
  }
  ...
```

n is not available in advance?

```
#define INIT_SIZE 8

  // declares an array as a triple <A, n, size>
  int *A=NULL, size=INIT_SIZE, n=0;
  A= malloc( size * sizeof(*A) );
  assert(A);

  for (i=0; scanf("%d", &x)==1; i++) {
    if (i==size) {
      size *= 2;   // makes size bigger
      // ?: free(A) then re-malloc A with a bigger memory chunk
      //    of course, with copying the old to the new chunk
    }
    A[i]= x;
  }
  ...
```

Scenario 3: If n is not available in advance:

```
#define INIT_SIZE 8
   // declares an array as a triple  <A, n, size>
   int *A=NULL, size=INIT_SIZE, n=0;
   A= malloc( size * sizeof(*A) );
   assert(A);

   for (i=0; scanf("%d", &x)==1; i++) {
     if (i==size) {
       size *= 2;   // makes size bigger
       A= realloc(A, size*sizeof(*A));
                    // allocate a new chunk of new size
                    // automatic copy and free the old chunk
       assert(A);
     }
     A[i]= x;
   }
   ...
   free(A);       // one malloc – one free, no worry about realloc
```