COMP10002 Workshop Week 10

dynamic data structures

- BST insert: 5-minute warming up
- malloc/assert/free, Discuss Ex. lec07.4, lec07.5
- linked lists, exercise lec07.9
- (time permitted) stack, exercise lec07.8

- Exercises 4, 5, 6, and 7 in lec07.pdf: implement and test solutions to at least two of them.
- Make sure that you know what is expected for Assignment 2, and make a solid start. Note: due **6PM Fri 14 OCT**

Warm-Up: 12345 Once I caught A fish Alive

```
typedef struct node node_t;

struct node {
  void *data;   // ptr to stored structure
  node_t *left; // left subtree of node
  node_t *rght; // right subtree of node };

typedef struct {
  node_t *root; // root node of the tree
  int (*cmp)(void*,void*);
} tree_t;

// creates & returns an empty tree
tree_t *make_empty_tree(int func(void*,void*));

// insert in BST order
tree_t *insert(tree_t *tree, void *value);
```

Insert the words into an initially empty BST:

Twinkle, twinkle, little star,
How I wonder what you are!
Up above the world so high,
Like a diamond in the sky.

`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*. Returns *the address* of that chunk, or `NULL` if failed. Example (3 equivalent `malloc`)

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

or:

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

`calloc` is the same as `malloc`, but it also initialize all memory to zero.
Quite useful sometimes.

-------- Need to check if `malloc` successful ---------

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

or:

```
assert(a);
```

`assert(a);` is equivalent to:

```
if (a==NULL) {
   exit(EXIT_FAILURE);
}
```

Note: memory created by malloc is not auto-freed!
Need to free the memory chunks that were malloc-ed.
Example:

```
free(a);
```

Rule:
one `malloc` needs one `free`
need to `free` *the address* that `malloc` returned

lec07.E4: 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`)

Discussion:

- What the difference between `strcpy` and `strdupe/strdup` ?
- How to fill in the missing parts?

```
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.E4: 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) {
  // Notes: do not use library function strdup
  //        but you can use strcp
  char dupl[MAXLEN+1]; // supposing MAXLEN is well defined

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


  return dupl;
}
```

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

Check your answer: 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.

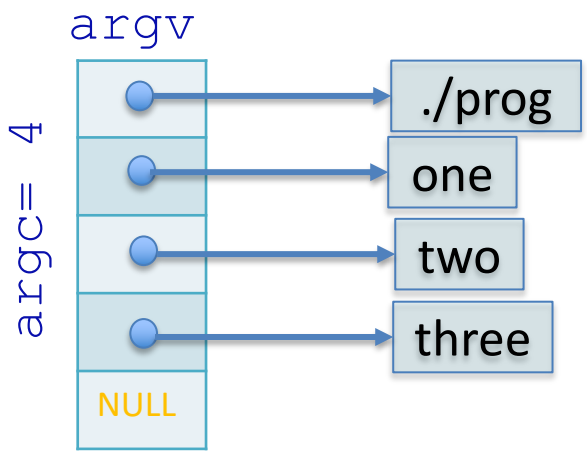
```
char *string_dupe(char *s) {
    // char t[MAXLEN]; → wrong because t will be automatically deleted at the end of this function
    char *t= malloc( (strlen(s)+1) * sizeof(char) );
                        // t dynamically created here, it survives until the associated memory is free-ed!
    strcpy(t,s);
    return t;
}
... main ... {
  char *s= string_dupe("Ta daa");
  ...
  free(s);              // the memory malloc-ed above for t is free-ed here
  return 0;
}
```

argc, argv and Exercise 5 - Discussion

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) {
  ... duplicated;   // first declare duplicated,   HOW?

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



  return duplicated;
}
```

“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 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 *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

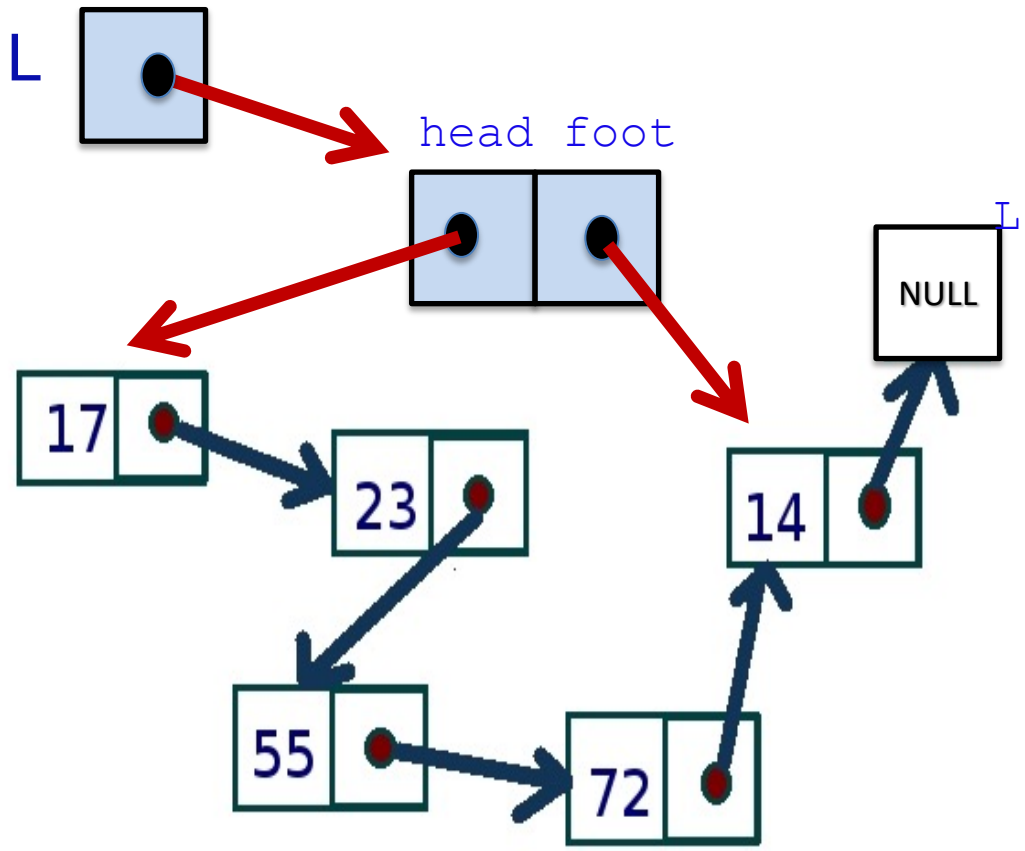
Dynamic array has flexibility: it can be re-sized when needed.
Note 1: static arrays are pointer constant, dynamic arrays are pointer variables
Note 2: for dynamic arrays, tuple (`A, size, n`) is conveniently packed into a struct

Static	Dynamic

`#define SIZE 100`
`int 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 *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`

Linked List version 1: as in `listops.c`

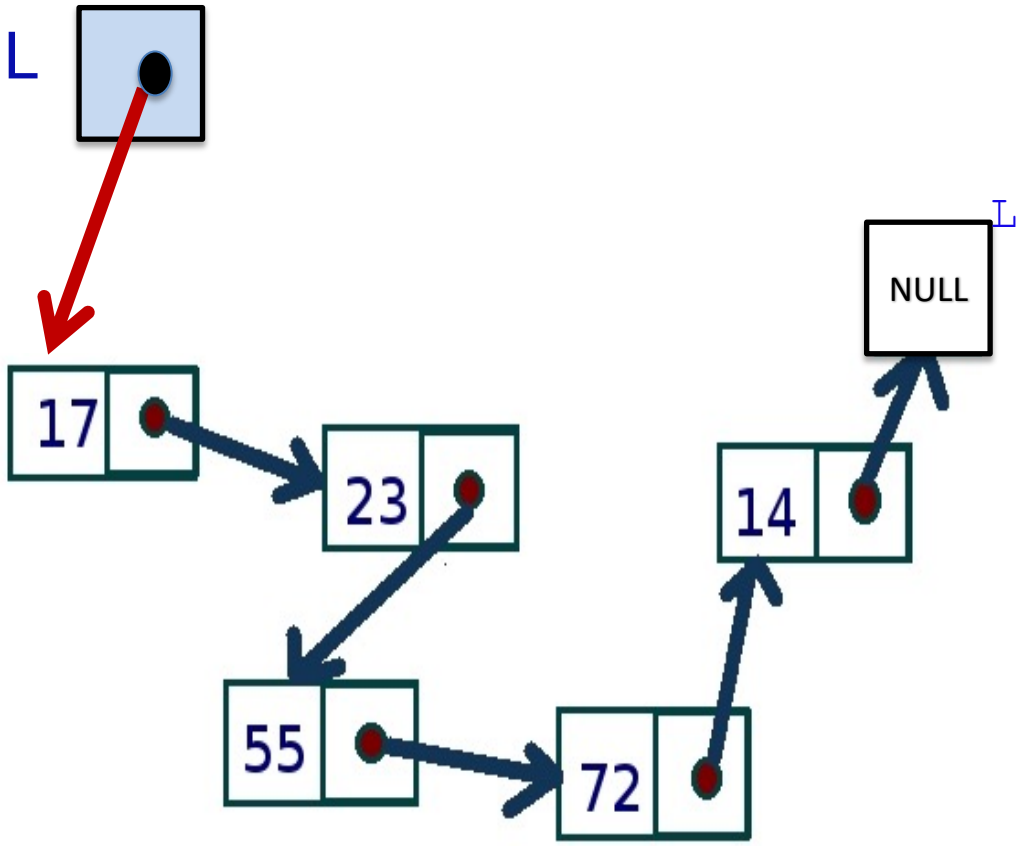
- ✓ O(1) insert at the start (insert_at_head)
- ✓ O(1) insert at the end (insert_at_foot)
- X need to maintain both head and foot when processing




```c
typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
};
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 version 2:

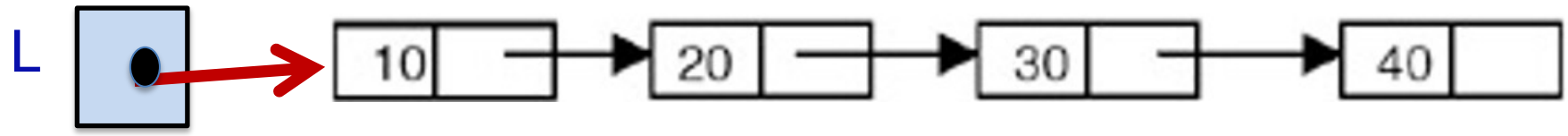
- ✓ O(1) insert at the start
- x O(n) insert at the end
- ✓ no need to maintain foot when processing



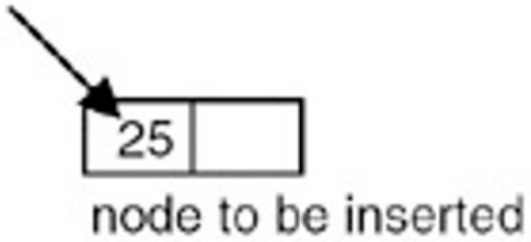

```c
typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
};
typedef node_t list_t;

int main(…) {
  list_t *L= NULL;
  L= insert_at_start(L, 14);
  L= insert_at_start(L, 72);
  L= insert_at_start(L, 55);

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

Example: Sorted Linked List & Insert

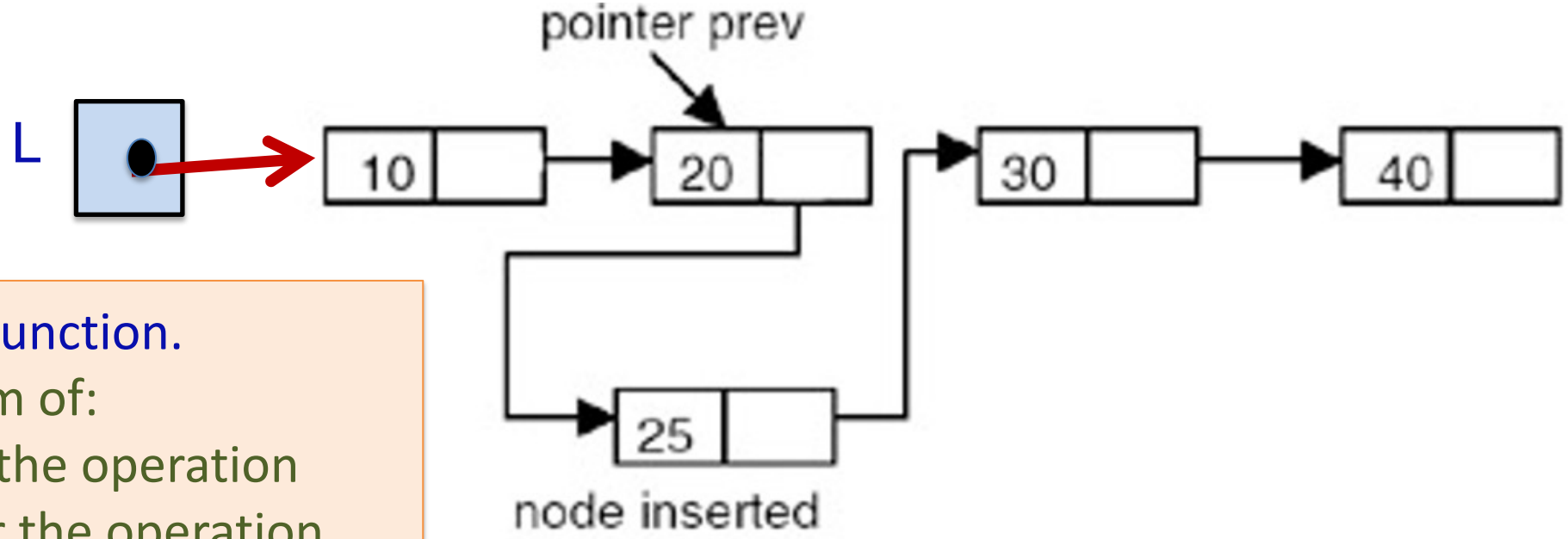
before insertion




```
inorder_insert(L, 25)
```



after insertion



Try to write this function.

Always have a clear diagram of:

- what's there before the operation
- what should be after the operation

and think of special cases!

Exercise 8 [time permitted]: 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:

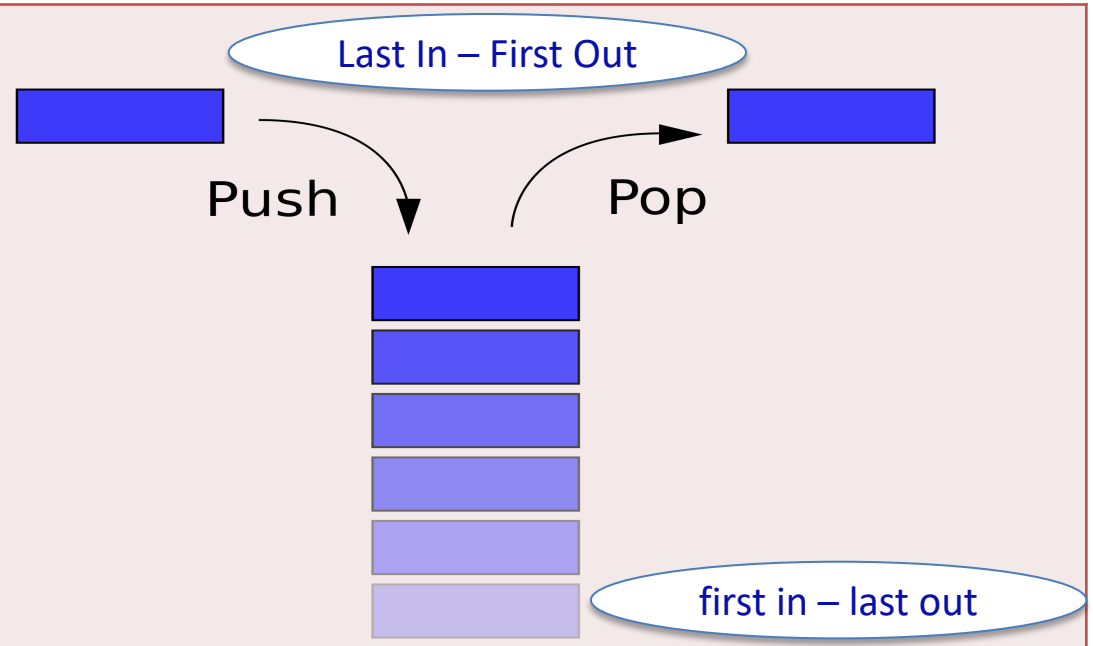
- Test it by adding 10 integers:

  `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

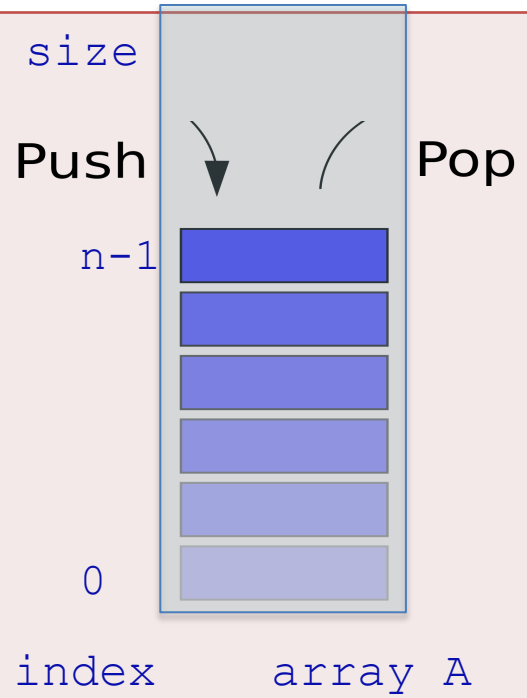
`pop()`: remove an element from (the top of) stack

`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;
```

// supposing no problem with array size

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

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

Lab Time: A2 and/or Exercises 4-8 from lec07.pdf

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.

Assignment 2

- Aim to finish Stage 0 today!
- Have a good plan for Stage 1

A2: Q&A, focusing on:

- understanding & approach
- Stage 0

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;

A2: some advices

1. Have a diagram of your data structure
2. Read through the whole specs and make notes to the structures, but don't strive for the perfection, initial code duplication is even OK!
3. Make things simple:
   - Break a complicated task into smaller tasks!
   - Don't strive too much for efficiency, simplicity is more important here
   - Don't use dynamic memory when you don't actually need it!
4. Do use in incremental development. For example, in stage 0:
   1. first make sure you can read and debug-print out the statements as they are
   2. make sure that you follow the rules about the length of strings
   3. then build the auto, this clearly done character by character:
      - what to do with the 1st character, with any next character?
      - what's before dealing with 1 character
      - what's after
5. Do test with very simple data, such as "A", "B", "AF", "AFL"
6. Use debug printing to debug or follow your code to make sure that it works as expected!

A2: Possible Tools

ctype.h : iscntrl

string.h :

- strdup,
- strcpy, strncpy
- strcmp, strncmp
- getline

dynamic arrays OR getline, attn when using getline

linked lists:

- sorted linked list?
- insert to sorted linked list
- delete from the list

Additional Slides

Review: Automatic memory allocation for arrays (by compilers)

```
#define SIZE 1000

int foo(...){
   int A[SIZE], n=0; // mem auto-allocated by compiler
   for (n=0; n<SIZE && scanf("%d", A+n)==1; n++);

   ...
   // A and n are  auto-released (auto-freed) by compiler
   //          and cannot be used outside function foo
}
```

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.

For arrays: Dynamic allocation if 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(int) );
   // 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
}
```

Dynamic 2D-arrays: we want A to be a 2D dynamic array of m rows x n columns, each element is of data type data_t;

Declaration: A is a pointer to (pointer to data_t)

```
data_t **A;
```

Building memory for A:

- make A be an array of m elements, each is an int *:

```
A= malloc(m*sizeof(*A));  assert(A);
```

- make each A[i] be an array on n elements, each is a data_t

```
for (i=0; i<m; i++) {
   A[i]= malloc( n * sizeof( *(A[i]) );
   assert(A[i]);
}
```

Use the array: (in row-majored order)

```
for (row=0; row<m; i++)
   for (col=0; col<n; col++) {
      // do stuffs with A[row][col]
   }
```

Free the array: free each row first, then free the whole array

```
for (row=0; row<m; i++) free(A[i]);
free(A);
```

Dynamic Arrays= Resizable Arrays

Notes:

as opposed to dynamic arrays, normal fixed-size arrays are sometimes referred to as static arrays

Memo: Tools for resizing (memory re-allocation)

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

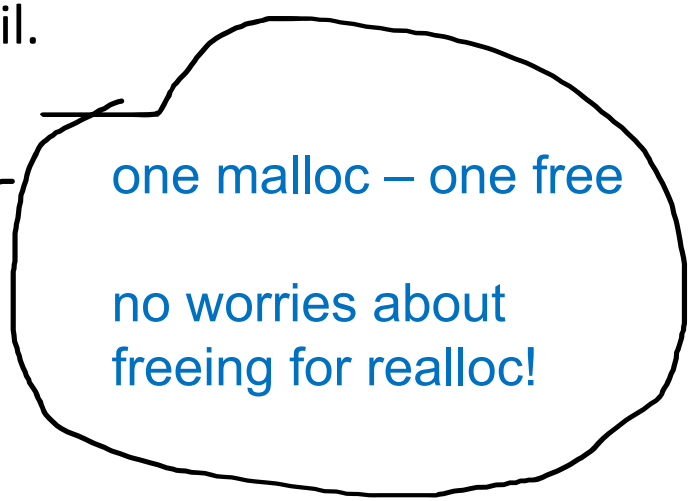
resizes the memory allocated to the old `p` by performing 4 actions:

- 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);
```



Resizable, or Dynamic Arrays

```
#define INIT_SIZE 8
  int size=INIT_SIZE              // estimated capacity

  // declares an array as a triple of variables <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
  }
  ...
```

Resizable Arrays

```
#define INIT_SIZE 5
  // 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) {   // avoid errors by re-sizing by:
      size *= 2;     // ... first makes size bigger
      A= realloc(A, size*sizeof(*A));
                     // ... then allocates a new chunk of new size
                     // ... that automatic copies and frees the old chunk
      assert(A);
    }
    A[i]= x;
  }
  ...
  free(A);       // one malloc – one free, no worry about realloc
```

“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

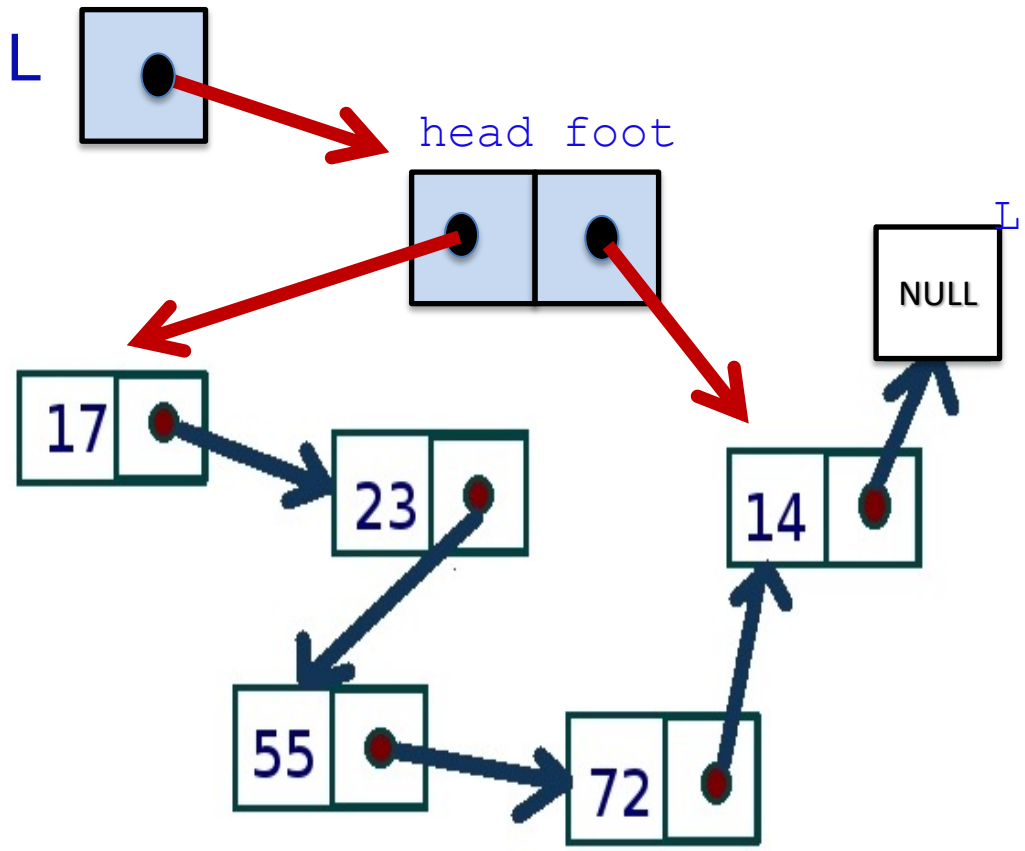
Static	Dynamic

`#define SIZE 100`
`int 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 *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);`

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

Linked List version 1: as in `listops.c`

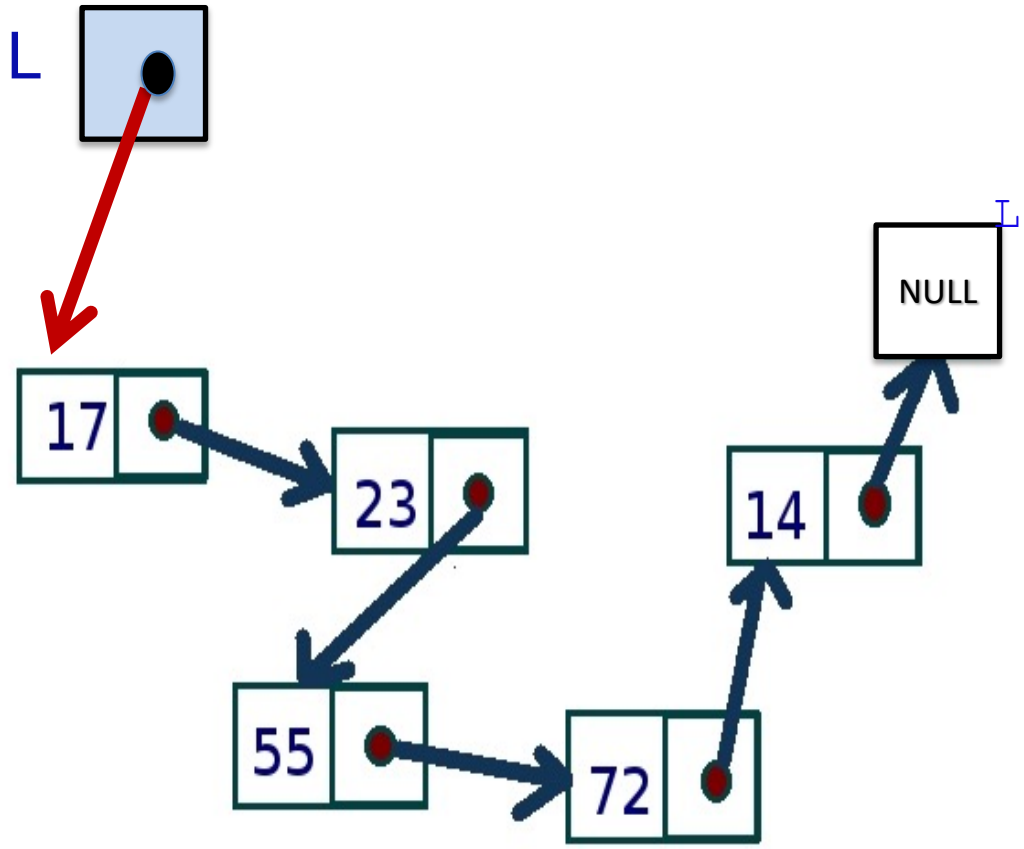
- ✓ O(1) insert at the start (insert_at_head)
- ✓ O(1) insert at the end (insert_at_foot)
- X need to maintain both head and foot when processing




```c
typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
};
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 version 2:

- ✓ O(1) insert at the start
- x O(n) insert at the end
- ✓ no need to maintain foot when processing

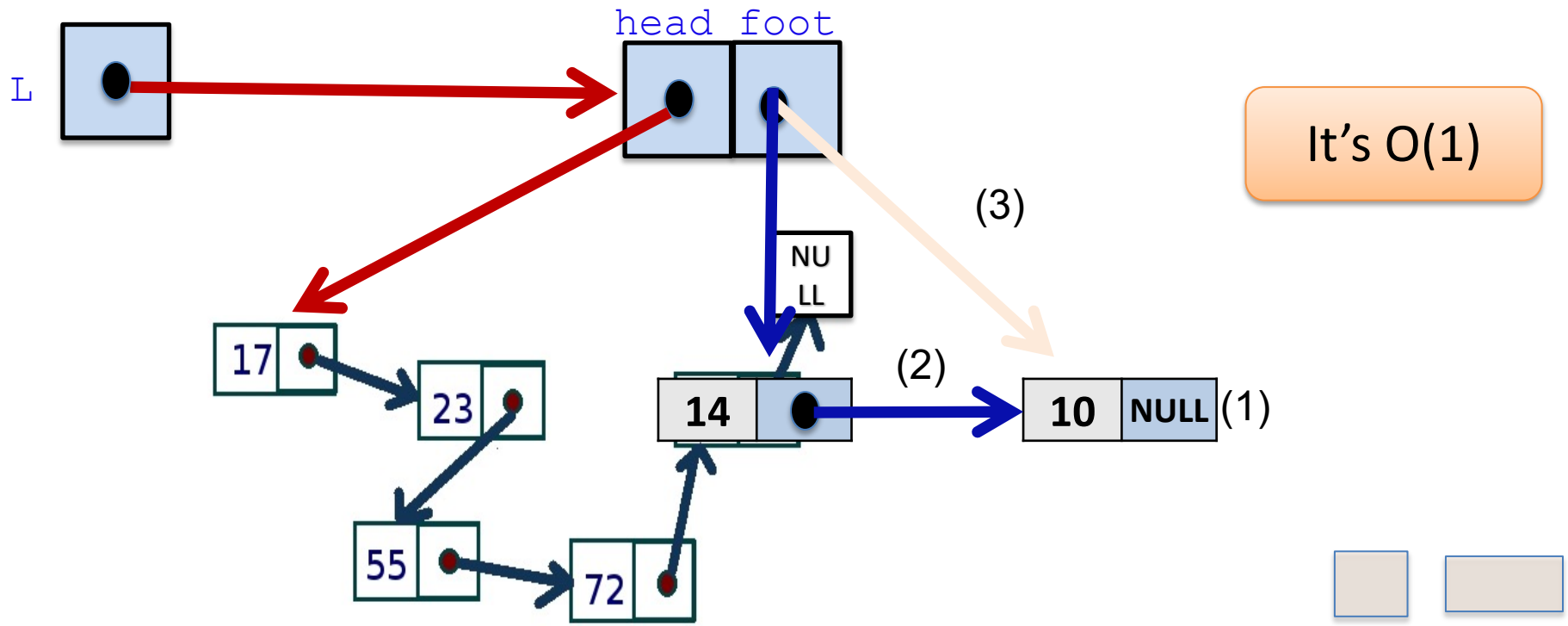



```c
typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
};
typedef node_t list_t;

int main(…) {
  list_t *L= NULL;
  L= insert_at_start(L, 14);
  L= insert_at_start(L, 72);
  L= insert_at_start(L, 55);

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

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

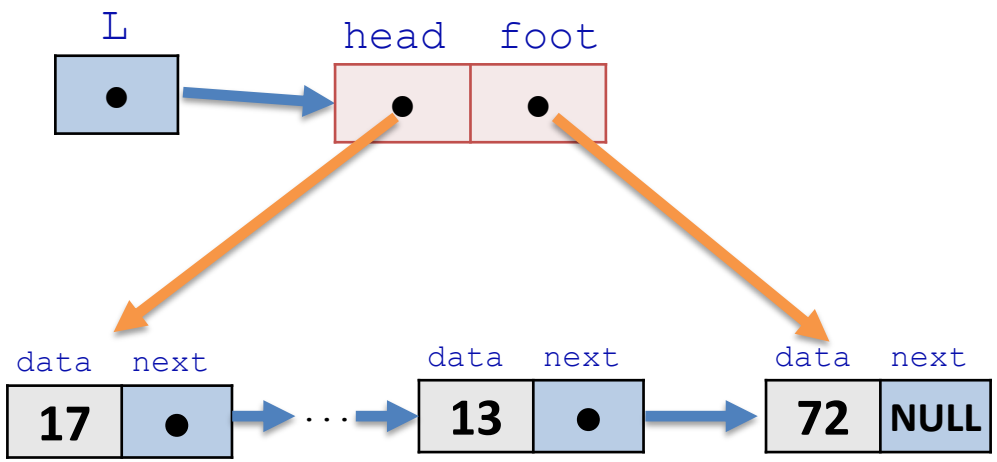
1. create new node and set data
2. Link the node to the chain
3. Repair the foot
4. if the list is empty before inserting

```
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: examples



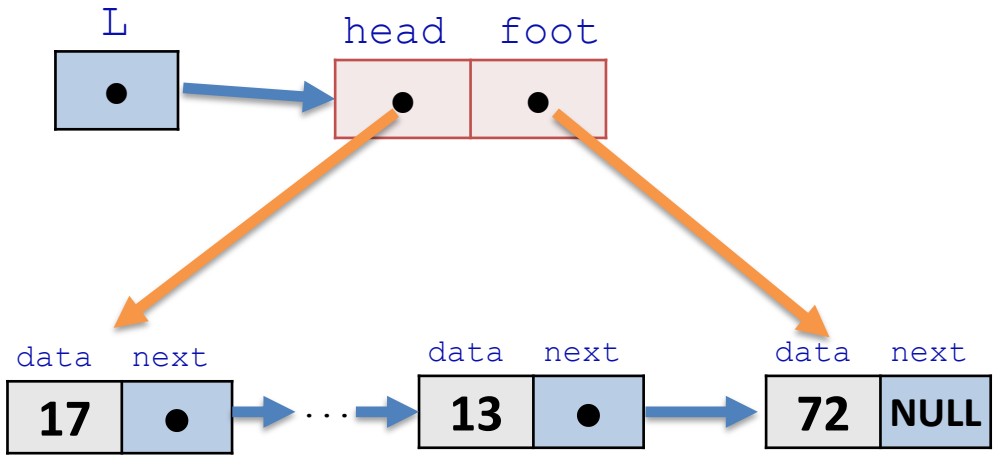

```c
typedef struct node node_t;
typedef struct node {
    data_t data;
    node_t *next;
} node_t;
typedef struct {
    node_t *head;
    node_t *foot;
} list_t;
```

```c
// print data of all nodes
void print_list(list_t *L) {
   ...
}

// cmp 2 list in lexicographical order
// returns -1, 0, 1
int listcmp(list_t *a, list_t *b) {
...
}
```

```c
// delete the 2nd node (supposed L has at least 2 nodes)
node_t *prev= L->head;      // preceding node
node_t *curr= prev->next;
...
```

Linked List: examples of using

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




```
// print data of all nodes
void print_list(list_t *L) {
  node_t *p= L->head;
  while ( p!=NULL) {
    printf("%d ", p->data);
    p= p->next;
  }
}
```

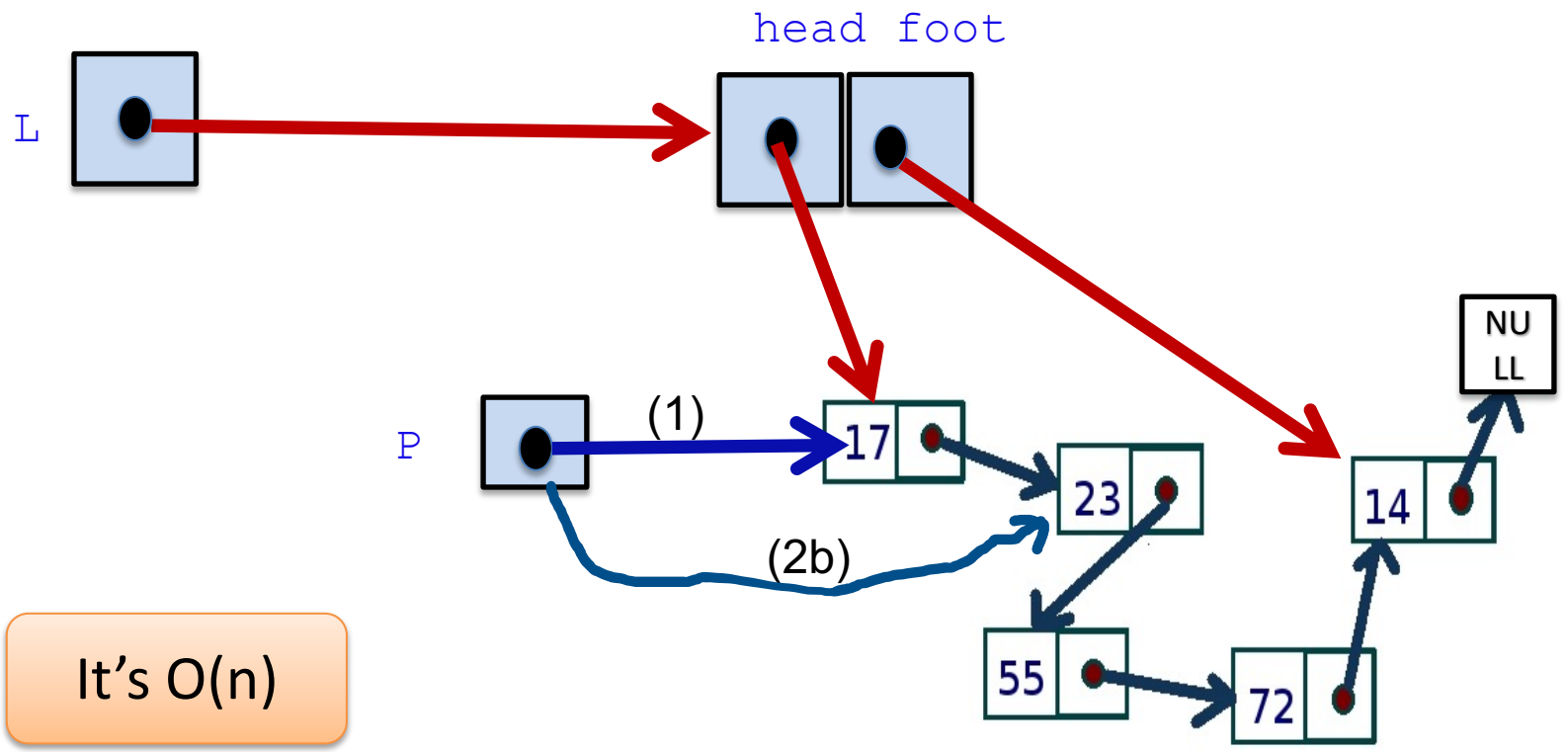
```
// delete the 2nd node (supposed L has at least 2 nodes)
node_t *prev= L->head;      // preceding node
node_t *curr= prev->next;
prev->next= curr->next;
if (curr->next==NULL) L->foot= prev;
free(curr);
```

/* Note 1: delete a generous node is more complicated (because it can be the first node). In principle:

- check if the list is empty
- check if the deleted node is the head
- keep track of the prev and curr to delete the curr
- check if the delete node is the foot

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)