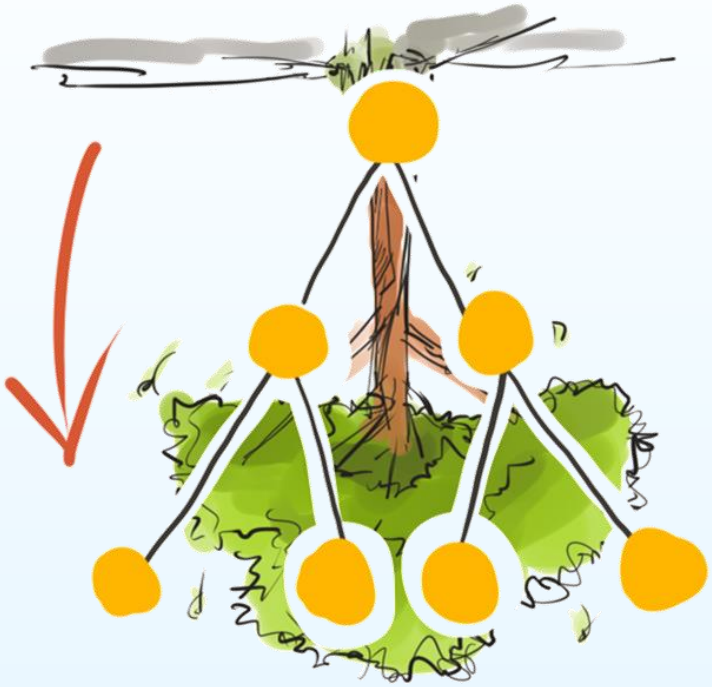


Lecture 5:
List ADT

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

Outline

- Abstract Data Types
- C Structure
- List ADT
  - Array List
  - Linked List

What is a Data Structure?

"A data structure (DS) is a way of organizing data so that it can be used effectively"

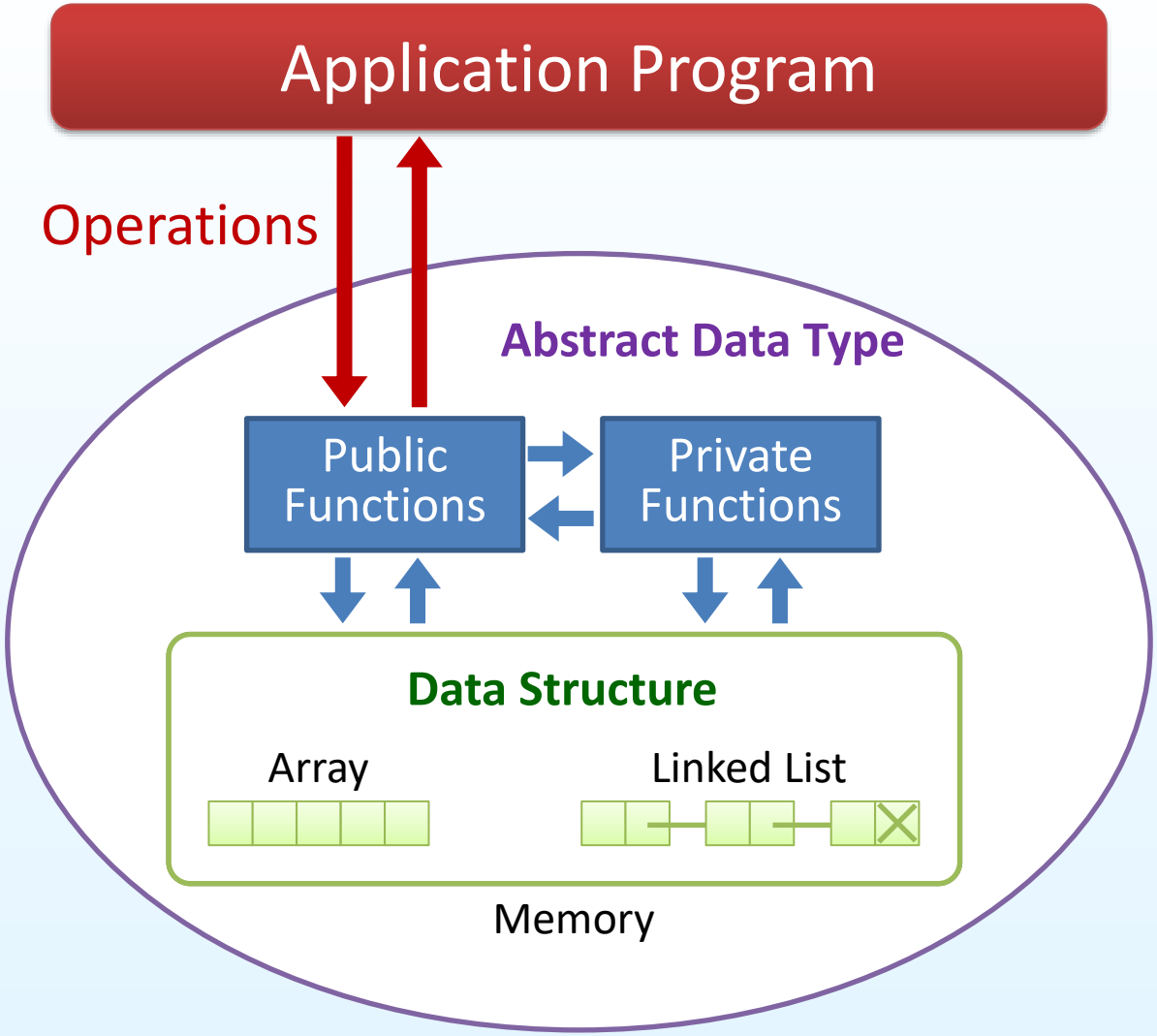
- It helps to manage and organize data
- It is essential ingredients in creating fast and powerful algorithms
- It makes code cleaner and easier to understand

What is an Abstract Data Type?

"An abstract data type (ADT) is an abstraction of a data structure which provides only interfaces to which a data structure must adhere to"

- The interface does not give any specific details about how something should be implemented or in what programming language

ADT vs. DS


Application Program
Operations
Abstract Data Type
Public Functions
Private Functions
Data Structure
Array
Linked List
Memory

Examples: ADT vs. DS

Abstract Data Type	Data Structure
List ADT Data: values are stored in a sequence Operations: insert, search, delete, …	• array • linked list
Stack ADT Data: values are stored linearly, but the order is performed by LIFO Operations: push, pop, isEmpty, isFull, …	• array • linked list
Queue ADT Data: values are stored linearly, but the order is performed by FIFO Operations: enqueue, dequeue, …	• array • linked list

Outline

- Abstract Data Types
- C Structure
- List ADT
  - Array List
  - Linked List

Motivation for a Structure

- In some situations, you may want to keep a person's information such as name, age, gender, salary:

```
char  name[20];
int   age;
char  gender;
float salary;
```

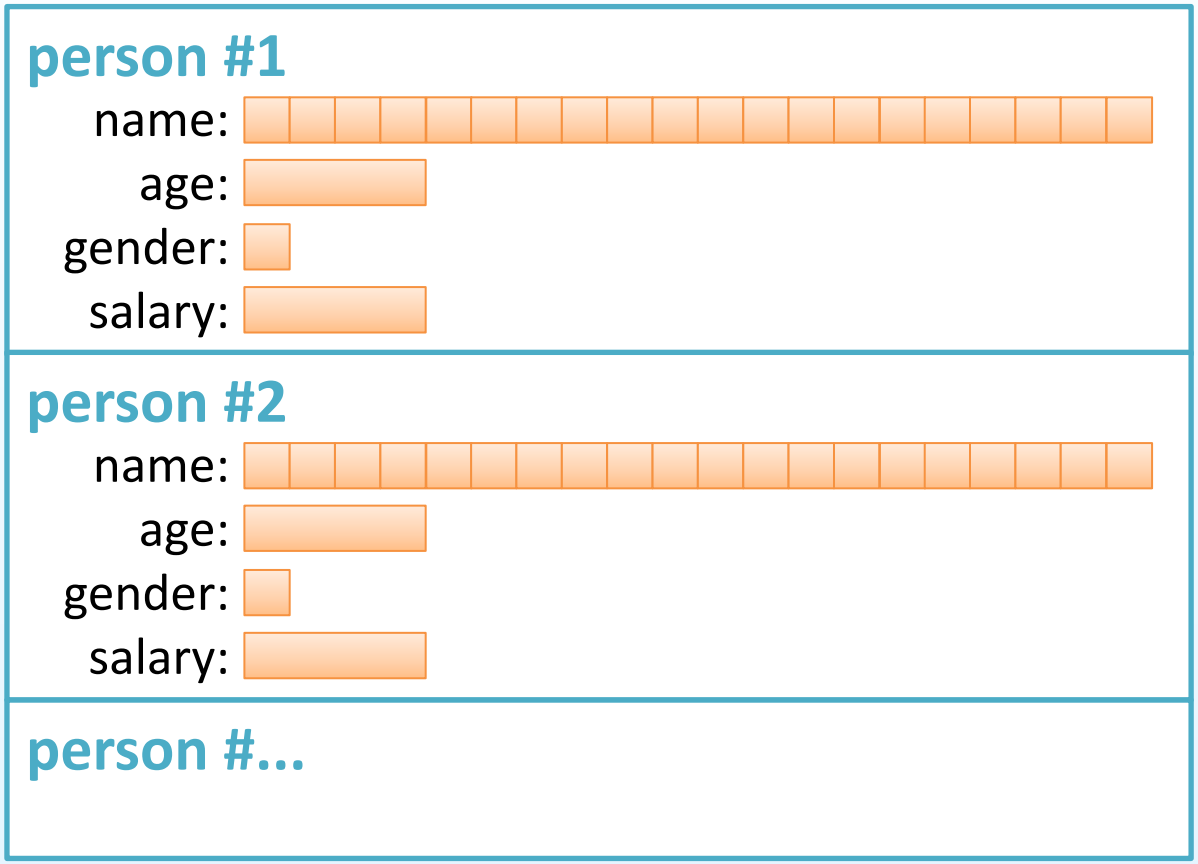
- If you want to keep information for 100 people:

```
char  name[100][20];
int   age[100];
char  gender[100];
float salary[100];
```

- So …, how do you sort all the information by the person names in increasing order?

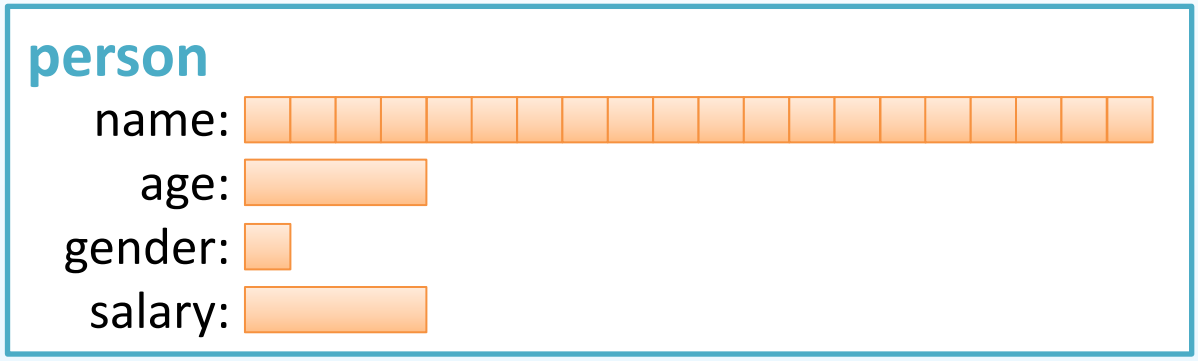
Motivation for a Structure

- We may need a data structure to encapsulate information of a person, and then allocate memory for 100 people



Structure in C

- A collection of one or more variables grouped under a single name
- Each variable in a structure can be of different types



Structure Definition

- Use `struct` keyword
- Define variables (members) within the block of structure
- Cannot initialize each member of the structure
- End structure declaration with semicolon (;)

Defined data type

```
struct person {
   char   name[20];
   int    age;
   char   gender;
   float  salary;
};
```

```
struct person {
   char   name[20] = "";
   int    age = 0;
   char   gender = 'm';
   float  salary = 0;
};
```

Each member cannot be initialized!!!

Structure Variable Declaration

- Declare variables at the end part of the definition

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
} staff;
```

- Declare variables later if need

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
...
struct person staff;
```

Variable Declaration with Initializer

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
...
struct person staff01 = {"John", 40,
                         'm', 9500};

struct person staff02 = {"Mary", 32,
                         'f', 8000};
```

Symbol		Address	Value
high		...	
		0x7fffe9	
	salary	0x7fffe5	8000.0
	gender	0x7fffe1	f
	age	0x7fffdd	32
staff02	name	0x7fffc9	Mary
	salary	0x7fffc5	9500.0
	gender	0x7fffc1	m
	age	0x7fffbd	40
staff01	name	0x7fffa9	John
		0x7fffa8	
low		...	

Accessing Members of Structure

- Dot operator (.) is used to normally access members

```c
#include <stdio.h>
#include <string.h>

struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};

int main(void) {
  struct person staff;

  strcpy(staff.name, "John");
  staff.age = 40;
  staff.gender = 'm';
  staff.salary = 9500;
  return 0;
}
```

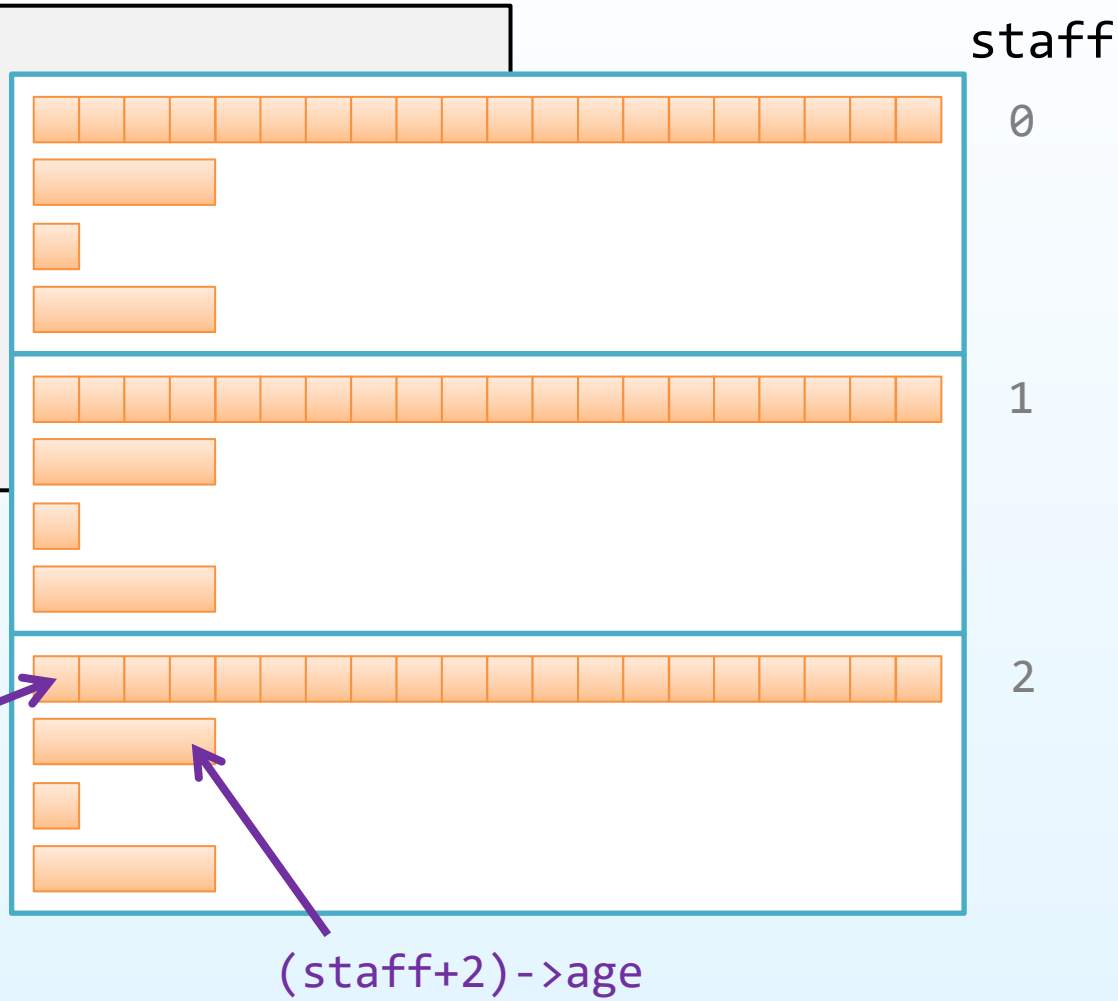
Accessing Members of Structure

- Arrow operator (->) is used to access members by pointer

```c
#include <stdio.h>
#include <string.h>

struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};

int main(void) {
  struct person staff;
  struct person *pStaff = &staff;

  strcpy(pStaff->name, "John");
  pStaff->age = 40;
  pStaff->gender = 'm';
  pStaff->salary = 9500;
  return 0;
}
```

Array of Structures

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
...
struct person staff[3];
```



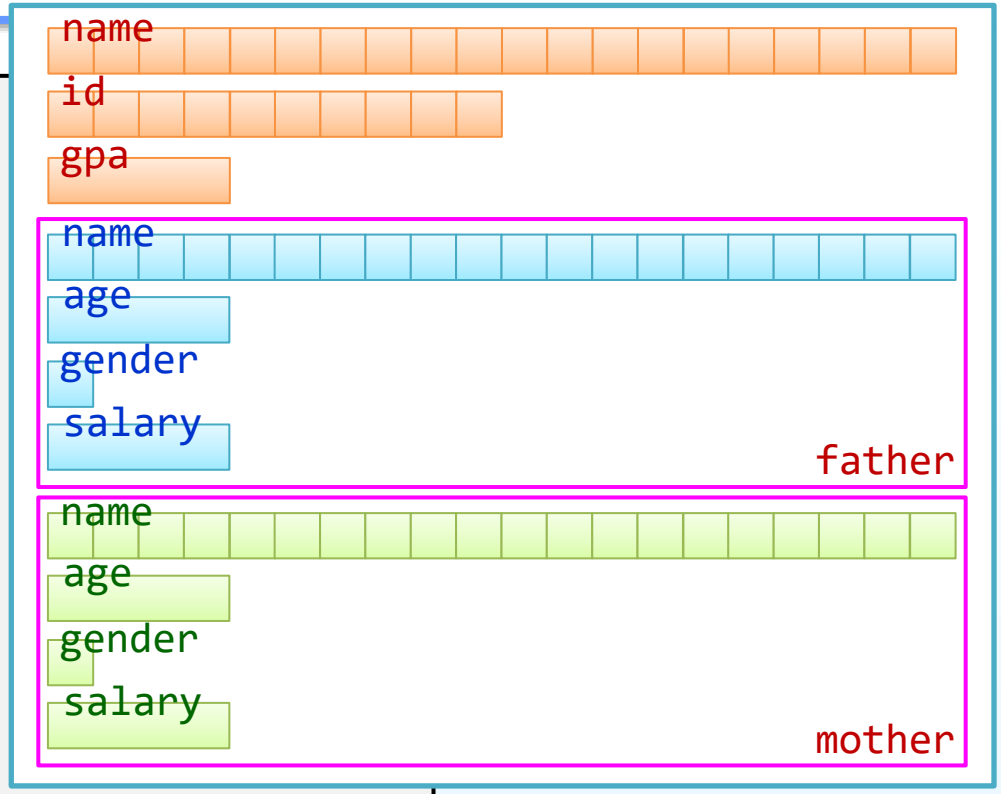
Exercise 1: Complex Structure

Design and implement a data structure to store:

- your name, id, and GPA
- your father's name, age, and salary
- your mother's name, age, and salary

Solution to Exercise 1

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
struct profile {
  char  name[20];
  char  id[10];
  float gpa;
  struct person father;
  struct person mother;
};
...
struct profile student;
```




```
struct profile student =
  {"Jim", "6350101234", 3.50,
    {"John", 40, 'm', 9500},
    {"Mary", 32, 'f', 8000}
  };
```

How do you print out your father's name?

The typedef Keyword

- Create a synonym (alias) for a data type
  - Do not create a new data type

```
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
struct profile {
  char  name[20];
  char  id[10];
  float gpa;
  struct person father;
  struct person mother;
};
typedef struct profile profile_t;
```

```
struct profile std01 =
  {"Jim", "6250101234", 3.50,
    {"John", 40, 'm', 9500},
    {"Mary", 32, 'f', 8000}
  };

profile_t std02 =
  {"Jay", "6250101235", 3.89,
    {"Eric", 41, 'm', 9800},
    {"Zena", 35, 'f', 9000}
  };
```

The `sizeof` Operator

- Obtain size in byte of an object, including a structure

```c
struct person {
  char  name[20];
  int   age;
  char  gender;
  float salary;
};
struct profile {
  char  name[20];
  char  id[10];
  float gpa;
  struct person father;
  struct person mother;
};
typedef struct profile profile_t;
```

```c
int main(void) {
  profile_t student =
    {"Jim", "6250101234", 3.50,
      {"John", 40, 'm', 9500},
      {"Mary", 32, 'f', 8000}
    };

  printf("Size : %ld\n", sizeof student);
  return 0;
}
```

```
Size : 100
```

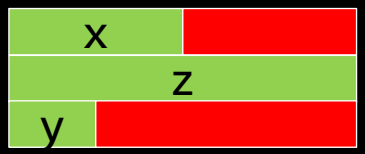
The sizeof Operator

- The sizeof for a struct is not always equal to the sum of sizeof of each individual member
- The padding added by the compiler to avoid alignment issues
  - Different compilers might have different alignment constraints

Program #1:

```
#include <stdio.h>
int main(void) {
  struct A {
    int x;
    double z;
    short y;
  } a;
  printf("Size: %ld\n", sizeof a);
  printf("x: %p\n", &a.x);
  printf("z: %p\n", &a.z);
  printf("y: %p\n", &a.y);
  return 0;
}
```

```
Size: 24
x: 0x7ffff0d18af0
z: 0x7ffff0d18af8
y: 0x7ffff0d18b00
```



Program #2:

```
#include <stdio.h>
int main(void) {
  struct B {
    double z;
    short y;
    int x;
  } b;
  printf("Size: %ld\n", sizeof b);
  printf("z: %p\n", &b.z);
  printf("y: %p\n", &b.y);
  printf("x: %p\n", &b.x);
  return 0;
}
```

```
Size: 16
z: 0x7ffd2b2c2390
y: 0x7ffd2b2c2398
x: 0x7ffd2b2c239c
```

Outline

- Abstract Data Types
- C Structure
- List ADT
  - Array List
  - Linked List

What is a List ADT?

- Data:
  - Ordered sequence of elements $e_1, e_2, \ldots, e_N$
  - Elements may be of arbitrary type, but all are the same type

- Common operations:
  - `insert(`*`list`*`, `*`value`*`, `*`position`*`)`
  - `delete(`*`list`*`, `*`position`*`)`
  - `find(`*`list`*`, `*`value`*`)`
  - `is_empty(`*`list`*`)`
  - `is_full(`*`list`*`)`
  - …

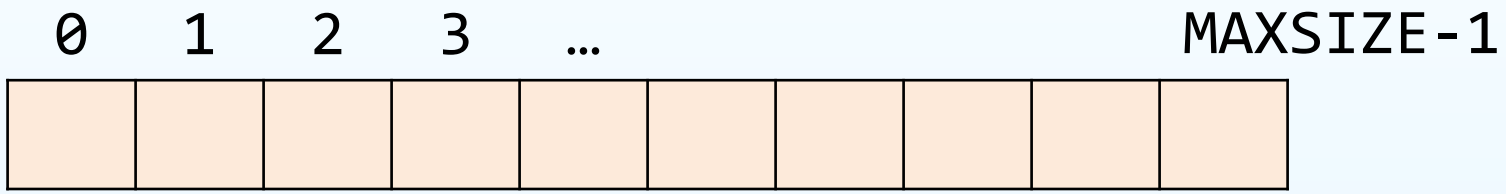
List Implementations

- Two types of implementation
  - Array-based list
  - Pointer-based list

List: Array Implementation

Basic idea:

- Pre-allocate a big array (of size `MAXSIZE`)
- Keep track of current size (using a variable `count`)



`count` = *`current_position`*

Note: the array index starts with `0`

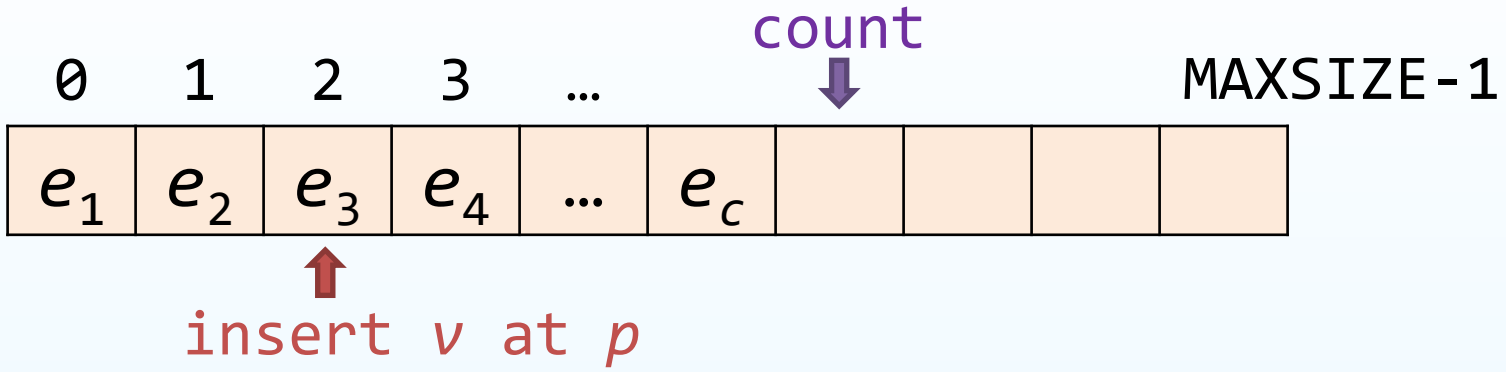
Array-based List: Construction

Assume that all data are integer

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #define MAXSIZE 10000
4:
5:  typedef struct list {
6:    int *arr;     // array-based list
7:    int count;    // current position
8:  } list_t;
9:
10: list_t create() {
11:   list_t l = {NULL, 0}
12:   l.arr = (int *)malloc(sizeof (int) * MAXSIZE);
13:   return l;
14: }
15: int main(void) {
16:   list_t l = create();
17:   return 0;
18: }
```

Array-based List: `insert()` Operation

Insert a value *v* at the position *p* in list *l*

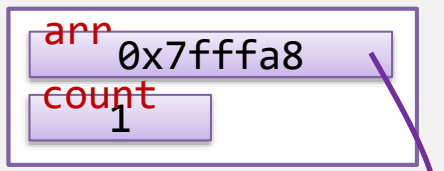


- If *p* < count, shift all elements from *p* to the right
- If *p* == count, insert *v* at *p* (end of list)
- If *p* > count, insert *v* at count (end of list)
- If the list is full (count == MAXSIZE), return an error
- If *p* is out of range, return an error

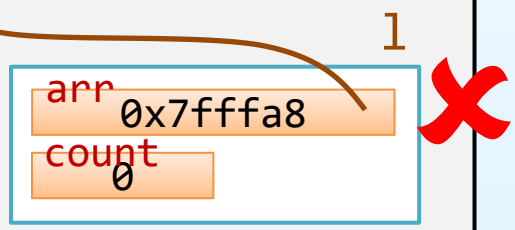
Array-based List: `insert()` Operation

Insert a value *v* at the position *p* in list *l*

```
1:  int insert(list_t l, int v, int p) {
2:    int i = 0;
3:    if (l.count == MAXSIZE)       // list is full
4:      return 0;
5:    if (p < 0 || p > MAXSIZE-1)   // out of range
6:      return 0;
7:
8:    if (p < l.count) {            // shift right
9:      for (i=l.count; i>p; i--)
10:       l.arr[i] = l.arr[i-1];
11:     l.arr[p] = v;
12:   }
13:   else                          // insert at the end of list
14:     l.arr[l.count] = v;
15:
16:   l.count++;
17:   return 1;
18: }
```




```
int main(void) {
  list_t l = create();
  insert(l, 10, 0);
  return 0;
}
```

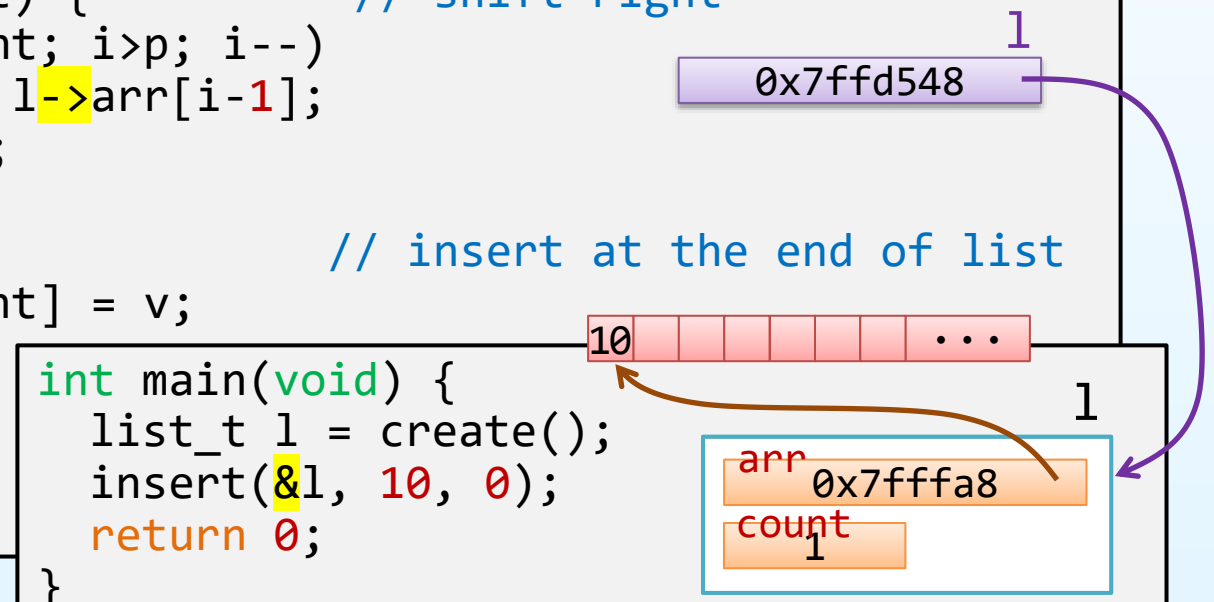


Array-based List: `insert()` Operation

Insert a value *v* at the position *p* in list *l*

```
1:  int insert(list_t *l, int v, int p) {
2:    int i = 0;
3:    if (l->count == MAXSIZE)     // list is full
4:      return 0;
5:    if (p < 0 || p > MAXSIZE-1)  // out of range
6:      return 0;
7:
8:    if (p < l->count) {          // shift right
9:      for (i=l->count; i>p; i--)
10:       l->arr[i] = l->arr[i-1];
11:     l->arr[p] = v;
12:   }
13:   else                         // insert at the end of list
14:     l->arr[l->count] = v;
15:
16:   l->count++;
17:   return 1;
18: }
```

```
int main(void) {
  list_t l = create();
  insert(&l, 10, 0);
  return 0;
}
```



Exercise 2: `delete()` and `find()`

Implement the functions `delete()` and `find()` operated on an array-based list:

- Delete at the position *p* in list *l*
  - Return 1 if it can be deleted normally, otherwise 0

- Find the value *v* in list *l*
  - Return the position *p* if it is found, otherwise -1

Array-based List: Running Time

Operation	Running Time
`insert()` at p	$O(n)$
`delete()` at p	$O(n)$
`find()`	$O(n)$
`replace()` at p	$O(1)$
`is_empty()`	$O(1)$
`is_full()`	$O(1)$
`concat()` two lists	$O(n)$

Limitation of Array-based List

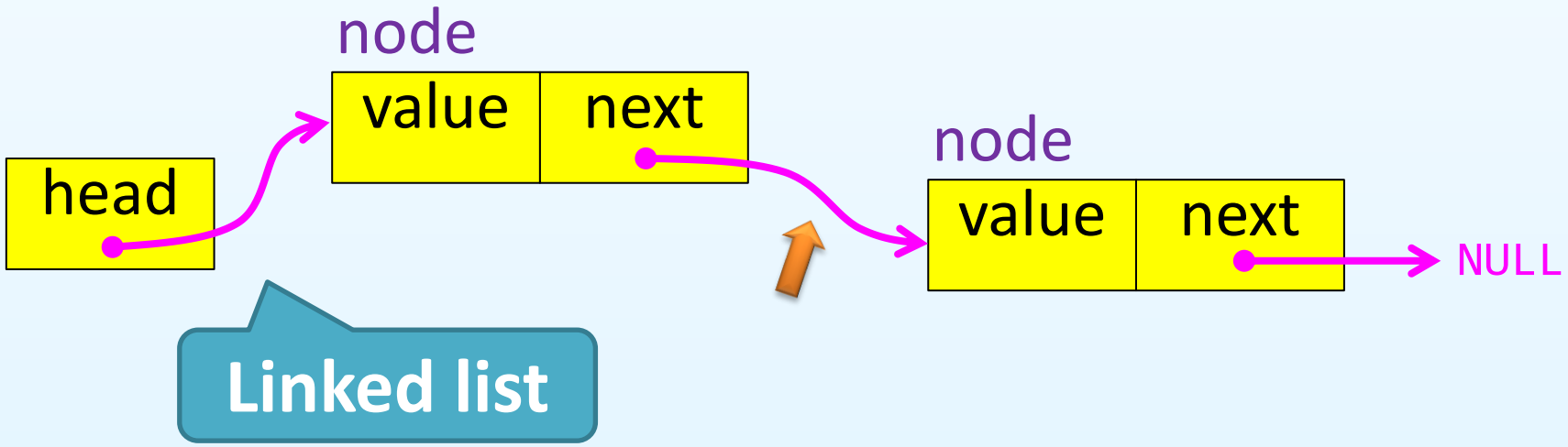
- If the list is full, reallocate a huge new array and move everything over

- If we need an `insert_front()` operation, $O(n)$ is required (worst-case)

List: Pointer Implementation

Basic idea:

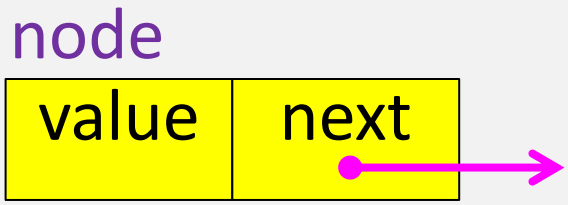
- Allocate little blocks of memory (nodes) as elements are added to the list
- Keep track of list by linking the node together
- Change links when you want to insert or delete



Linked List: Construction

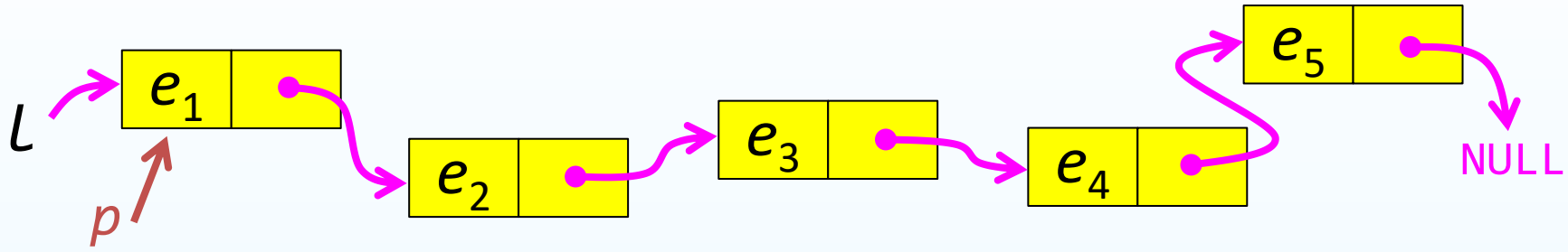
Assume that all data are integer

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: typedef struct node {
 5:   int value;
 6:   struct node *next;
 7: } node_t;
 8:
 9: typedef node_t list_t;
10:
11: int main(void) {
12:   list_t *l = NULL;
13:   return 0;
14: }
```



Linked List: `find()` Operation

Find the value *v* through list *l*

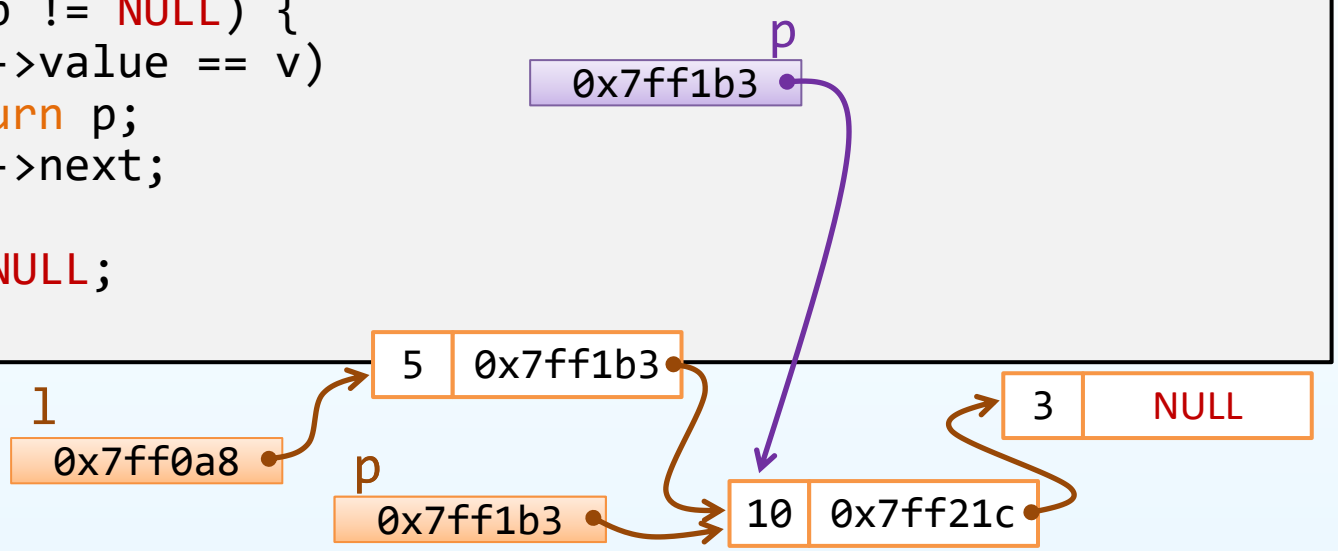


- Start the pointer *p* at the header and explore the list till the tail
  - If the value of node pointed by *p* is equal *v*, return *p*
  - Otherwise, return `NULL`

Linked List: `find()` Operation

Find the value *v* through list *l*

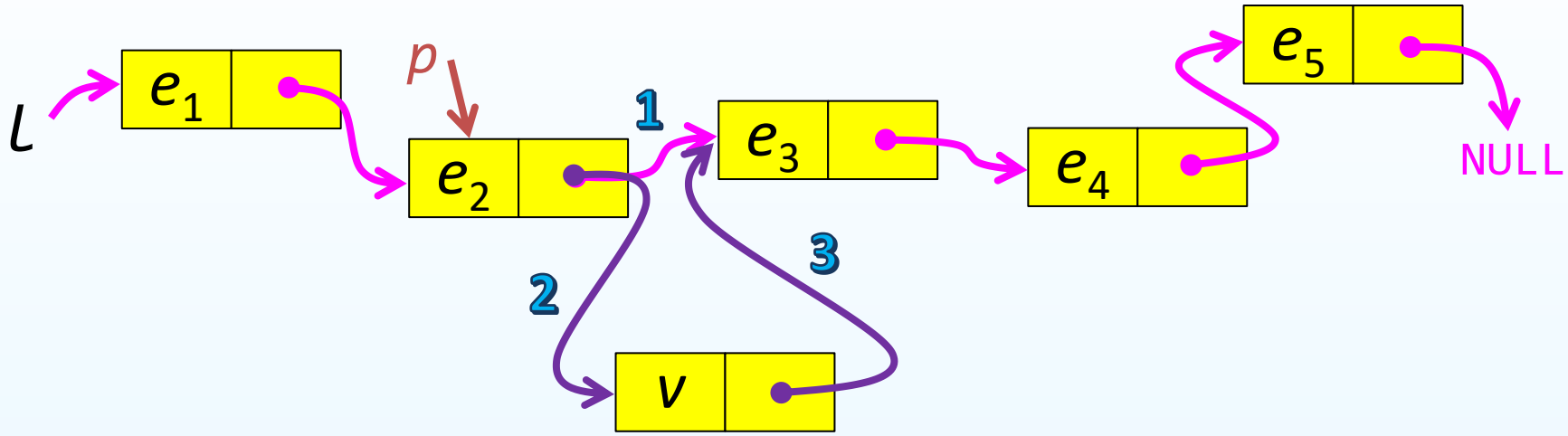
```
1: list_t *find(list_t *p, int v) {
2:   while (p != NULL) {
3:     if (p->value == v)
4:       return p;
5:     p = p->next;
6:   }
7:   return NULL;
8: }
```




```
int main(void) {
  list_t *l = NULL, *p = NULL;
  ...
  p = find(l, 10);
  printf("%s\n", (p != NULL)? "found" : "not found");
  return 0;
}
```

Linked List: `insert_after()` Operation

Insert a value *v* after the node pointed by *p* in list *l*



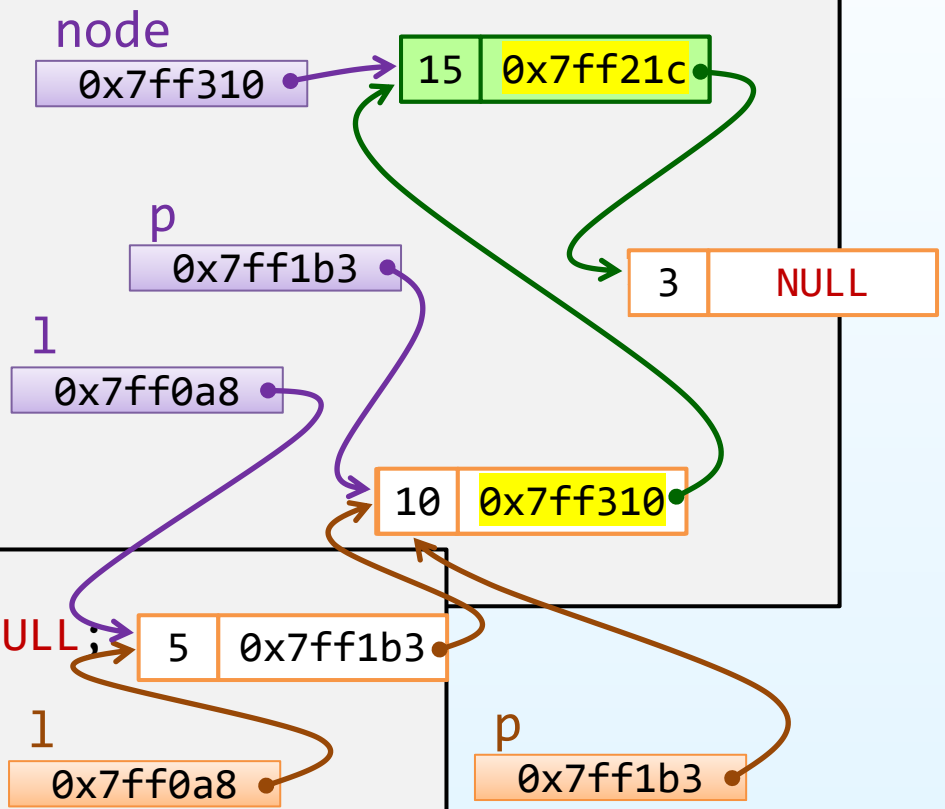
- If *l* is `NULL`, insert at head of the list
- Otherwise, insert normally and keep track the list
  - add link (3)
  - replace link (1) by link (2)

Linked List: `insert_after()` Operation

Insert a value *v* after the position *p* in list *l*

```
1:  int insert_after(list_t *l, int v, list_t *p) {
2:    node_t *node = (node_t *)malloc(sizeof (node_t));
3:    node->value = v;
4:    node->next = NULL;
5:
6:    if (l == NULL)
7:      l = node;
8:    if (p == NULL) return 0;
9:    else {
10:     node->next = p->next;
11:     p->next = node;
12:   }
13:   return 1;
14: }
```

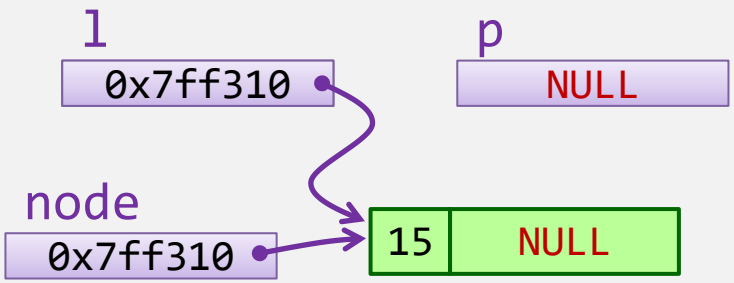
```
int main(void) {
  list_t *l = NULL, *p = NULL;
  ...
  p = find(l, 10);
  insert_after(l, 15, p);
  return 0;
}
```



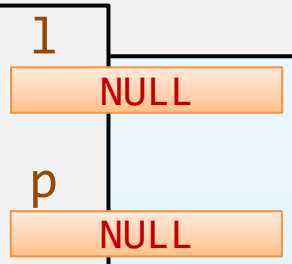
Linked List: `insert_after()` Operation

Insert a value *v* after the position *p* in list *l*

```
1:  int insert_after(list_t *l, int v, list_t *p) {
2:    node_t *node = (node_t *)malloc(sizeof (node_t));
3:    node->value = v;
4:    node->next = NULL;
5:
6:    if (l == NULL)
7:      l = node;
8:    if (p == NULL) return 0;
9:    else {
10:     node->next = p->next;
11:     p->next = node;
12:   }
13:   return 1;
14: }
```




```
int main(void) {
  list_t *l = NULL, *p = NULL;
  insert_after(l, 15, p);
  return 0;
}
```

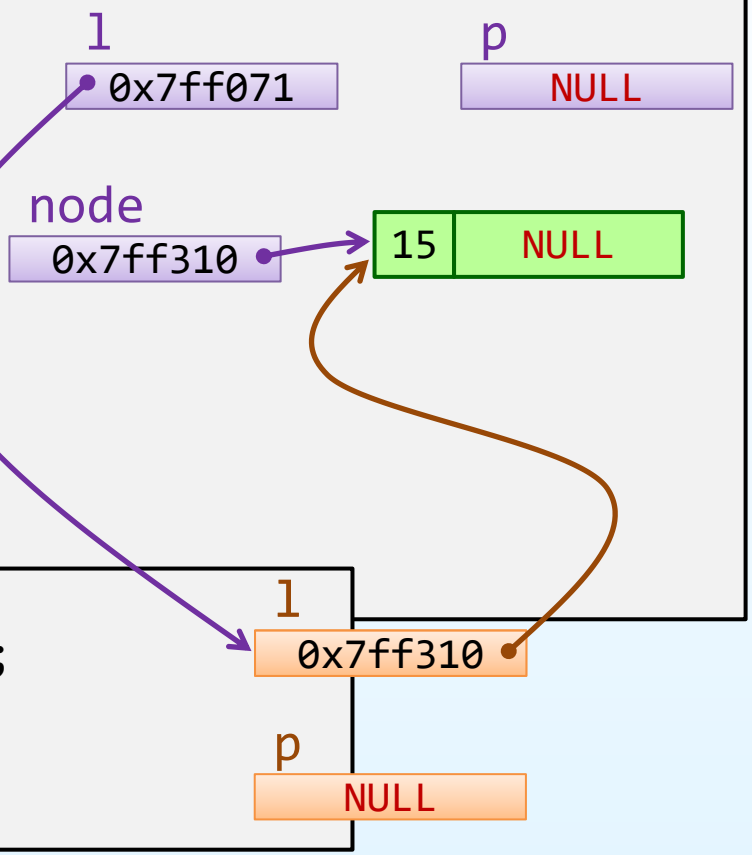


Linked List: `insert_after()` Operation

Insert a value *v* after the position *p* in list *l*

```
1:  int insert_after(list_t **l, int v, list_t *p) {
2:    node_t *node = (node_t *)malloc(sizeof (node_t));
3:    node->value = v;
4:    node->next = NULL;
5:
6:    if (*l == NULL)
7:      *l = node;
8:    if (p == NULL) return 0;
9:    else {
10:     node->next = p->next;
11:     p->next = node;
12:   }
13:   return 1;
14: }
```

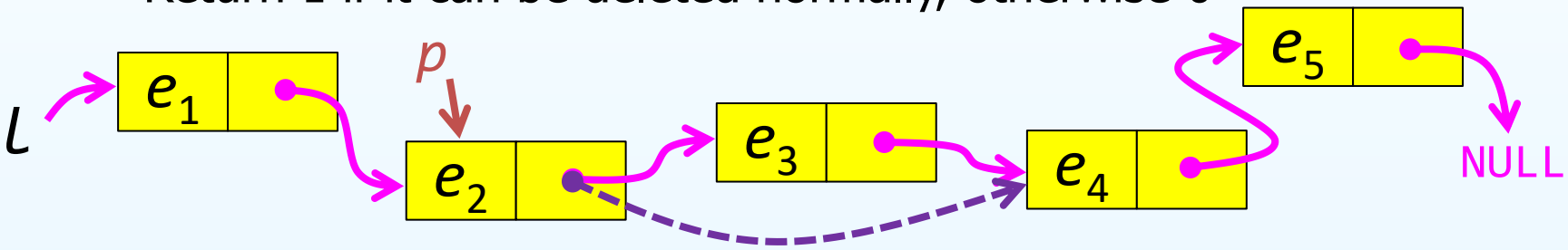
```
int main(void) {
  list_t *l = NULL, *p = NULL;
  insert_after(&l, 15, p);
  return 0;
}
```



Exercise 3: `delete_after()` and `explore()`

Implement the functions `delete_after()` and `explore()` operated on a linked list:

- Delete the value *v* after the position *p* in list *L*
  - Return 1 if it can be deleted normally, otherwise 0



- Explore (i.e., print) all values in list *L*
  - No returned value

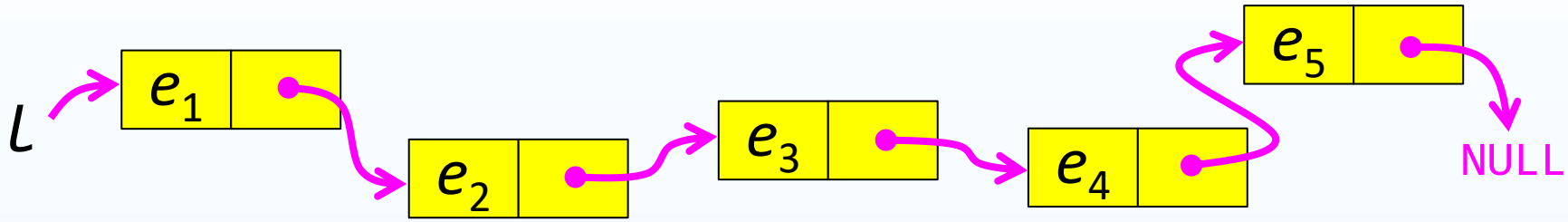
linked List: Running Time

Operation	Running Time
`insert_after()`	$O(1)$
`insert_at()`	$O(n)$
`delete_after()`	$O(1)$
`delete_at()`	$O(n)$
`find()`	$O(n)$
`explore()`	$O(n)$
`replace_at()`	$O(1)$
`destroy()`	$O(n)$

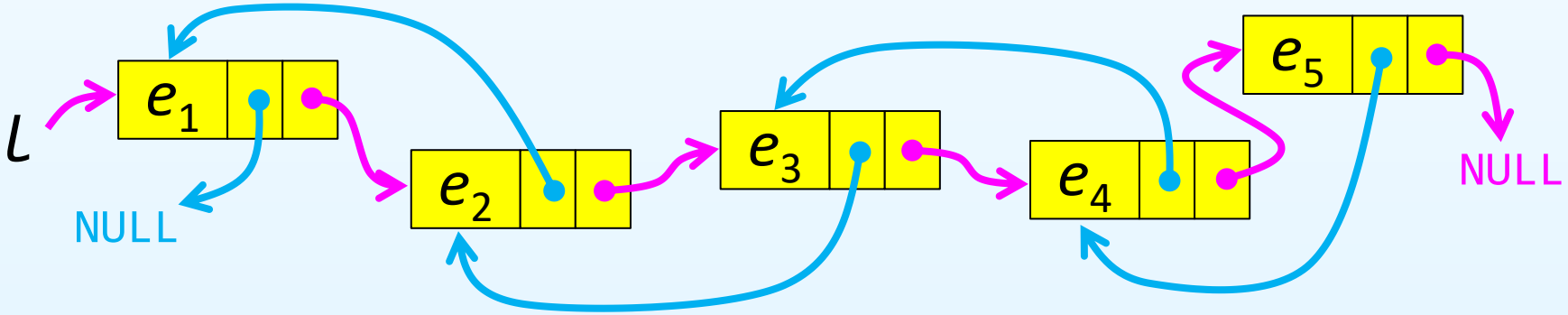
How can you make them more efficient?

Doubly Linked Lists

- The previous version is called a singly linked list

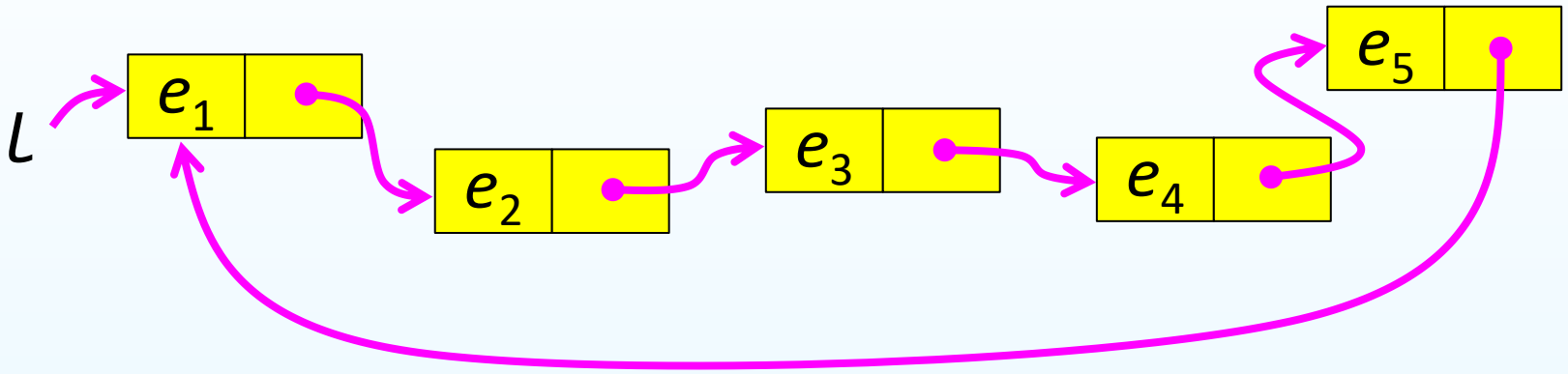


- Doubly linked list is a list that each node points to the previous and next node



Circularly Linked Lists

- A circular linked list is a list that the last node points to the first node instead of NULL



- You may design the two-way pointers for a circular doubly linked list to make it more effective

Any Question?

Solution to Exercise 2: Delete()

Delete the value *v* at the position *p* in list *l*

```
 1: int delete(list_t *l, int p) {
 2:   int i = 0;
 3:   if (l->count == 0)           // list is empty
 4:     return 0;
 5:   if (p < 0 || p > MAXSIZE-1)  // out of range
 6:     return 0;
 7:
 8:   if (p < l->count)            // shift left
 9:     for (i=p; i<(l->count)-1; i++)
10:       l->arr[i] = l->arr[i+1];
11:
12:   l->count--;
13:   return 1;
14: }
```

Solution to Exercise 2: `find()`

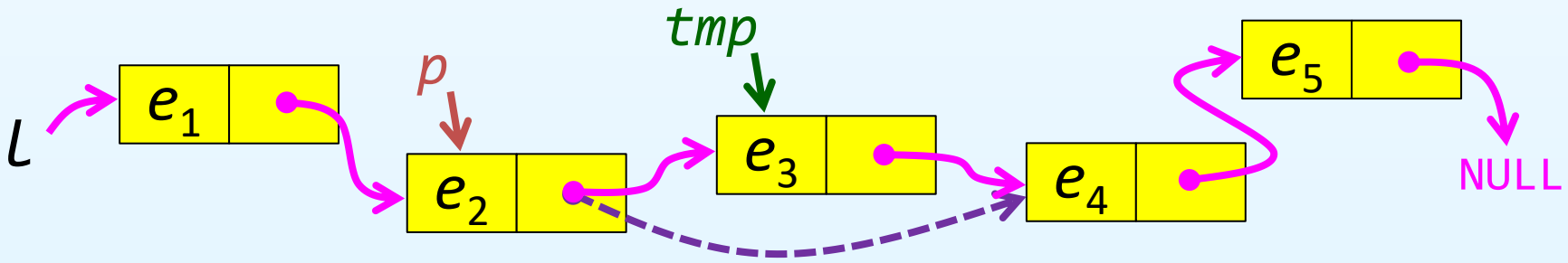
Find the value *v* in list *l*

```
1: int find(list_t l, int v) {
2:   int i = 0;
3:
4:   for (i=0; i<l.count; i++)
5:     if (l.arr[i] == v)
6:       return i;
7:   return -1;
8: }
```

Solution to Exercise 3: `Delete_after()`

Delete the value *v* after the position *p* in list *l*

```
1:  int delete_after(list_t *l, list_t *p) {
2:    list_t *tmp = NULL;
3:
4:    if (l == NULL || p == NULL)
5:      return 0;
6:    else {
7:      tmp = p->next;
8:      p->next = p->next->next;
9:      free(tmp);
10:   }
11:   return 1;
12: }
```



Solution to Exercise 3: `explore()`

Explore (i.e., print) all values in list *l*

```
1: void explore(list_t *p) {
2:   while (p != NULL) {
3:     printf("Data: %d\n", p->value);
4:     p = p->next;
5:   }
6: }
```