First there were linked lists…then…

Abstract Data Types

Abstract Data Type (ADT)

- A collection of data structures, variables and functions that operate on them.
- Formally, a mathematical entity with some operations that may be performed on it.
- We want to separate the data structure from the details of its implementation.
- You do not want a change to the data structure to entail a large rewrite of the code you have already written for it.

List ADT

- The details of the implementation are mostly hidden.
- The structure and operations can be implemented in many ways.
- The operations inside the functions do not concern the user.

List ADT

Operations that must be available on a list

- `List * create_list ()`
- `int  size (list)`
- `int  is_empty (list)`
- `int  push (list, value)` - i.e. "add_front"
- `int  append (list, value)` - optional
- `int  add_after (list, value)`
- `int  in_list (list, value)`
- `int remove_value (list, value)` - removes and frees Node
- `void print_list_all (list);` - optional (app dependant)
- `void empty_list (list)` - delete/free the inside of the list

Preliminary Setup

- We will be building the List ADT using linked list functions
  - from the previous lectures
    - `Node * add_front(Node ** head, int value)`
    - `Node * insert_value(Node * head, int value)`
    - `Node * remove_after(Node * node)`
    - `Node * find(Node * head, int value )`
    - `Node * find_prev (Node * head, int value)`
    - `void print_list(Node * head)`
    - `void free_list(Node * node)`
  - new code (for the append function)
    - `Node * find_last(Node * head)`

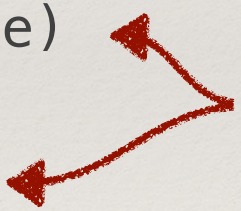
Preliminary Setup

- We will be building the List ADT using linked list functions
  - Assume all these functions are in following files
    - `linked_list_functions.c`
      - holds the source code
    - `linked_list_functions.h`
      - holds the signatures
  - `void free_list(Node * node)`
- new code (for the append function)
  - `Node * find_last(Node * head)`

Preliminary Setup

linked_list_functions.h

```
typedef struct {
   int value;                  /* contents                */
   Node * next;                /* node pointer            */
} Node;


Node * add_front(Node ** head, int value)
...
void free_list(Node * node)
```



public signatures

Preliminary Setup

- before getting to the List ADT, some of the link_list functions need to be modified or created
  - modified to add guard code for edge cases (first two) or increase speed (find_prev)
    - `Node * add_front(Node ** head, int value)`
    - `Node * insert_value(Node * head, int value)`
    - `Node * find_prev (Node * head, int value)`
  - new code (for the append function)
    - `Node * find_last(Node * head)`

Modified Link_list Functions

linked_list_functions.c

```
Node * add_front(Node ** head, int value){
    Node *new_node = malloc(sizeof(Node));
    if (new_node != NULL) {
        new_node->num = value;
        new_node->next = *head;
        *head = new_node;
    }
    return new_node
}
```

```
typedef struct {
   int value;
   Node * next;
} Node;
```

Modified Link_list Functions

linked_list_functions.c

```
Node * insert_value(Node * prev, value) {
    Node * new_node = malloc(sizeof(Node));
    if (new_node != NULL) {
        new_node->value = value;
        new_node->next = prev->next;
        prev->next = new_node;
    }
    return new_node;
}
```

```
typedef struct {
   int value;
   Node * next;
} Node;
```

Modified Link_list Functions

linked_list_functions.c

```
Node * find_prev (Node * head, int value) {
    Node * node = head, * prev = NULL,
    while (node != NULL && node->value != value) {
        prev = node;
        node = node->next
    }
    return prev;
}
```

combining find(head, value)
and find_before(head, node),
so the linked list only needs to be searched once

```
typedef struct {
   int value;
   Node * next;
} Node;
```

New Function for Link List

linked_list_functions.c

```
Node * find_last(Node * node) {
    if (node != NULL) {
        while (node->next != NULL) {
            node = node->next
        }
    }
    return node;
}
```

Added for append(list, value))

```
typedef struct {
   int value;
   Node * next;
} Node;
```

List Type Information

list_adt.h

list_adt.h

```
#define SUCCESS 1
#define FAILURE 0

typedef struct {
   Node * head;              /* pointer to first node */
   int size;                 /* number of list nodes  */
} List;

List * create_list ();
...
void empty_list (List * list);
```

public signatures

List ADT Interface

Operations that must be available on a list

- `List * create_list ()`
- `int  size (list)`
- `int  is_empty (list)`
- `int  push (list, value)`
- `int  add_after (list, value)`
- `int  append (list, value)`
- `int  in_list (list, value)`
- `int remove_value (list, value)`
- `void print_list_all (list);`
- `void empty_list (list)`

List ADT Interface

list_adt.c

```
List * create_list () {
   List * new_list = malloc(sizeof(List));
   if (new_list == NULL) {
      new_list->head = NULL;
      new_list->size = 0;
   }
   return new_list
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int is_empty (List * list) {
   return list->head == NULL;
}
```

```
int size (List * list) {
   return list->size;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int push(List * list, int value) {
   Node * node = add_front(&(list->head), value);
   int result = (node != NULL)
   if (result == SUCCESS)
      list->size++;
  return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int add_after(List * list, int prev_value, int value) {
    int result = FAILURE;
    Node * prev = find_value(list, prev_value);
    if (prev != NULL)
        result = (insert_value(prev, value) != NULL);
    if (result == SUCCESS)
        list->size++;
   return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int append (List * list, int value) {
    return  is_empty(list)  ?  push(list, value)
                            :  append_list(list, value);
}


int append_list (List * list, int value) {
    int result = FAILURE;
    Node * last = find_last(list->head, value);
    if (last != NULL)
          result = (insert_value(last, value) != NULL);
    if (result == SUCCESS)
        list->size++;
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int append (List * list, int value) {
    return  is_empty(list)  ?  push(list, value)
                            :  append_list(list, value);
}


int append_
    int res
    Node * last = find_last(list->head, value);
    if (last != NULL)
          result = (insert_value(last, value) != NULL);
    if (result == SUCCESS)
        list->size++;
    return result;
}
```

append_list is a "private" function

- signature is not placed in header
- although it is placed at top of single_end_list.c

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int remove_value(List * list, int value) {
    Node * head = list->head, * removed = NULL;
    if (head != NULL) {
        removed = (head->value == value)
                    ?   remove_head(list);
                    :   remove_within(list, value);
        if (removed != NULL) {
            free(removed);
            list->size--;
        }
    }
    return (removed != NULL);
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

"private" functions list_adt.c

```
Node * remove_head (List * list) {
    Node * head = list->head;
    list->head = head->next = NULL;
    return head;
}

Node *  remove_within (List * list, int value) {
    Node * removed = NULL;
    Node * prev = find_prev(list->head, value);
    if (prev != NULL)
        removed = remove_after(prev);
    return removed;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c

```
int in_list(List * list, int value) {
   return find(list->head, value) != NULL;
}
```

```
void print_list_all(List * list) {
  print_list(list->head);
}
```

```
void empty_list(List * list) {
   free_list(list->head);
   list->head == NULL;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Using the List ADT

```
#include “list_adt.h"

void main()
   List * list = create_list();
   push(list, 7);
   push(list, 10);
   append(list, 2);
   append(list, 8);
   print_list_all(list);
   add_after(list, 2, 13);
   node = remove_value(list, 2);
   print_list(list);
   empty_list(list);
   free(list);
}
```

< 10, 7, 2, 8 >

< 10, 7, 13, 8 >

Using the List ADT

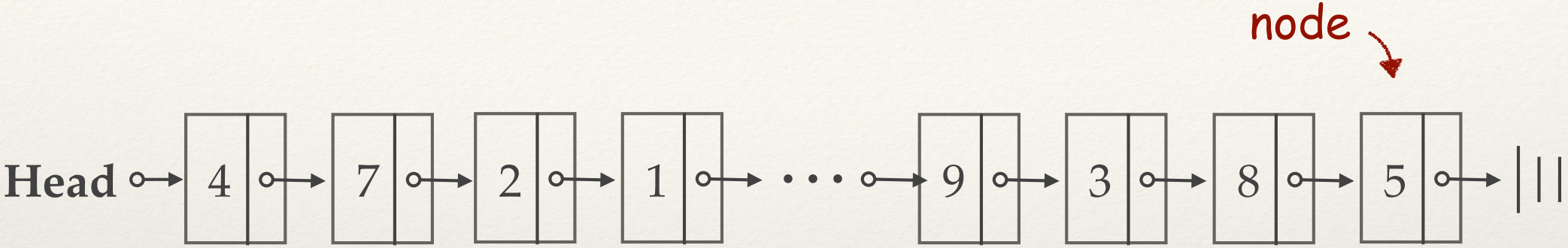
```
#include “list_adt.h"

void main()
    List * li
    push(list
    push(list
    append(li
    append(li
    print_lis                                    7, 2, 8 >
    add_after
    node = remove_value(list, 2);
    print_list(list);                  →    < 10, 7, 13, 8 >
    empty_list(list);
    free(list);
}
```

What if we are adding nodes to the end (appending) constantly

e.g. if we are using a List ADT to implement a QUEUE?

List ADT Interface

list_adt.c

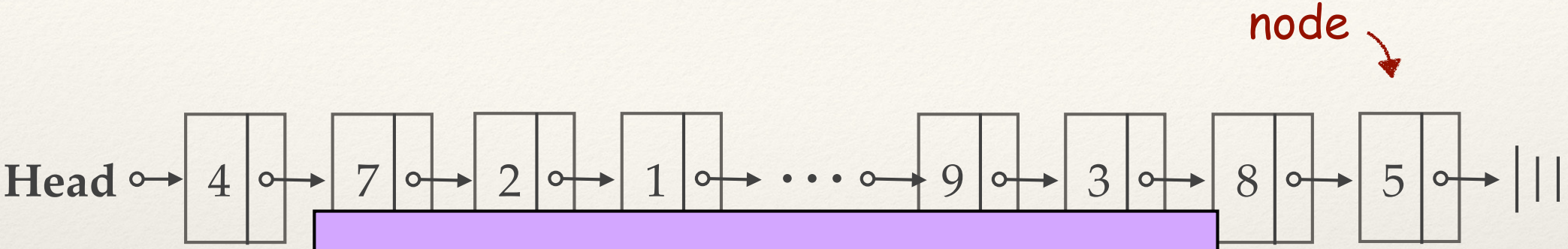



```
int append_list (List * list, int value) {
    int result = FAILURE;
    Node * last = find_last(list->head, value)
    if (last != NULL)
          result = (insert_value(last, value) != NULL);
    if (result == SUCCESS)
        list->size++;
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

list_adt.c



It would be faster if we had a pointer to the tail so we didn't need to search for it.

```
int append_li
    int resu
    Node * la
    if (last
          result = (insert_value(last, value) != NULL);
    if (result == SUCCESS)
        list->size++;
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

with tail pointer

```
#define SUCCESS = 1
#define FAILURE = 0

typedef struct NODE {
   int value;                  /* contents                */
   struct NODE * next;         /* node pointer            */
} Node;

typedef struct {
   Node * head;                /* pointer to first node */
   Node * tail;                /* pointer to last node  */
   int size;                   /* number of list nodes  */
} List;

List * create_list ();
...
void empty_list (List * list);
```

List ADT Interface

with tail pointer

```
List * create_list () {
   List * new_list = malloc(sizeof(List));
   new_list->head = NULL;
   new_list->tail = NULL;

   new_list->size = 0;
   return new_list
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

head pointer only

```
int push(List * list, int value) {
   Node * node = add_front(&(list->head), value);
   int result = (node != NULL)
   if (result == SUCCESS)
      (list->size)++;
  return result;
}
```

Single ended version

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

with tail pointer

```
int push(List * list, int value) {
   int was_empty = is_empty(list);
   Node * node = add_front(&(list->head), value);
   int result = (node != NULL)
   if (result == SUCCESS) {
      list->size++;
      if (was_empty)
          list->tail = list->head;
   }
  return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Double ended version

List ADT Interface

head pointer only

```
int add_after(List * list, int prev_value, int value) {
    int result = FAILURE;
    Node * prev = find_value(list, prev_value);
    if (prev != NULL)
        result = (insert_value(prev, value) != NULL);
    if (result == SUCCESS)
        list->size++;
   return result;
}
```

Single ended version

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

with tail pointer

```
int add_after(List * list, int prev_value, int value) {
    int result = FAILURE;
    Node * prev = find_value(list, prev_value);
    if (prev != NULL)
        result = (insert_value(prev, value) != NULL);
    if (result == SUCCESS) {
        list->size++;
        if (list->tail == prev)
            list->tail = prev->next;
   }
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Double ended version

List ADT Interface

"private" functions list_adt.c

```
Node * remove_head (List * list) {
    Node * head = list->head;
    list->head = head->next = NULL;
    return head;
}

Node *  remove_within (List * list, int value) {
    Node * removed = NULL;
    Node * prev = find_prev(list->head, value);
    if (prev != NULL)
        removed = remove_after(prev);
    return removed;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Single ended version

List ADT Interface

"private" functions list_adt.c

```
Node * remove_head (List * list) {
    Node * head = list->head;
    list->head = head->next = NULL;
    return head;
}

Node *  remove_within (List * list, int value) {
    Node * removed = NULL;
    Node * prev = find_prev(list->head, value);
    if (prev != NULL) {
        removed = remove_after(prev);
        if (list->tail == removed)
            list->tail = prev;
    }
    return removed;
}
```

Double ended version

```
typedef struct {
   Node * head;
   int size;
} List;
```

List ADT Interface

head pointer only

```
int append (List * list, int value) {
    return  is_empty(list)  ?  push(list, value)
                            :  append_list(list, value);
}


int append_list (List * list, int value) {
    int result = FAILURE;
    Node * last = find_last(list->head, value);
    if (last != NULL)
          result = (insert_value(last, value) != NULL);
    if (result == SUCCESS)
        list->size++;
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Single ended version

List ADT Interface

with tail pointer

```
int append (List * list, int value) {
    return  is_empty(list)  ?  push(list, value)
                            :  append_list(list, value);
}


int append_list (List * list, int value) {
    int result = insert_value(list->tail, value) != NULL;
    if (result == SUCCESS) {
        list->size++;
        list->tail = list->tail->next;
    }
    return result;
}
```

```
typedef struct {
   Node * head;
   int size;
} List;
```

Double ended version

Using the List ADT

```
#include "list_adt.h"

void main()
   List * list = create_list();
   push(list, 7);
   push(list, 10);
   append(list, 2);
   append(list, 8);
   print_list_all(list);
   add_after(list, 2, 13);
   node = remove_value(list, 2);
   print_list(list);
   empty_list(list);
   free(list);
}
```

< 10, 7, 2, 8 >

< 10, 7, 13, 8 >

No Change to the code!