

Chapter 15

Linked Lists

15.1 Motivation

- Lists are a datastructure that can grow
 - Not necessary to know how many elements are stored in a list
- When inserting an element into an array, all elements behind it must move to the right.
- When removing an element from an array, all elements behind it must move to the left.

Array: [2, 4, 6, 3, 7, _, _]

Inserting 5 after 4:

Shift: [2, 4, _, 6, 3, 7, _]

Set: [2, 4, 5, 6, 3, 7, _]

15.2 Lists

- In lists, only the first element is accessible (=head).
- The head contains a link to the tail of the list.
 - Very flexible datastructure.

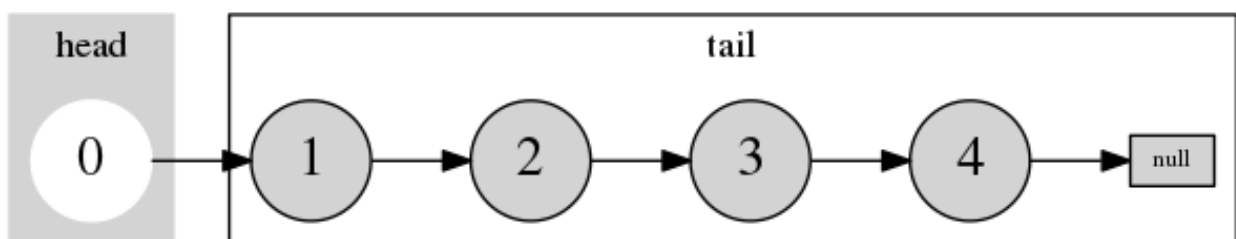


Figure 15.1:

- To obtain the third element, one needs to start from the head and go to the second element that itself links to the third element.
- The end of a list is marked with a special node (*null*).

15.3 Recursive datastructure

- Basic element of a list is a **node**.

- A node contains the value, and
- a reference to its successor.

```
struct node {
    int value;
    struct node *next;
};
```

15.4 Example: A list containing one single element

- Individual nodes must be allocated using `malloc`.

```
struct node *head = malloc(sizeof(struct node));
head->value = 0;
head->next = NULL; // marks the end of the list.
```

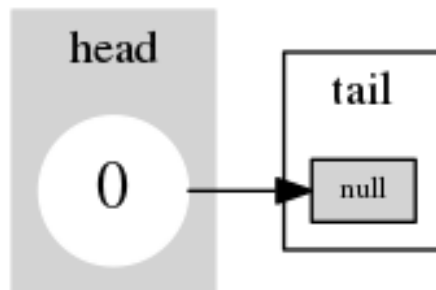


Figure 15.2:

15.5 Adding an element to the beginning

- The old head will be appended on a new node that will be the new head.

```
// head from before

struct node *n = malloc(sizeof(struct node));
n->value = 0;
n->next = head; // the old head becomes the tail
head = n; // n is the new head.
```

15.6 As a procedure

```
struct node *add_first(int value, struct node *old_head) {
    struct node *new_head =
        (struct node *) malloc(sizeof(struct node));
    new_head->element = value;
    new_head->next = old_head;
    return new_head;
}
```

- Procedure returns the new head.

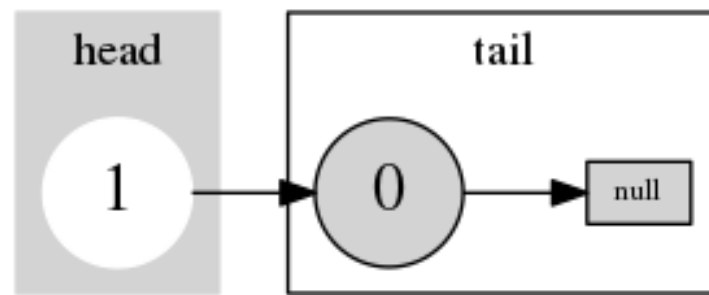


Figure 15.3:

15.7 Removing an element from the beginning

- The second element will be the new head. The old head is erased.

```
// head from before

struct node *n = head->next; // store second element
free(head);                 // the new head is the second element.
head = n;                   // delete the old head.
```

15.8 As a procedure

```
struct node *remove_first(struct node *old_head) {
    struct node *new_head = old_head->next;
    free(old_head);        // delete the old head.

    // return the new head (which was the second element)
    return new_head;
}
```

15.9 Iterate list

- Using a loop:
 - Store `head` in a variable
 - Then advances to the next element in a loop.
- Using recursion:
 - Call recursive function on rest of list

15.10 Size of a list (while-loop)

```
// Size of a list
int size(struct node *head) {
    int s = 0;
    struct node *n = head;
```

```
while(n != NULL) {  
    s++; // one more element  
    n = n->next; // go to next element.  
}  
  
return s;  
}
```

15.11 Size of a list (for-loop)

```
// Size of a list  
int size(struct node *head) {  
    int s = 0;  
  
    for(struct node *n = head; n != NULL; n = n->next) {  
        s++; // one more element  
        n = n->next; // go to next element.  
    }  
  
    return s;  
}
```

15.12 Size of a list (recursion)

- Recursive definition usually shorter and more readable but additional complexity of recursive call.

```
// Size of a list  
int size_rec(struct node *head) {  
    return 1 + size(head->next);  
}
```

15.13 Exercises

- Write a program that calculates
 - the sum of all elements in a list.
 - the maximum/minimum of all elements in a list.
- Write a program that finds the first occurrence of a certain element.