a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

Linked lists

Unordered arrays

- Arrays are easy to code and to visualise, and offer rapid access to elements using the `[]` operator
  - e.g. `printf("salary is $%.2f\n", staff[85].salary);`

- Consider an array with a capacity of n
  - Note that capacity may be different from the number of items stored in the array

- What is the (best) worst-case time complexity of inserting an item into an *unordered* array without holes, when order does not matter?
  - Assume that the number of currently stored items is known (n)

arr.

0	1	2	3	4	5	6	7	8			
18	9	26	41	12	75	37	60	15			

↑

Insert 15?

$n = 8$

$$\left.\begin{array}{l} arr[n] = 15; \\ n{+}{+}; \end{array}\right\} O(1)$$

A. $O(1)$

B. $O(n)$

C. both A and B

D. neither A nor B

E. 🤷‍♀️

iClicker 06.2

Unordered arrays

- What is the worst-case time complexity of removing an item from an *unordered* array without holes, when order does not matter?
  - assume that the index (i) of the item to remove is known, as well as the number of items stored (n)

	0	1	2	3	4	5	6	7	8			
arr	~~18~~ 60	9	26	41	12	75	37	60				

Remove 18 (from index 0)?

$n = \cancel{8}\ 7$

arr[0] = arr[--n];

A. $\boxed{}$ $O(1)$

B. $O(n)$

C. both A and B

D. neither A nor B

E. 😅

iClicker 06.3

Ordered arrays

- What is the worst-case time complexity of inserting an item into an *ordered* array without holes, when order *does* matter?
  - Assume that the number of currently stored items is known (n)

9	12	18	26	37	41	60	75				

15

Insert 15?

A. $O(1)$

B. $O(n)$

C. both A and B

D. neither A nor B

E. 😐

iClicker 06.4

Ordered arrays

- What is the worst-case time complexity of removing an item from an *ordered* array without holes, when order *does* matter?
  - assume that the index (i) of the item to remove is known, as well as the number of items stored (n)

9	12	~~18~~	26	37	41	60	75				

Remove 18 (from index 2)?

A. $O(1)$

B. $O(n)$

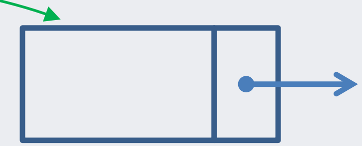
C. both A and B

D. neither A nor B

E. 😱

Linked list nodes

- A linked list is a dynamic data structure that consists of nodes linked together
- A *node* is a data structure that contains
  - data
  - the location (address) of the next node



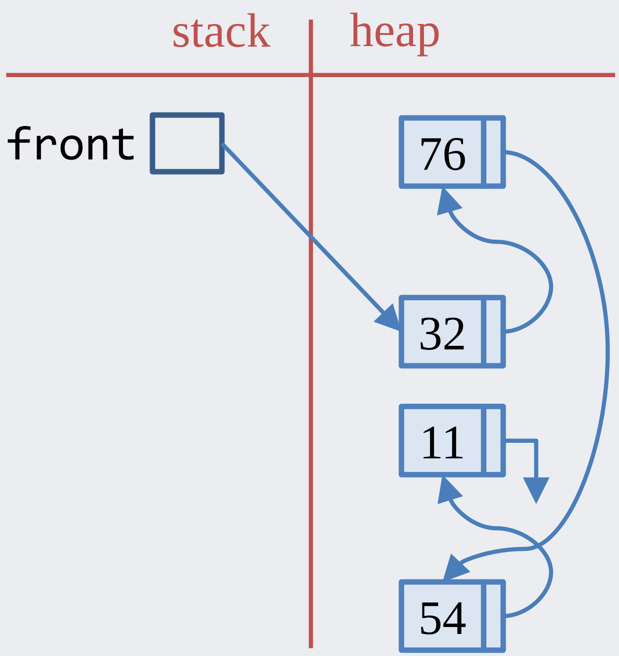
- The data portion of a node can contain one or more items or structures

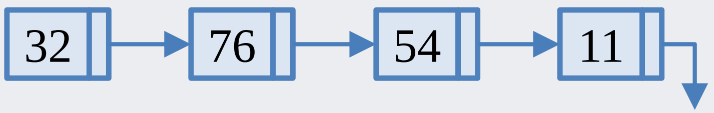
```c
typedef struct node {
  int data;
  struct node* next;
} node;
```

```c
typedef struct player {
  int jersey_number;
  char* name;
  struct player* next;
} player;
```

Linked lists

- A linked list is a chain of nodes, where each node indicates where in (heap) memory the next item can be found

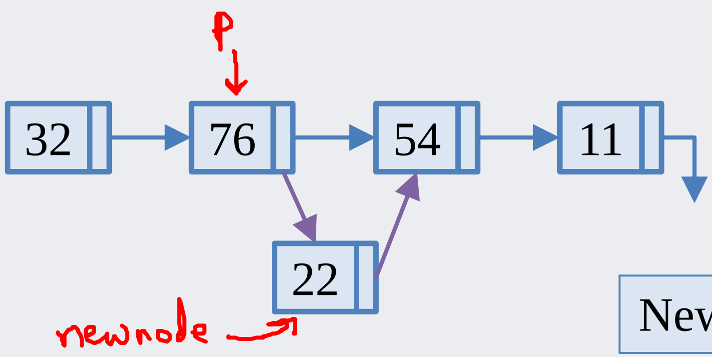




Modifying linked lists

- Inserting an item into a linked list

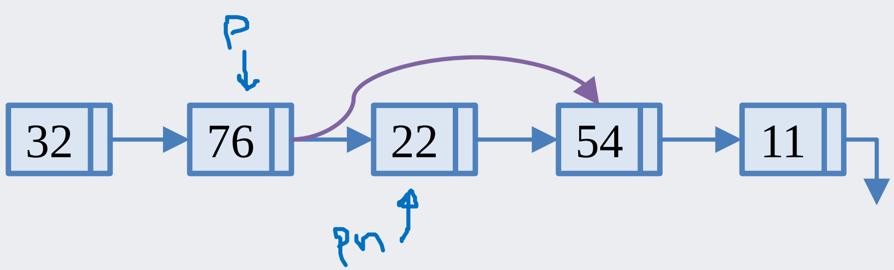
```
struct node* newnode = (struct node*) malloc(sizeof(struct node));
newnode->data = 22;
newnode->next = p->next;
p->next = newnode;
```

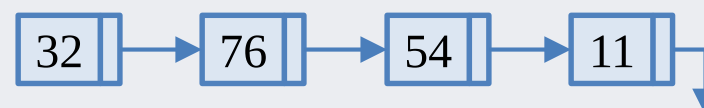


New nodes created using `malloc()` as needed

- Removing an item from a linked list

```
struct node* pn = p->next;
p->next = pn->next;
free(pn);
```






```
p->next = p->next->next;
```

memory leak

Removed nodes must be deallocated using `free()`!

Linked list dis/advantages

- Advantages
  - Linked list is constructed from nodes in the heap, can be added as needed, and removed at runtime
  - The size of the list does not need to be "guessed" ahead of time

- Disadvantages
  - To access a particular node (starting from the front of the list), may need to traverse the list to reach – $O(n)$
  - Linked list nodes have additional overhead to store pointers
  - Harder to program, debug, and test

Linked list traversal

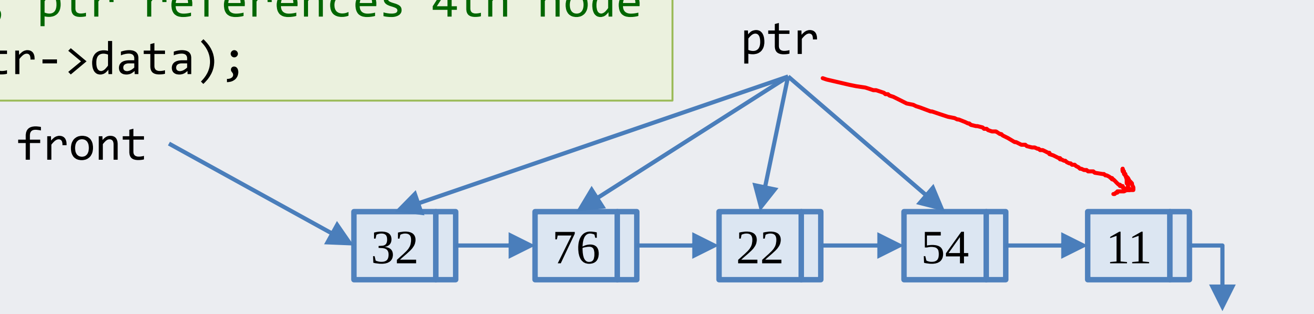
- Traversal through a linked list can be done with a node pointer variable
  - e.g. to access the 4th element in a list:

null-terminated singly-linked list

```
// assuming we have node* front
node* ptr = front;
int i;
for (i = 0; i < 4; i++) {
  ptr = ptr->next;
}
// exited loop, ptr references 4th node
printf("%d", ptr->data);
```

while (ptr != NULL)

while (ptr->data != key && ptr != NULL)



Example

Linked list for hockey players

- Recall: nodes defined as structures with attributes, and a pointer to the next node

```
typedef struct Player {
  int jersey_number;
  char* name;
  struct Player* next;
} Player;
```

local Player variable

Note that head_list1 only needs a pointer type, so syntactically we could assign it a pointer to a local Player variable

(not how we usually make nodes)

```
int main() {
  Player* head_list1 = NULL;
  Player* head_list2 = NULL;
  Player gretzky = {99,
                    "Wayne Gretzky", NULL};

  // example 1
  head_list1 = &gretzky;
  displayList(head_list1);

  // example 2
  head_list2 = insertAtHead(head_list2,
                            35, "Thatcher Demko");
  head_list2 = insertAtHead(head_list2,
                            83, "Bo Horvat");
  displayList(head_list2);
}
```

iCliker 06.5

insertAtHead

- `insertAtHead` should insert a new node at the head of a (possibly empty) list, and return the node as the new head
- Will this code work?

attach to this list

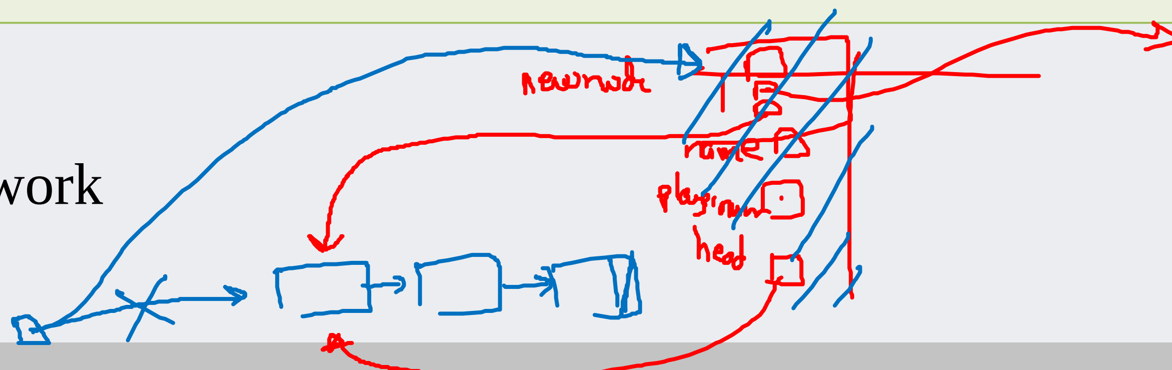
```
Player* insertAtHead(Player* head, int player_number, char* player_name) {
  Player new_node;

  new_node.jersey_number = player_number;
  new_node.name = player_name;
  new_node.next = head; // point to current head of list

  printf("Node was added.\n\n");
  return &new_node; // the new node becomes the new head of list
}
```

local variable

A. Yes, this is fine
B. No, this will not work
C. I have no idea



Hockey example continued

insertAtHead

- `insertAtHead` should insert a new node at the head of a (possibly empty) list, and return the node as the new head
  - is this better?

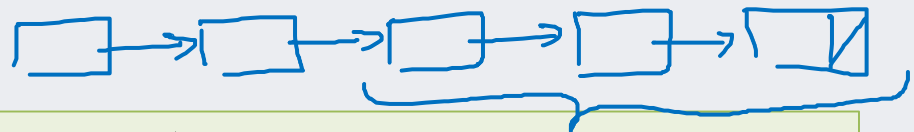
```
Player* insertAtHead(Player* head, int player_number, char* player_name) {
  Player* new_node;
  new_node = (Player*) malloc(sizeof(Player));

  new_node->jersey_number = player_number;
  new_node->name = player_name;
  new_node->next = head; // point to current head of list

  printf("Node was added.\n\n");
  return new_node; // the new node becomes the new head of list
}
```

Hockey example continued

- A function to iterate through a (possibly empty) linked list, starting from the head of the list, and printing out information from each node



```c
void displayList(Player* node) {
  int k = 0;

  while (node) { // loop breaks when node becomes NULL
    printf("Node %d is: %s, Jersey number %d\n",
           k, node->name, node->jersey_number);
    node = node->next;
    k++;
  }

  printf("There are %d node(s) in the list.\n\n", k);
}
```

See hockey_players_linked_list_V1.c

Comparison of worst case complexities

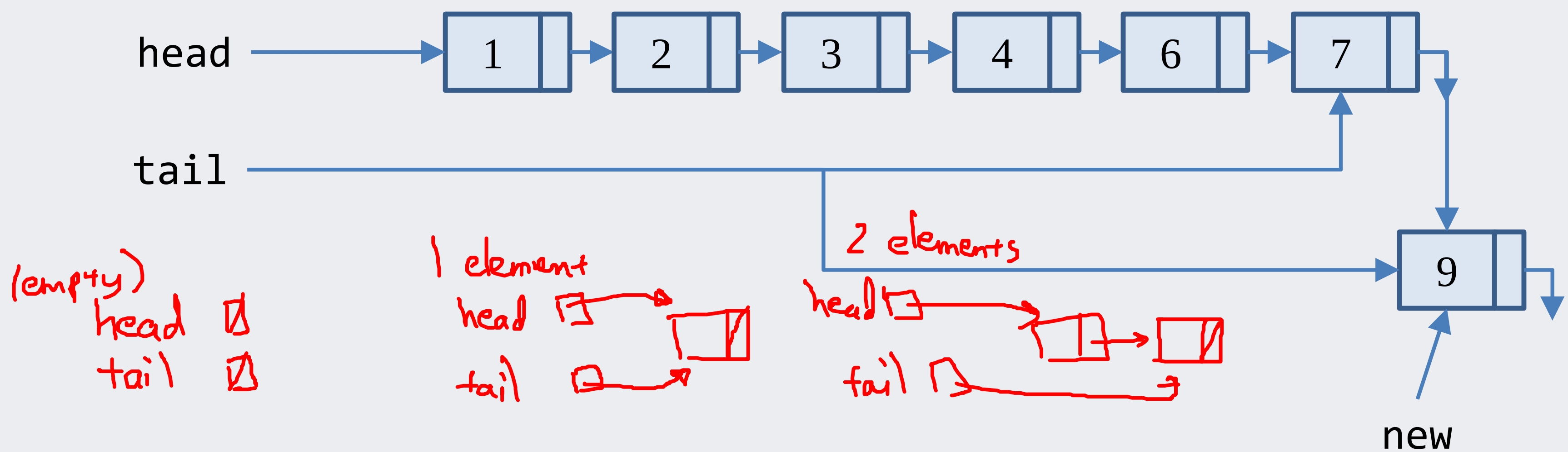
- Assume we know the number of entries in the arrays
  - also assume current position (in lists) is known (getting that position may cost up to $O(n)$)

Operation	Array (unordered)	Array (ordered)	Linked list (unordered)	Linked list (ordered
Insert at front	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Insert at back, using head ptr	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Insert after current position				
Search for a value	$O(n)$ linear search	$O(\log n)$ binary search	$O(n)$	linear search $O(n)$
Remove at current position				

- Notice how the operations at the end of the list have (relatively) poor complexity, involving a complete traversal of the list
  - we can give ourselves a tail pointer, for very little additional overhead

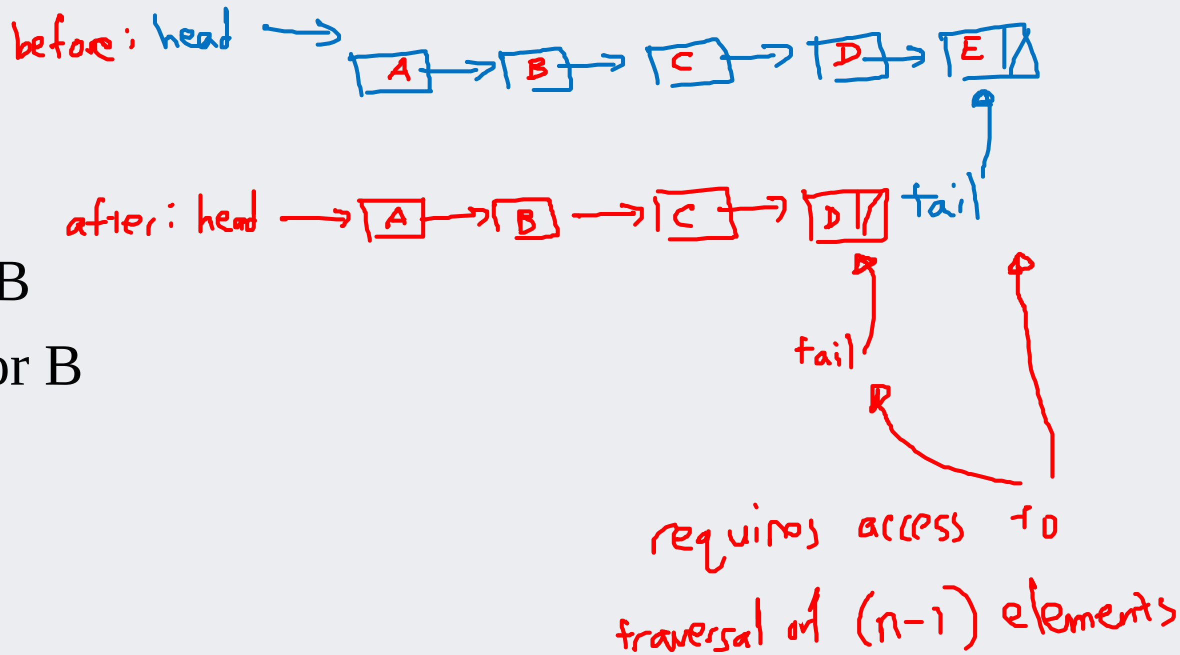
null-terminated singly-linked list with head and tail pointers

how the list ends / links / movement / where do we have access



iClicker 06.6

- Consider a singly-linked list, with head and tail pointers and contains n elements. What is the tightest upper bound on the complexity of removing the last element of the list?



A. $O(1)$

B. $O(n)$

C. both A and B

D. neither A nor B

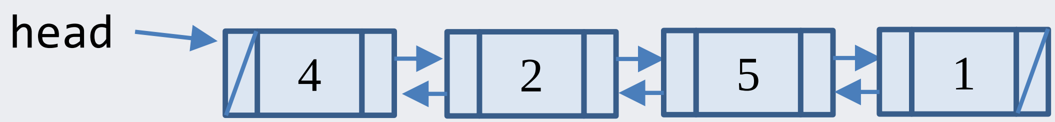
E. 

Doubly-linked list

- Node definition contains an additional pointer
  - links to previous node in the list, allows traversal or access towards the front of the list

null-terminated doubly-linked list
with head pointer

```c
typedef struct {
  int data;
  struct Node* prev;
  struct Node* next;
} Node;
```



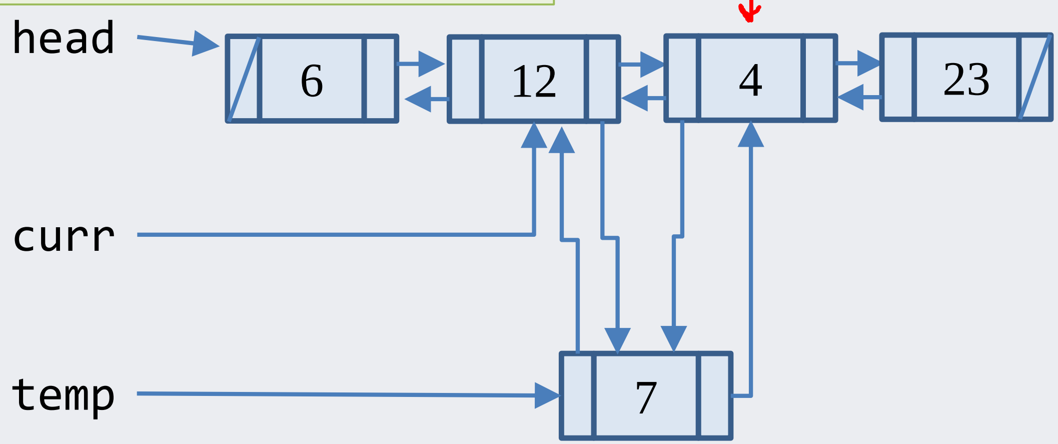
- Provides access to the previous and next nodes from a single pointer (e.g. for insertion/removal)
  – but, requires more pointer management in programming

Doubly-linked list insertion

- After some specified node

```
Node* curr, * temp;
... // move curr into place        <- curr_n = curr->next;
temp = (Node*) malloc(sizeof(Node));
temp->data = 7;
temp->prev = curr;
temp->next = curr->next;
curr->next->prev = temp; // curr_n ->prev = temp;
curr->next = temp;
```

curr_n

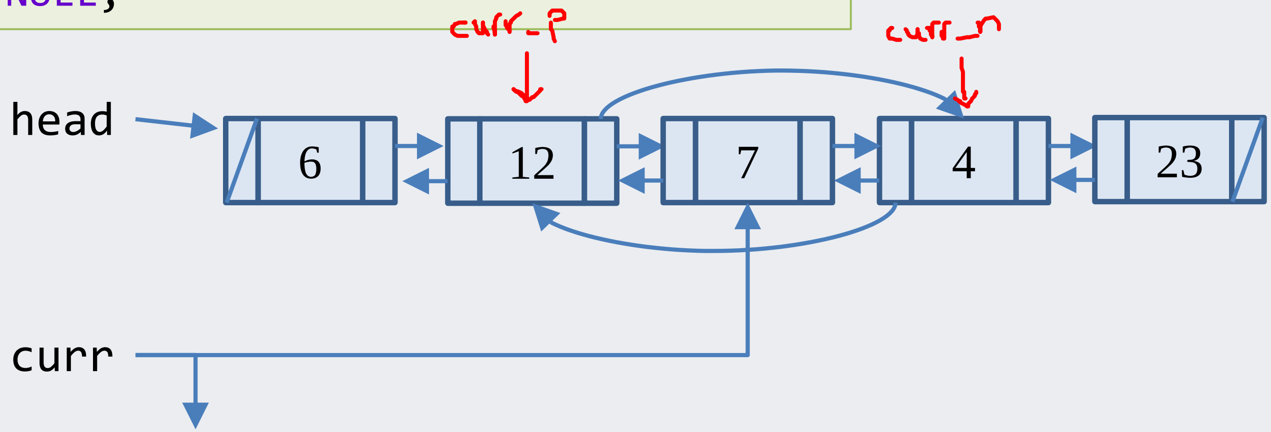


Doubly-linked list removal

- At some specified node

```
Node* curr;
... // move curr to the node to be removed
curr->next->prev = curr->prev;
curr->prev->next = curr->next;
free(curr);
curr = NULL;
```

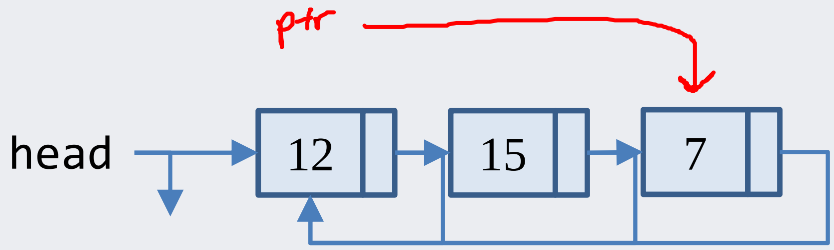
curr_p = curr -> prev;
curr_p -> next = curr_n;
curr_n -> prev = curr_p;



Circular linked lists

Singly-linked version

- The last node in the list points back to the first node



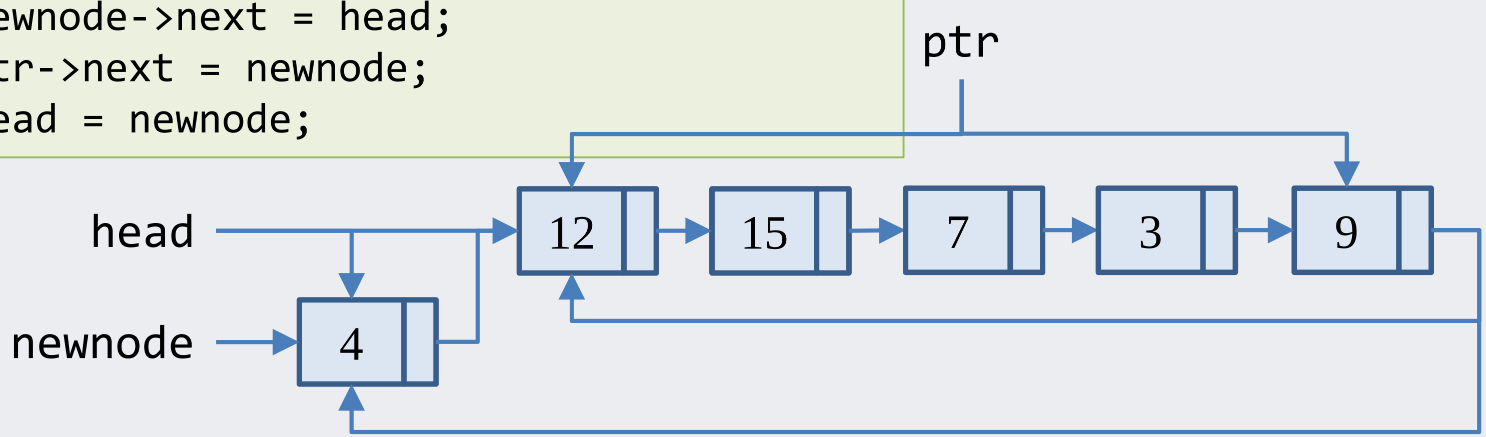
- How to check when we reach the end in a traversal?
  - address of next is the same as the address of the front
  - but still must be careful to do NULL check on empty list!

Circular singly-linked list

Insertion at head

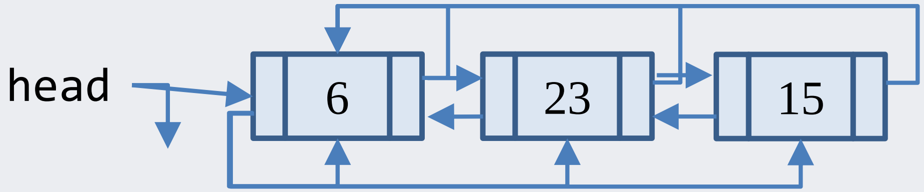
- Insertion in the middle of a circular singly-linked list is no different from inserting into a NULL-terminated list
  - what about inserting at the head?
  - need to iterate a pointer to the last node in the list!

```
Node* ptr, * newnode;
ptr = head;
while (ptr->next != head)   } O(n)
  ptr = ptr->next;
newnode = (Node*) malloc(sizeof(Node));
newnode->data = 4;
newnode->next = head;
ptr->next = newnode;
head = newnode;
```



Circular doubly-linked list

- The last node in the list points to the first node
  - and the first node points to the last node



What is the time complexity of accessing the last element of a circular doubly-linked list?

Exercise

- Write a function that inserts a node at the front of a (possibly empty) doubly-linked list, with the following signature:

```
Node* insertHead(Node* front, int value);
```

```
typedef struct {
  int data;
  struct Node* prev;
  struct Node* next;
} Node;
```

- Write a function that inserts a node at the front of a (possibly empty) *circular* doubly-linked list, with the same signature

Readings for this lesson

- Thareja
  - Chapter 6
  - Chapter 7.1 – 7.3