CPSC 259: Data Structures and Algorithms for Electrical Engineers

Linked Lists

Textbook References:

(a) Etter, 3rd edition: Chapter 7, pages 340-359
(b) Thareja (first edition) Chapter 8
(c) Thareja (second edition): Chapter 6

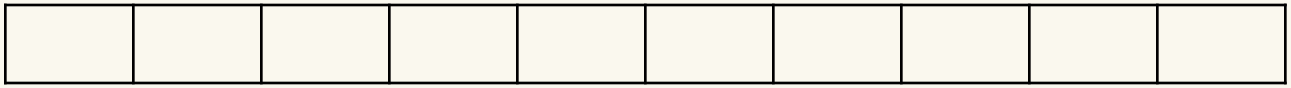
Hassan Khosravi
Borrowing many slides from Ed Knorr

Learning Goals for this Unit

- Define and use linked lists in an implementation with dynamic memory allocation.
- Traverse a node-based linked list using a loop
- Mutate a node-based linked list
- Determine the time complexities of operations on arrays and linked lists.
- Compare and contrast the implementation of a list using arrays, singly-linked lists, and doubly-linked lists in C.

Unordered Arrays

Let us begin this section by considering operations on arrays that have a given *capacity* or size denoted by *n*. Note that the capacity may be different than the current number of elements in the array.



By now, you should be very familiar with arrays. Arrays tend to be easy to code and easy to visualize. In terms of complexity, they offer fast access.

e.g., `printf("salary is $%.2f\n", staff[85].salary);`

What is the time complexity of adding an element to an array that is not currently full, when order is unimportant? *O(1)*

Unordered Arrays

- The time complexity of deleting an element from an unordered array, such that we don't leave a hole in the array (with a garbage value), is _____

  - A: O(1)
  - B: O(n)
  - C: neither
  - D: Both
  - E: I don't know

Unordered Arrays

- The time complexity of deleting the current element from an unordered array, such that we don't leave a hole in the array (with a garbage value), is _____*O(1)*_____ because:

2	4	5	6	4	4	9			

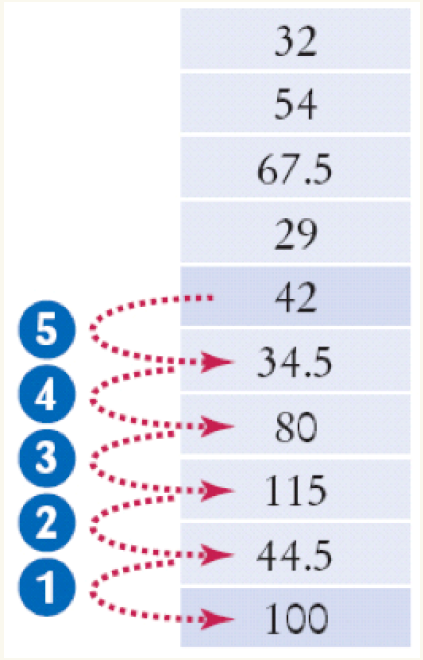
Delete 5

2	4	9	6	4	4				

Ordered Arrays

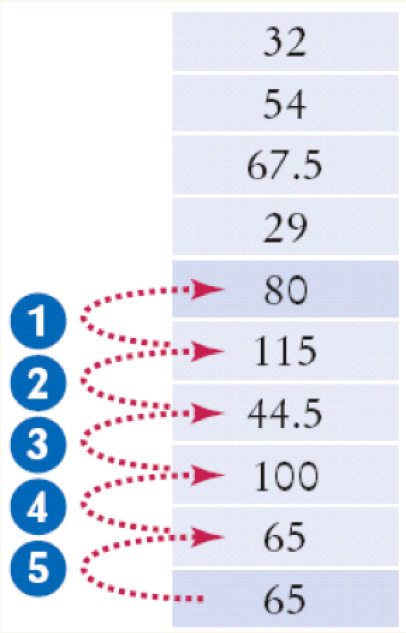
Inserting to an array without holes, (in which order matters) has time complexity?

O(n)



Deleting from an array without holes, (in which order matters) has time complexity?

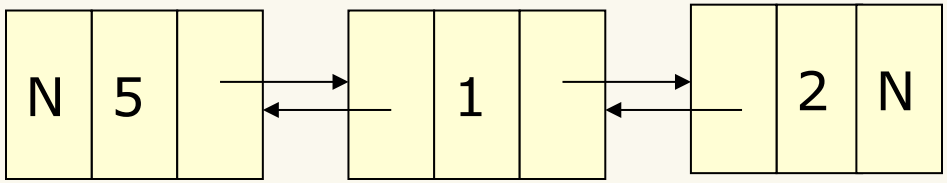
O(n)



Linked Lists

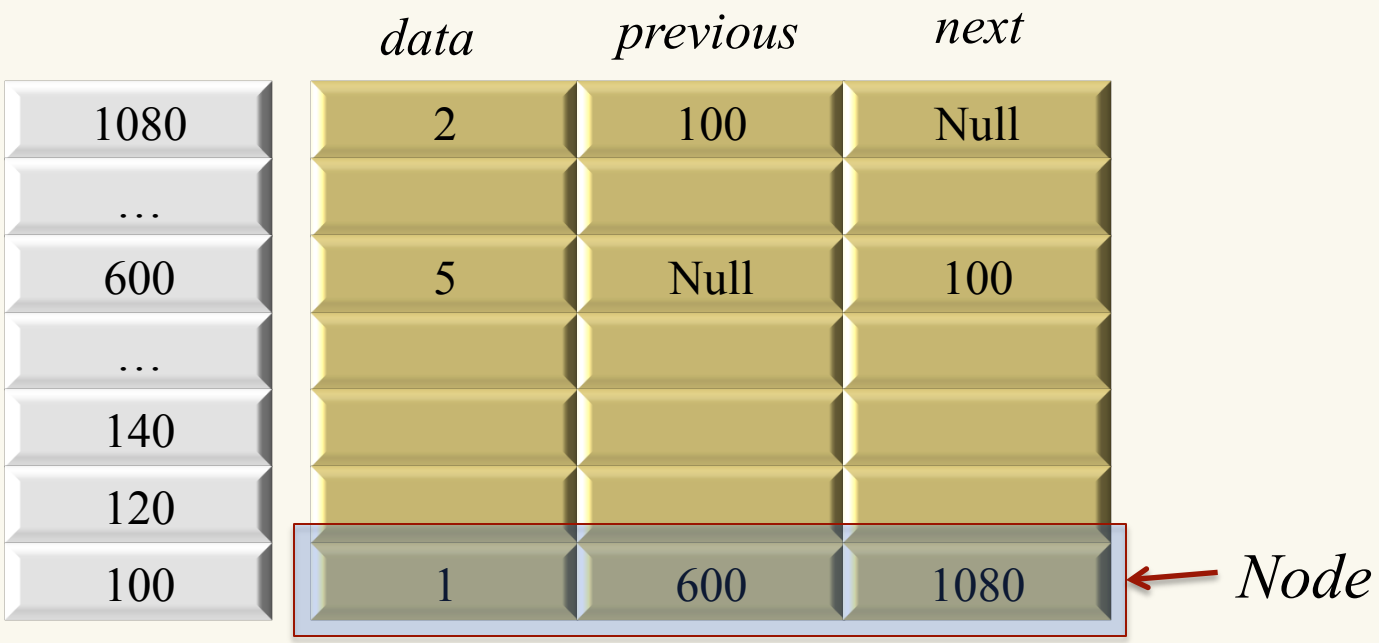
- Consider the following abstraction, picturing a short linked list:

N represents NULL

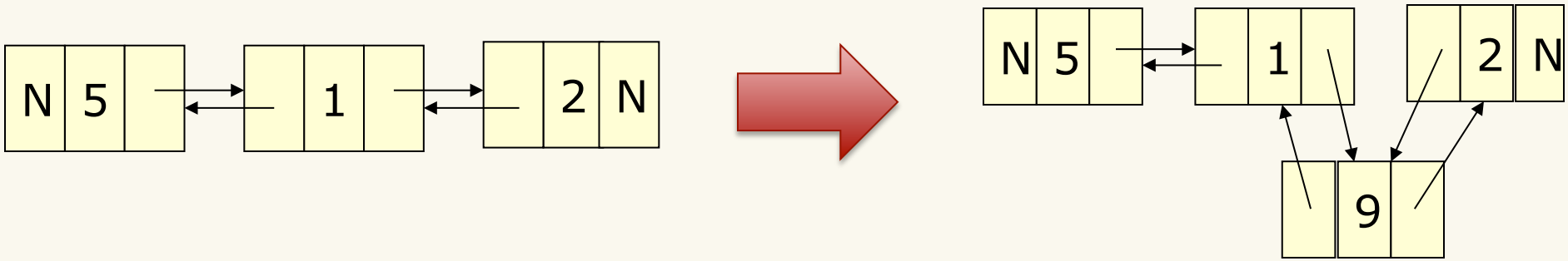


- What might it look like in memory?

`

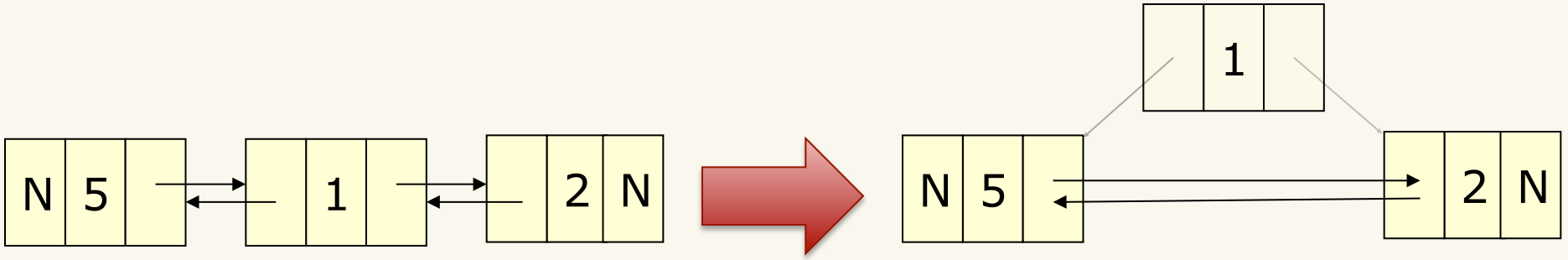


Inserting an Element to a Linked List



	data	*previous*	*next*
1080	2	140	Null
…			
600	5	Null	100
…			
140	9	100	1080
120			
100	1	600	140

Removing an Element from a Linked List



	data	*previous*	*next*	
1080	2	600	Null	
…				
600	5	Null	1080	
…				
140				
120				
100	1	600	1080	← *delete*

Website for Visualizing Data Structures

- http://visualgo.net/list.html

Linked Lists -- Advantages

- **Advantage**: Linked lists are connected "nodes" of data that we can add to, or delete from, on-the-fly. Thus, a linked list can grow or shrink "dynamically", while the program is running.
- **Advantage**: The programmer doesn't have to determine (or "guess") the size of the linked list ahead of time. If we need another node, we just get one (via `malloc`) and connect it to the list

Linked lists -- Disadvantages

- **Disadvantage**: To access a node, we may need to go through quite a bit of the linked list to find it (i.e., O(n) access). Arrays have O(1) access (e.g., arrays can get to `staff[85]` instantly; however, a linked list would have to visit 85 elements in its chain before getting to the 86th one).
- **Disadvantage**: A linked list needs more space to store the data because of the pointer field.
- **Disadvantage**: Linked lists are harder to program, debug, and test—than arrays.

Hockey Player Example:

```
struct player {
  int              jersey_number;
  char *           name;
  struct player *  next;     /* to link this node to the next node */
};
typedef struct player player;

int main(void){
  player *  head_list1 = NULL;
  player *  head_list2 = NULL;
  player    gretzky = {99, "Wayne Gretzky", NULL};
  /* Example 1 */
  head_list1 = &gretzky;
  display_list(head_list1);

  /* Example 2 */
  head_list2 = insert_at_head(head_list2, 22, "Daniel Sedin");
  head_list2 = insert_at_head(head_list2, 33, "Henrik Sedin");
  display_list(head_list2);
}
```

Hockey Player Example:

- Here is a function to insert a new node at the start of a (possibly empty) linked list:

```
player *  insert_at_head( 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 is the new head */
}
```

Clicker question

- Would this work?

```
player* insert_at_head(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 is the new head */
}
```

A: Yes this is totally fine
B: No, this would not work
C: I have no idea

Clicker question

- Would this work?

```
player* insert_at_head(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 is the new head */
}
```

A: Yes this is totally fine

B: No, this would not work

C: I have no idea

Memory is allocated on the stack and will automatically be deallocated when the function ends

Hockey Player Example:

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

```
void display_list(player * node){
  int  k = 0;

  while (node){   /* only stop 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 were %d node(s) in the list.\n\n", k);
}
```

See the hockey_players_linked_list files

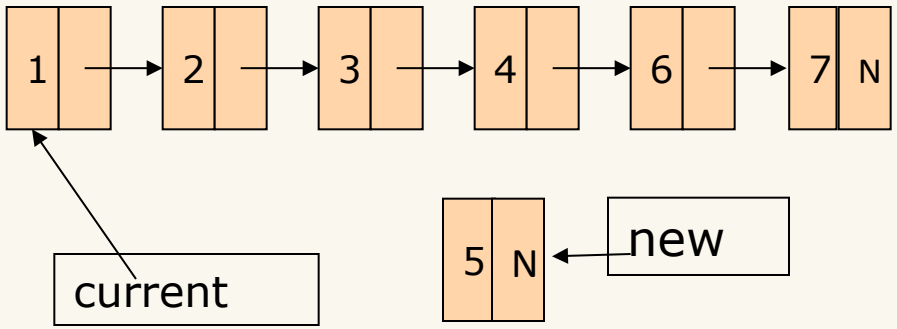
More operations on linked lists

- What if you wanted to insert/delete nodes into the middle of a list?

Clicker question (Inserting into a list)

- Consider the following linked list, and possible commands

```
W: current->next = new
X: current= current->next
Y: new->next = current->next
Z: current = new
```



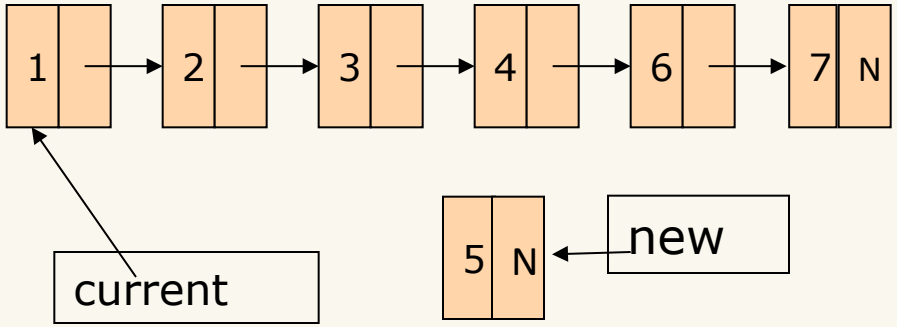
- Assuming that we would like to keep the list sorted, which of the following list of commands correctly inserts the new node into the list

A: X X X Y W
B: X X X X W Y
C: X X X W Y
D: X X X W Z Y
E: None of the above

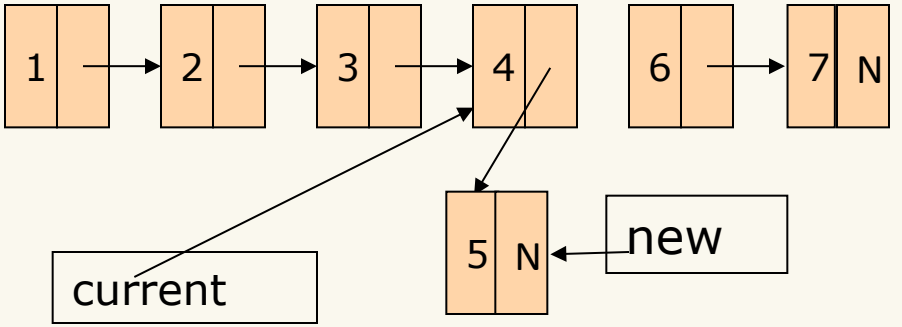
Clicker Question (answer)

- Consider the following linked list, and possible commands

```
W: current->next = new
X: current= current->next
Y: new->next = current->next
Z: current = new
```



- Assuming that we would like to keep the list sorted, which of the following list of commands correctly inserts the new node into the list



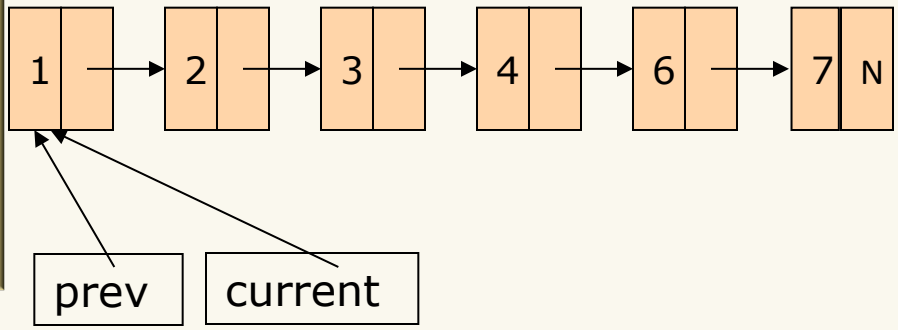
A: X X X Y W

If W is performed before Y, then the second half of the list is lost

Clicker question (deleting from a list)

- Consider the following linked list, and possible commands

```
V: current= current->next
W: prev = prev->next
X: prev->next = current->next
Y: current->next = prev->next
Z: free(current); current= NULL;
```



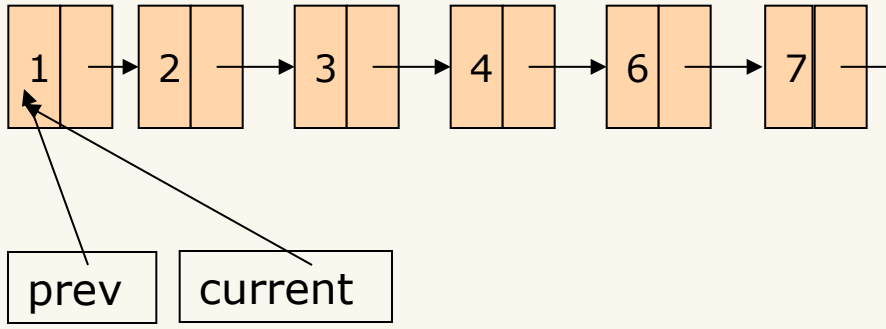
- Which one of the following list of commands correctly deletes 3 from the list

A: V W V Y Z
B: W V W X Z
C: V W V X Z
D: V V W W Y Z
E: None of the above

Clicker question (deleting from a list)

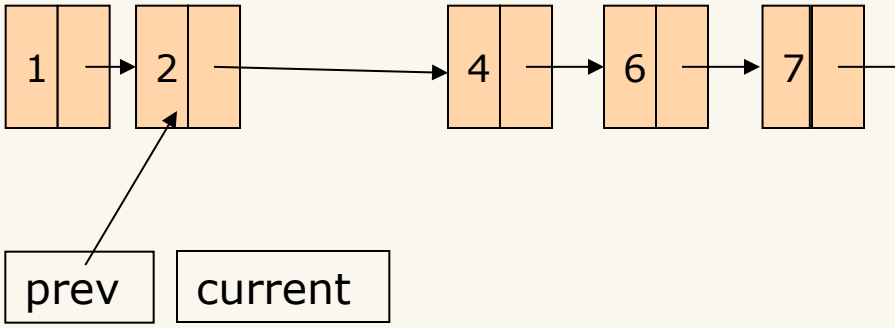
- Consider the following linked list, and possible commands

```
V: current= current->next
W: prev = prev->next
X: prev->next = current->next
Y: current->next = prev->next
Z: free(current);current= NULL;
```



- Which one of the following list of commands correctly deletes 3 from the list

C: V W V X Z



Comparison of Worst Case Complexities

- Assume we know the number of entries in the arrays.

Operation	**Array (unordered)**	**Array (ordered)**	**Linked List (unordered)**	**Linked List (ordered)**
Insert at start				
Insert at end using head ptr				
Insert after current position				
Find (search for) a value				
Delete at current position				

Comparison of Worst Case Complexities

- Assume we know the number of entries in the arrays.

Operation	**Array (unordered)**	**Array (ordered)**	**Linked List (unordered)**	**Linked List (ordered)**
Insert at start	O(1)	O(*n*)	O(1)	O(1)
Insert at end using head ptr	O(1)	O(1)	O(*n*)	O(*n*)
Insert after current position	O(1)	O(n)	O(1)	O(1)
Find (search for) a value	O(*n*)	O(lg *n*) for unique keys	O(*n*)	O(*n*)
Delete at current position	O(1)	O(n)	O(1)	O(1)

Clicker question

- If we also had a tail pointer for an ordered linked list (lowest to highest), what would be the worst-case complexity for an end-of-list insertion, if each new key was always larger than the highest current value, every time?

A. O(1)

B. O(lg n)

C. O(n)

D. O(n lg n)

E. None of the above

Clicker question (answer)

- If we also had a tail pointer for an ordered linked list (lowest to highest), what would be the worst-case complexity for an end-of-list insertion, if each new key was always larger than the highest current value, every time?

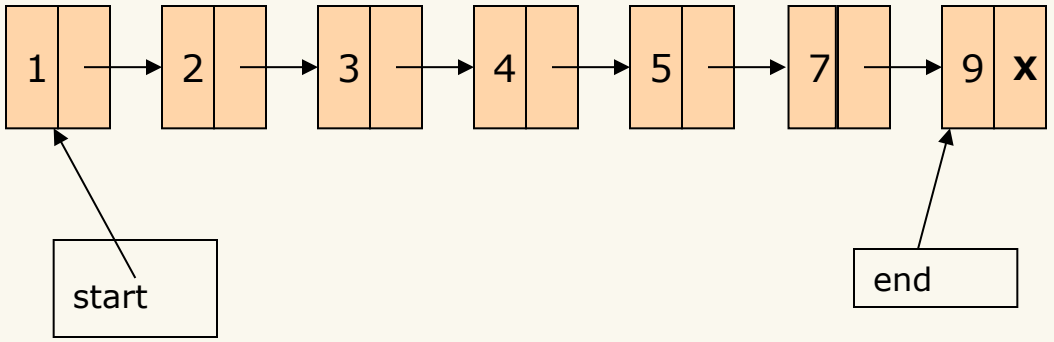
A. O(1)

B. O(lg n)

C. O(n)

D. O(n lg n)

E. None of the above



Clicker question

- If we also had a tail pointer for the ordered linked list case, what would be the worst-case complexity for inserting one new key, if new keys weren't necessarily higher than the current highest current value?

A. O(1)

B. O(lg n)

C. O(n)

D. O(n lg n)

E. None of the above

Clicker question (answer)

- If we also had a tail pointer for the ordered linked list case, what would be the worst-case complexity for inserting one new key, if new keys weren't necessarily higher than the current highest current value?

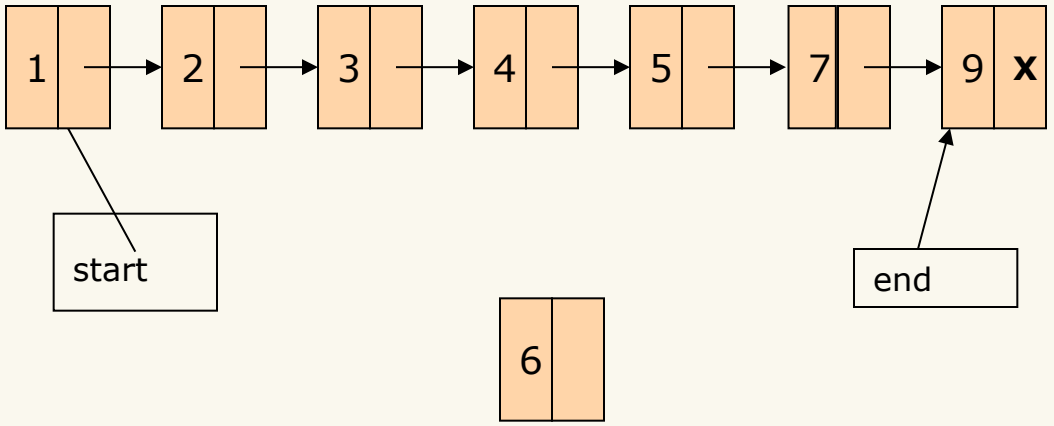
A. O(1)

B. O(lg n)

C. O(n)

D. O(n lg n)

E. None of the above



Clicker question

(same idea) If we also had a tail pointer for the ordered linked list case, what would be the worst-case complexity for inserting 100 new keys, if new keys weren't necessarily higher than the current highest current value? Choose the best answer.

A. O(1)
B. O(lg n)
C. O(100 lg n)
D. O(n)
E. O(100 n)

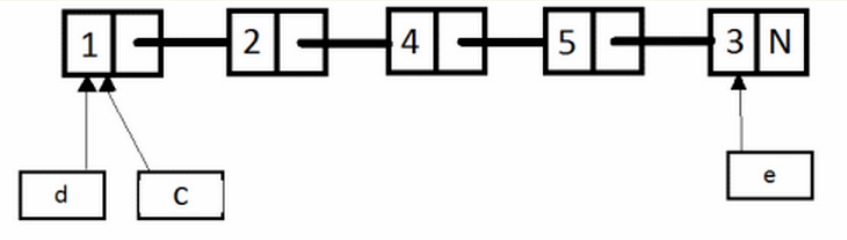
Clicker question (answer)

(same idea) If we also had a tail pointer for the ordered linked list case, what would be the worst-case complexity for inserting 100 new keys, if new keys weren't necessarily higher than the current highest current value? Choose the best answer.

A. O(1)
B. O(lg n)
C. O(100 lg n)
D. O(n)
E. O(100 n)

Clicker Question

- Consider the following linked list and possible commands. Which correctly orders the list?

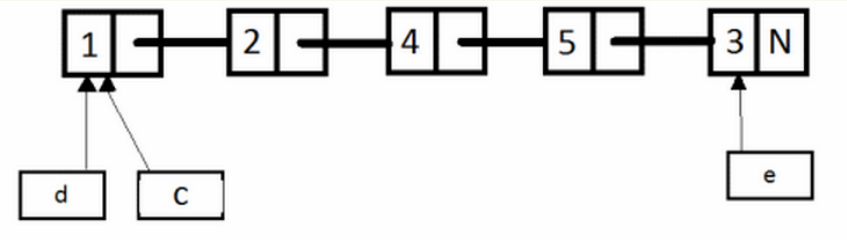


A: c = c -> next
B: d = d -> next
C: c -> next = e
D: d -> next = c
E: e -> next = d
F: d -> next = NULL
G: e -> next = NULL

A: A A B E B B B F
B: A B B C E B F
C: A C B B E B F
D: A B B C D G
E: none of the above

Clicker Question Answer

- Consider the following linked list and possible commands. Which correctly orders the list?

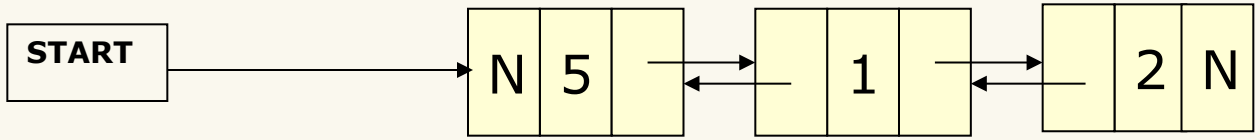


A: c = c -> next
B: d = d -> next
C: c -> next = e
D: d -> next = c
E: e -> next = d
F: d -> next = NULL
G: e -> next = NULL

A: A A B E B B B F
B: A B B C E B F
C: A C B B E B F
D: A B B C D G
E: none of the above

Doubly Linked List

- A doubly linked list or a two way linked list is a more complex type of linked list which contains a pointer to the next as well as previous node in the sequence. Therefore, it consists of three parts and not just two. The three parts are data, a pointer to the next node and a pointer to the previous node



Doubly Linked List

- In C language, the structure of a doubly linked list is given as,

```
struct node{
   struct node *prev;
   int data;
   struct node *next;
};
```

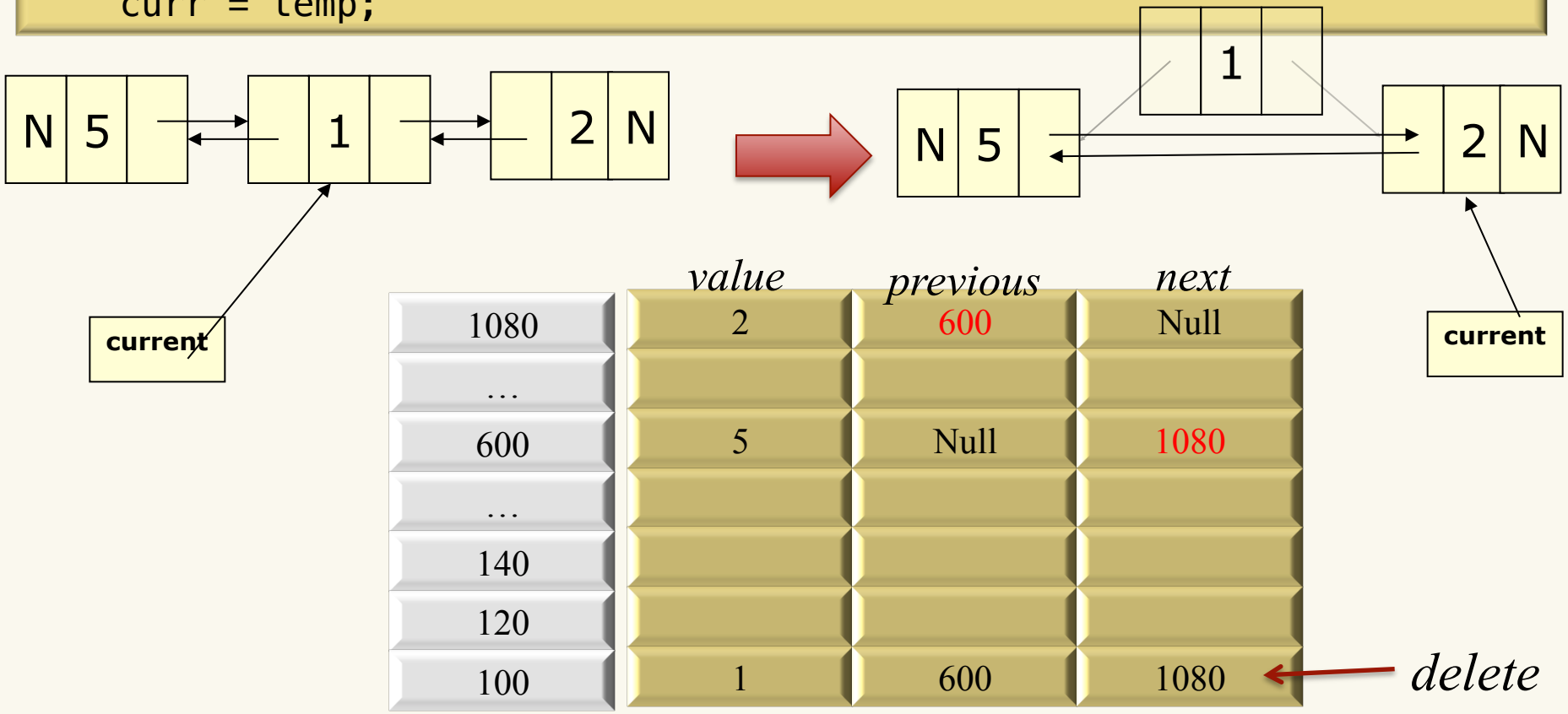
- The prev field of the first node and the next field of the last node will contain NULL. The prev field is used to store the address of the preceding node. This would enable to traverse the list in the backward direction as well.

Doubly Linked List

- Advantages:
  - Can navigate back and forth (and visit nodes in either direction) without going back to the start/end
  - Can add a node before/after the current node in O(1)
  - Can remove a node before/after the current node in O(1)
- Disadvantage
  - Requires more space
  - Harder to program

Removing from a doubly linked list

```
node  *curr, *temp;
 ...
 /* curr points to the current node;  curr is not head or tail */
 curr->prev->next = curr->next;
 curr->next->prev = curr->prev;
 temp = curr->next;          /* Why are we using "temp" here? */
 free(curr);                 /* free (deallocate) memory /*
 curr = temp;
```



Linked list exercise (1)

- Write a count() function that counts the number of times a given int occurs in a list.

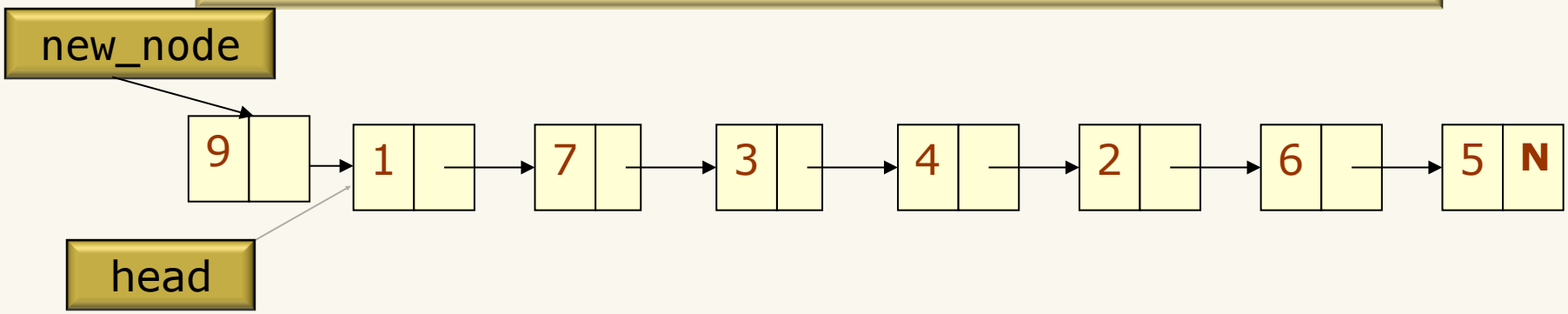
```
int count(node* head, int searchFor) {
  node* current = head;
  int count = 0;
  while (current != NULL) {
    if (current->data == searchFor)
      count++;
    current = current->next;
  }
  return count;
}
```

Linked list exercise

- Write a insertHead1 function that uses the following prototype

```
node * insertHead1(node * head, int value);
```

```
node * insertHead1(node * head, int value){
  node * newNode;
  newNode = (node*)malloc(sizeof(node));
  newNode->data = value;
  newNode->next = head;
  return newNode;
}
```

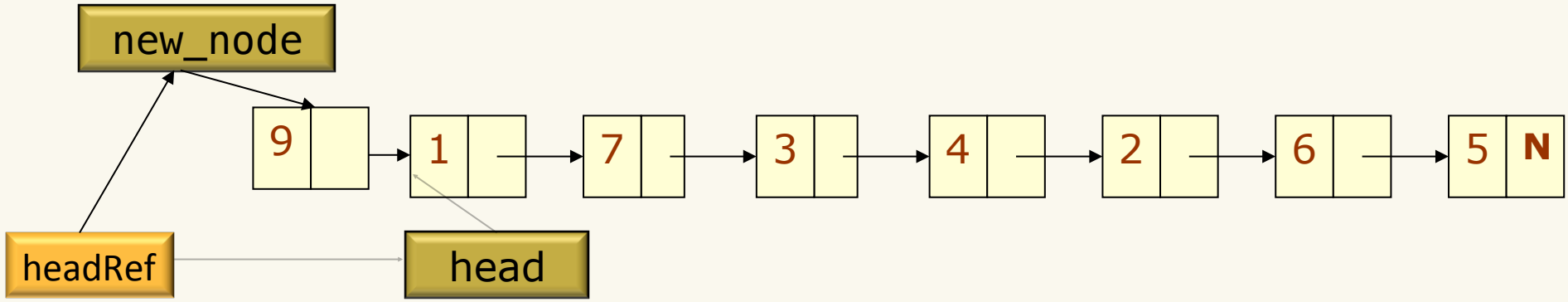


Linked list exercise

- Write a insertHead2 function that uses the following prototype

```
void insertHead2(node ** headRef, int value);
```

```
void insertHead2(node ** headRef, int value){
  node * newNode;
  newNode = (node*)malloc(sizeof(node));
  newNode->data = value;
  newNode->next = *headRef;
  *headRef = newNode;
}
```



Popular Interview Question

- Write a function that partitions a linked list around a value x, such that all nodes less than or equal to x come before all nodes greater than x.

```
Node * partition1(Node * head, int num)
```

```
/* Move values smaller than num to beginning of the list */
Node * partition1(Node * head, int num)
{
  Node * current = head;
  Node * after = NULL;
  while (current){
    if (current->next)
      after = current->next;
    else
      after = NULL;

    if(after && after->data <= num){
      current->next = after->next;
      after->next = head;
      head = after;
    }
    else
    current = current->next;
  }
  return head;
}
```

Learning Goals revisited

- Define and use linked lists in an implementation with dynamic memory allocation.
- Traverse a node-based linked list using a loop
- Mutate a node-based linked list
- Determine the time complexities of operations on arrays and linked lists.
- Compare and contrast the implementation of a list using arrays, singly-linked lists, and doubly-linked lists in C.