Today’s class

- Assignment-1
- Linked lists

DSA assignment-1

DSA assignment-1

- Store records corresponding to users in a dynamic array
- Implement dynamic arrays with O(1) amortized cost for an insert and a delete operation
- Implement merge sort, quick sort, and selection sort to enable fast searching

DSA assignment-1

- struct record represents a record corresponding to a user
  - It has several fields name, uid, age, etc.
  - For this assignment, name, uid, and status fields are relevant
    - uid is a 16-byte character array (not '\0' terminated) that is unique to every user
    - name is '\0' terminated string (maximum 16 bytes) that may not be unique
  - You are not allowed to change the structure of struct record

- Store the address of the dynamic array in record_arr
  - Initially, record_arr is set to NULL
    - You can think of NULL as an invalid address
    - Dereferencing a NULL address may cause an access violation

DSA assignment-1

- You are to implement a library that implements a dynamic array to insert and delete records
- Implement quick sort, merge sort, and selection sort to sort the dynamic array, using uid as the key
  - Code to compare two uids are already provided
- Implement linear and binary search algorithms to search the record corresponding to a given uid

DSA assignment-1

- Implement quick sort to sort the dynamic array, using name as the key
- Implement an API to find the number of records corresponding to a given name using binary and linear search algorithms

DSA assignment-1

- We have provided three test cases to test your library
- Submit the output of the "make submit1", "make submit2", and "make submit3" in the final report
    - These commands will compile and run the test cases
- You can compile the test cases using the "make" command in the PA1 folder
- Use allocate_memory/free_memory instead of malloc/free

DSA assignment-1

- Submit the “p1.c” file and your report in the pdf format
  - Make sure that you follow all the submission instructions properly; otherwise, you will receive ZERO in the assignment
  - Late submissions are not allowed
  - Our test scripts may not work if you leave print statements, use malloc or free directly in your code
    - Make sure to remove all print statements and only use the allocator APIs provided to you

How to debug?

- Use printf to debug your program
- Try testing your program for smaller input size first before moving on to the large input size
- Make sure to remove all the print statements before submitting

Doubts

- Post any assignment-related queries on the Google Classroom
- You can also meet me during my office hours or after the class
- Make sure that you can compile and run the skeleton code on your machine at the earliest
  - Any installation-related queries after the first three days from releasing the assignment will not be answered

Other details

- We inserted NOT_IMPLEMENTED macros at different places in “pa1.c”, where implementation is not available. Remove these calls before implementing these routines.

Bonus part

- In the bonus component, you need to implement an O(n) algorithm to find the median and use that as the pivot in your quick sort implementation

Linked lists

References

- Read section-10.2 from the CLRS
- Read chapter-3 from Mark Allen Weiss

Problems with dynamic arrays

Problems with dynamic arrays

- The entire array needs to be consecutive
  - On a 32-bit system, the size of an address is just 32-bit, and allocating a large consecutive buffer might be an issue

- Memory wastage: half of the array could be empty at a given point. This could be an issue if the array size is very large.

- A large number of copy operations when the array is full or only 25% occupied

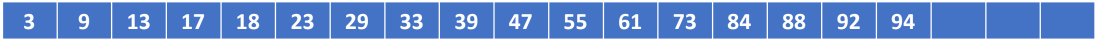
Arrays

- You can also think of an array as a sequence of n elements: $A_0, A_1, \ldots, A_{n-1}$, where,
  - the position of the 1st element, A_0, is 0
  - the position of the 2nd element, A_1, is 1
  - …
  - the position of the nth element, A_{n-1}, is n-1

Arrays

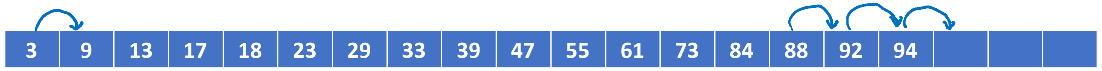
- We want to support the following operations without changing the order of the elements in the sequence
  - Insert an element at the i^{th} position
  - Delete the element at the i^{th} position
  - Find the element at the i^{th} position

Inserting an element



`Insert 1 at position 0 without changing the order.`

Inserting an element



`Insert 1 at position 0 without changing the order.`

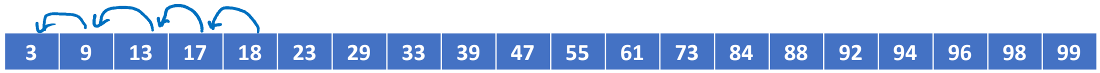
Let’s say there are n elements in the array. The first element is at position 0, the second element is at position 1, and so on. Inserting a value at position zero without changing the order of the elements, we need to shift all the elements by one, starting from the last element. First, we need to copy the element at position n-1 to n, followed by n-2 to n-1, and so on in that order. Notice that this will take O(n) operations.

Deleting an element

3	9	13	17	18	23	29	33	39	47	55	61	73	84	88	92	94	96	98	99

```
Delete 3 without changing the order.
```

Deleting an element



```
Delete 3 without changing the order.
```

To delete the element at position 0 without altering the order, we need to copy the element from position 1 to 0, position 2 to 1, and so on in that order. This also requires O(n) operations.

Problems with dynamic arrays

- Can we insert an element at position zero in O(1) operations without changing the order of other elements?

Problems with dynamic arrays

- Can we insert an element at position zero in O(1) operations without changing the order of other elements?
  - How about allocating a new array of size one and storing the element in the new array? Subsequently, the new array is used to access the first element, and the original array is used to access the other elements.
  - What is the problem with the above approach?

Problems with dynamic arrays

- Can we insert an element at position zero in O(1) operations without changing the order of other elements?
    - How about allocating a new array of size one and storing the element in the new array? Subsequently, the new array is used to access the first element, and the original array is used to access the other elements.
    - What is the problem with the above approach?
        - The array elements must be stored at consecutive addresses. Two different allocations are not guaranteed to be consecutive.

Linked list

- To facilitate fast insertion while preserving the order, we need to ensure that the array elements can be stored in non-contiguous memory

- This is the basic idea behind linked-lists

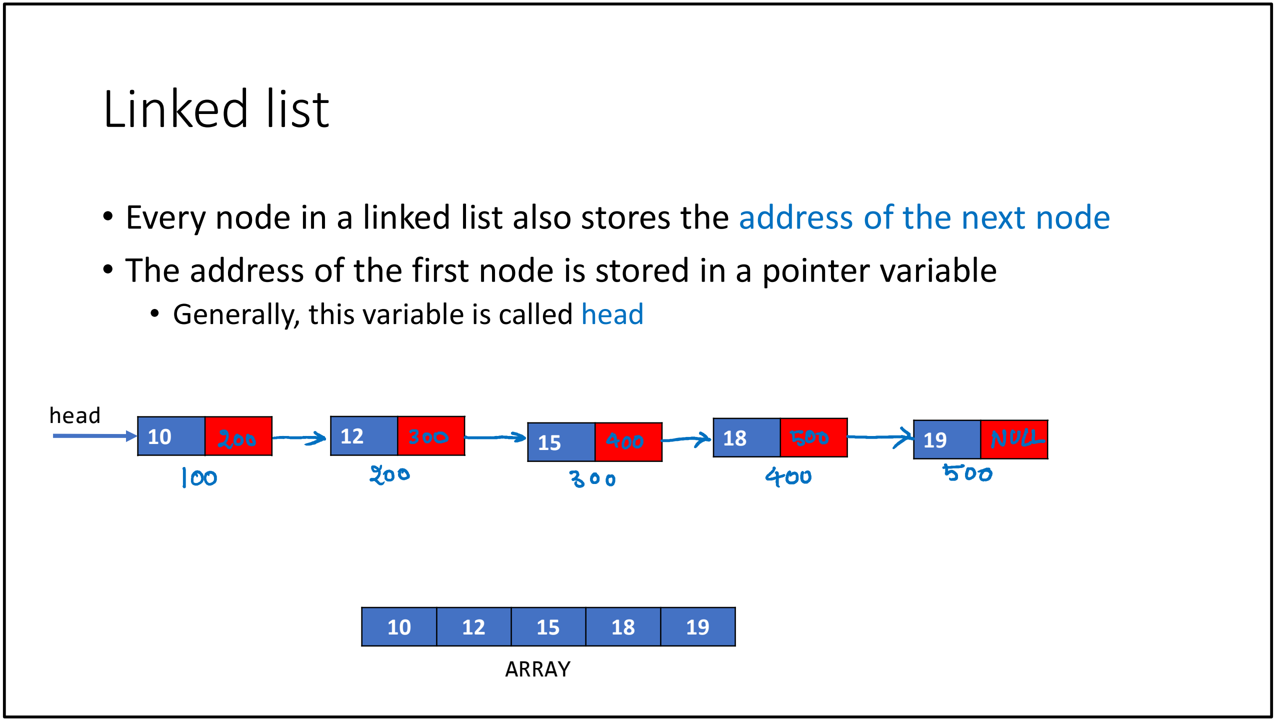
Linked list

- Linked list is almost like an array, except the addresses of two consecutive elements in the list may not be consecutive
- Thus, we don’t need to copy elements during linked-list insertion and deletion
- The elements of a linked list are also called nodes

Linked list

- Suppose the base address of an array is stored in the variable `ptr`
- How can we access the fifth element of the array?

- Suppose the base address of a linked list is stored in the variable `ptr`
- How can we access the fifth element of the linked list?



Linked list

- Every node in a linked list also stores the address of the next node
- The address of the first node is stored in a pointer variable
  - Generally, this variable is called head

In this example, the address of the first node of the linked list is 100, the second node is 200, and so on. The first node stores the address of the second node, i.e., 200. The second node stores the address of the third node, i.e., 300, and so on. The last node stores the NULL address, which is an invalid address or the address of an empty list.

Linked list

- The last node of the linked list contains NULL in the address field
    - You can think of NULL as an invalid address or the address of an empty list

Type

- What is the type of a linked-list node that stores integer values?

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

In addition to the value field, a linked list node also contains the address of the next node. If the type of a linked-list node is “struct node”, the type of the address of the linked-list node would be “struct node*”. Therefore, “struct node” contains a field of type “struct node*” to store the address of the next node.

Allocation

$\underline{(*n)}.\underline{val}$

$n \rightarrow val$

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

struct node* allocate_node(int val) {
  struct node *n = (struct node*)malloc(sizeof(struct node));
  if (n == NULL) {
    printf(“Unable to allocate more memory\n”);
    return NULL;
  }
  n->val = val;
  n->next = NULL;
  return n;
}
```

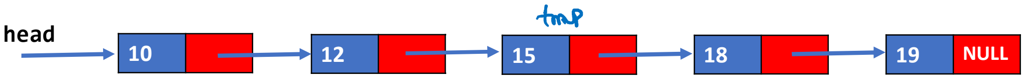
Allocating a linked list node.

allocate_node allocates a new node, stores the element's value in the val field, and stores NULL corresponding to the address of the next node.

Printing Linked-List

Printing list

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

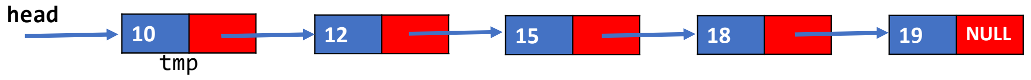


```
printf ("%d", head->val);
10

struct node * tmp = head->next;
  printf ("%d", tmp->val);
  12

tmp = tmp->next;
  printf ("%d", tmp->val); 15

tmp = tmp->next;
  printf ("%d", tmp->val); 18
```

To print the entire linked list, we can store the address of the head in a temporary variable, tmp. Until tmp is not NULL, we can print the value of tmp and update tmp with the address of the next node in a loop.

Printing list

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

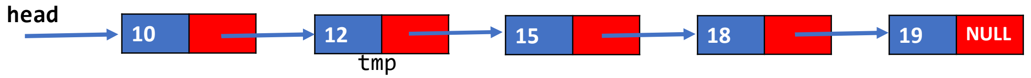


tmp = head

printf("%d ", tmp->val); 10

Printing list

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

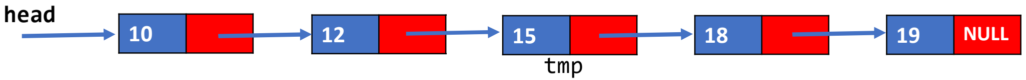


```
tmp = head
printf("%d ", tmp->val); 10
```

`tmp = tmp->next;`
`printf("%d ", tmp->val); 12`

Printing list

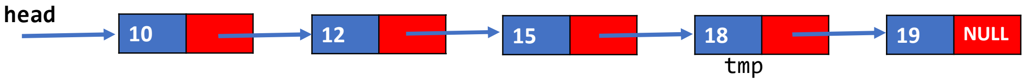
```
struct node {
  int val;
  struct node *next;
};
```



```
tmp = head
printf(“%d ”, tmp->val); 10
tmp = tmp->next;
printf(“%d ”, tmp->val); 12
```
tmp = tmp->next;
printf(“%d ”, tmp->val); 15

Printing list

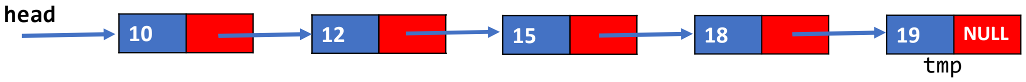
```
struct node {
  int val;
  struct node *next;
};
```



```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

Printing list

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

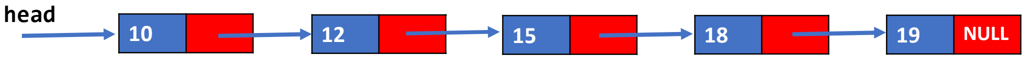


```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

tmp = tmp->next;
printf("%d ", tmp->val); 19

Printing list

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



tmp = NULL

```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

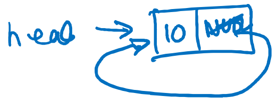
```
tmp = tmp->next;
printf("%d ", tmp->val); 19
tmp = tmp->next
```

Printing list

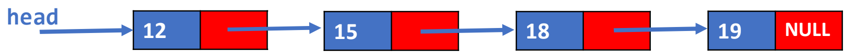
```
struct node {
  int val;
  struct node *next;
};

void print_list(struct node *head) {
  struct node *tmp = head;
  while (tmp != NULL) {
     printf(“%d ”, tmp->val);
     tmp = tmp->next;
  }
  printf(“\n”);
}
```

Insertion (front)

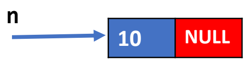
Insertion (front)



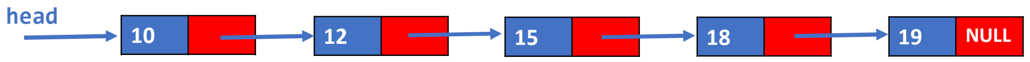
- Before insertion



```
head = n;
n -> next = head;
```

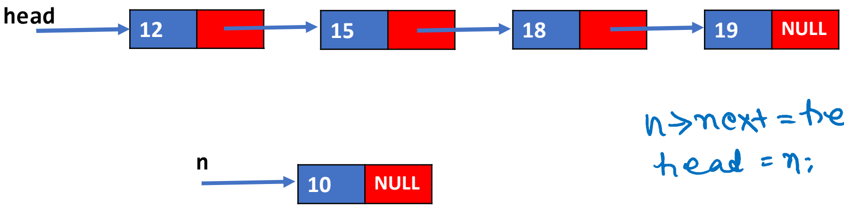


- After insertion



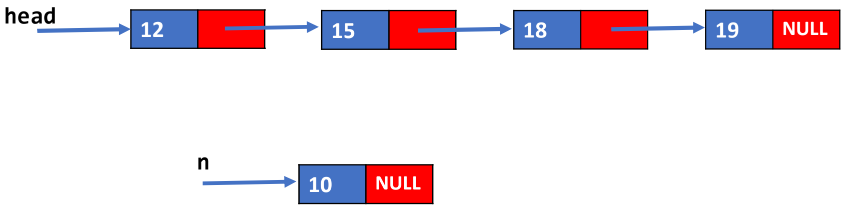
In this example, head points to the first node of the linked list. n points to a new node. After inserting the new node in the front, the head will point to the new node, and the new node will store the address of the node that was the first node before the insertion. If we first update the head to point to the new node, we will lose the reference to the original linked list. Therefore, we first need to store the address of the first node in the new node and then make the new node the first node of the linked list.

Insertion (front)

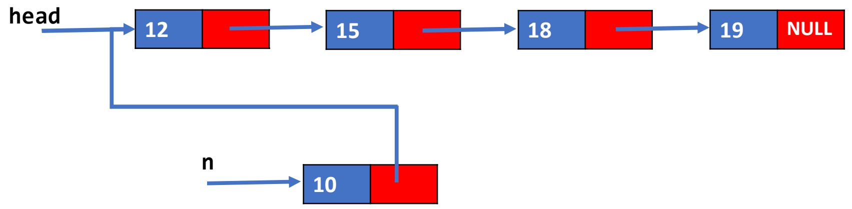


```
n->next = head;
head = n;
```

Insertion (front)

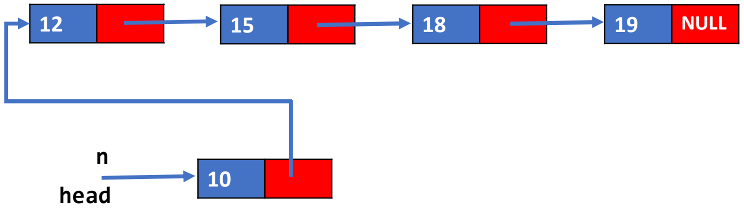


Insertion (front)



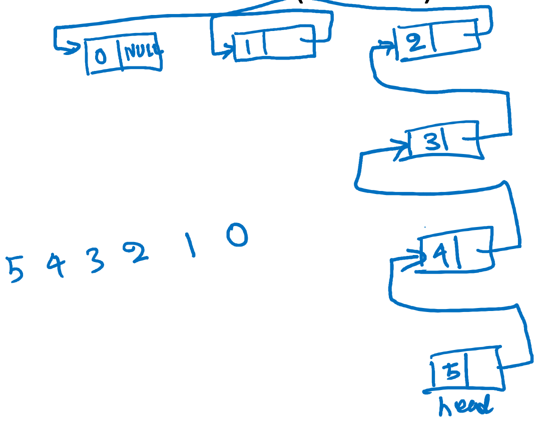
```
n->next = head;
```

Insertion (front)



```
n->next = head;
head = n
```

Insertion (front)



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

// returns the new head
struct node* insert_front(struct node *head,
                          struct node *n) {
  n->next = head;
  return n;
}

int main() {
  struct node *head = NULL;
  struct node *n;
  int i;

  for (i = 0; i <= 5; i++) {
    n = allocate_node(i);
    head = insert_front(head, n);
  }
  print_list(head);
  return 0;
}
```

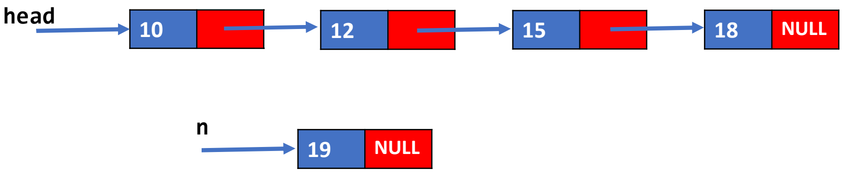
Time complexity: $O(1)$

In this program, insert_front takes a pointer to the starting node (or head) of the linked list and a pointer to the node that needs to be inserted. This function returns the new head after inserting the new node at the front of the linked list. insert_front requires O(1) operations.

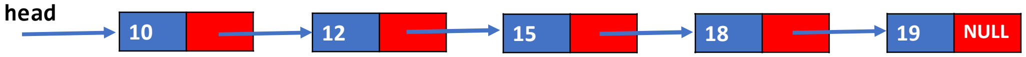
Insertion (rear)

Insertion (rear)

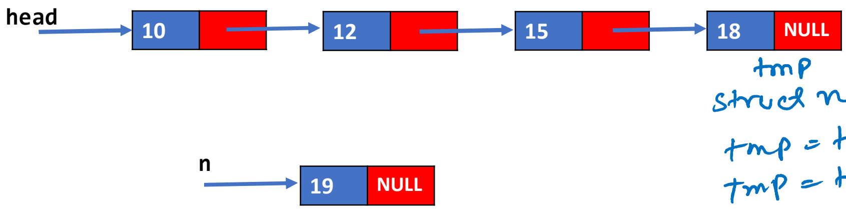
- Before insertion



- After insertion



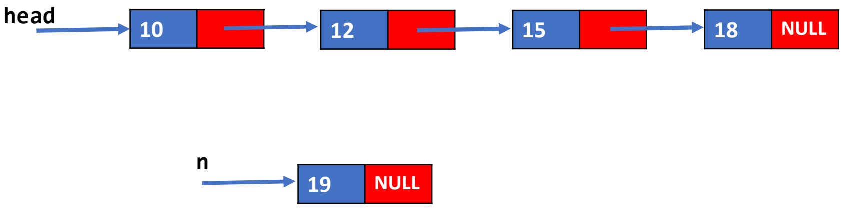
Insertion (rear)



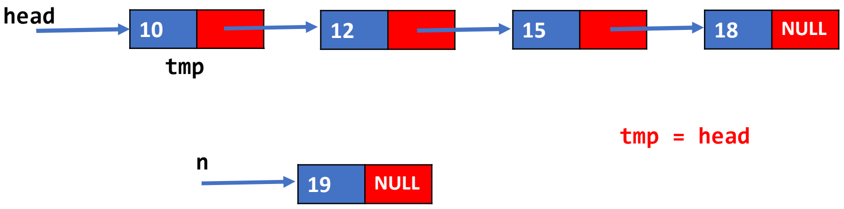
```
struct node* tmp=head;
tmp = tmp->next;
tmp = tmp->next;
tmp = tmp->next;
tmp->next = n;
```

To insert a node at the rear end of the list, we first need to find the address of the last node. To compute the address of the last node, we can initialize a temporary variable tmp with the value of head and keep updating it with the address of the next node in a loop until we reach a node that stores NULL in its next field. At this point, we can store the address of the new node in the next field of tmp. Consequently, the new node will be inserted at the rear end of the linked list.

Insertion (rear)

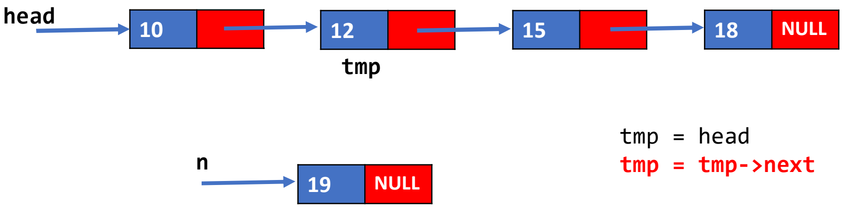


Insertion (rear)



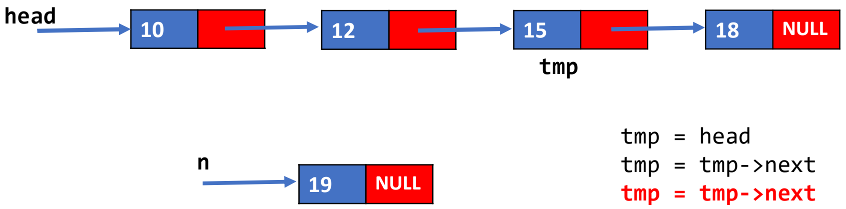
```
tmp = head
```

Insertion (rear)



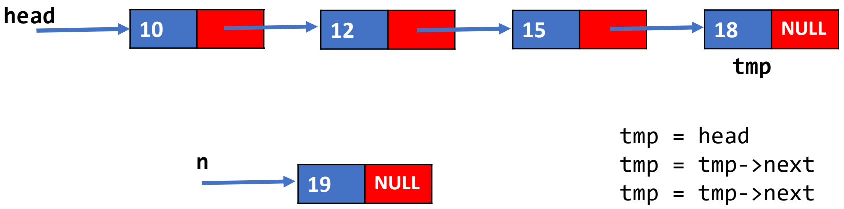
```
tmp = head
tmp = tmp->next
```

Insertion (rear)



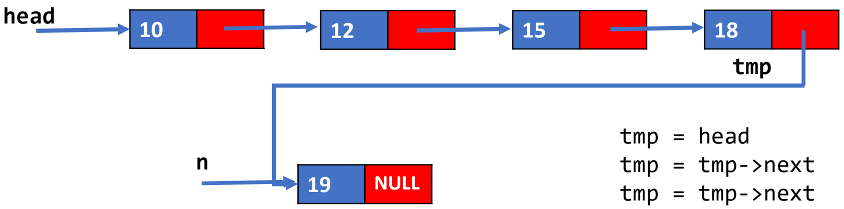
```
tmp = head
tmp = tmp->next
tmp = tmp->next
```

Insertion (rear)



```
tmp = head
tmp = tmp->next
tmp = tmp->next
tmp = tmp->next
```

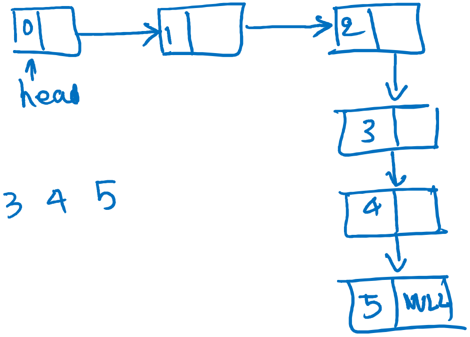
Insertion (rear)



```
tmp = head
tmp = tmp->next
tmp = tmp->next
tmp = tmp->next
tmp->next = n
```

```
struct node* insert_rear(struct node *head, struct node *n) {
  if (head == NULL) {
     return n;
  }
  struct node *tmp = head;
  while (tmp->next != NULL) {
     tmp = tmp->next;
  }
  tmp->next = n;
  return head;
}

int main() {
  struct node *head = NULL;
  struct node *n;
  int i;

  for (i = 0; i <= 5; i++) {
    n = allocate_node(i);
    head = insert_rear(head, n);
  }
  print_list(head);
  return 0;
}
```

Insertion at rear



0 1 2 3 4 5

Time complexity: $O(n)$

Notice that the insertion at the rear takes O(n) operations. insert_rear returns the new head of the linked list. The head can change if the original list is an empty list.

Insertion (rear)

- How can we implement insertion rear in O(1) operations?

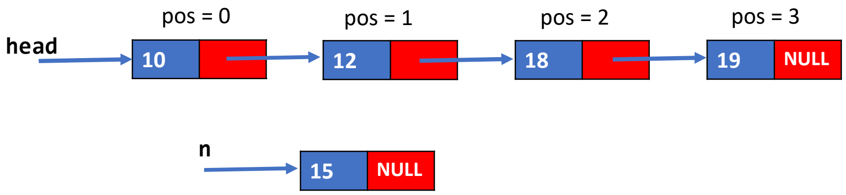
Insertion (rear)

- How can we implement insertion rear in O(1) operations?
  - Keep track of the tail (last node) of the linked list in addition to head
  - No need to walk the entire list to find the tail
  - After insertion, update the tail to point to the inserted node

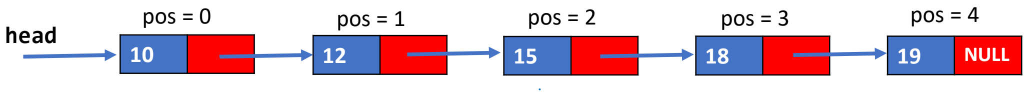
Insertion (at specific position)

Insertion (at specific position)

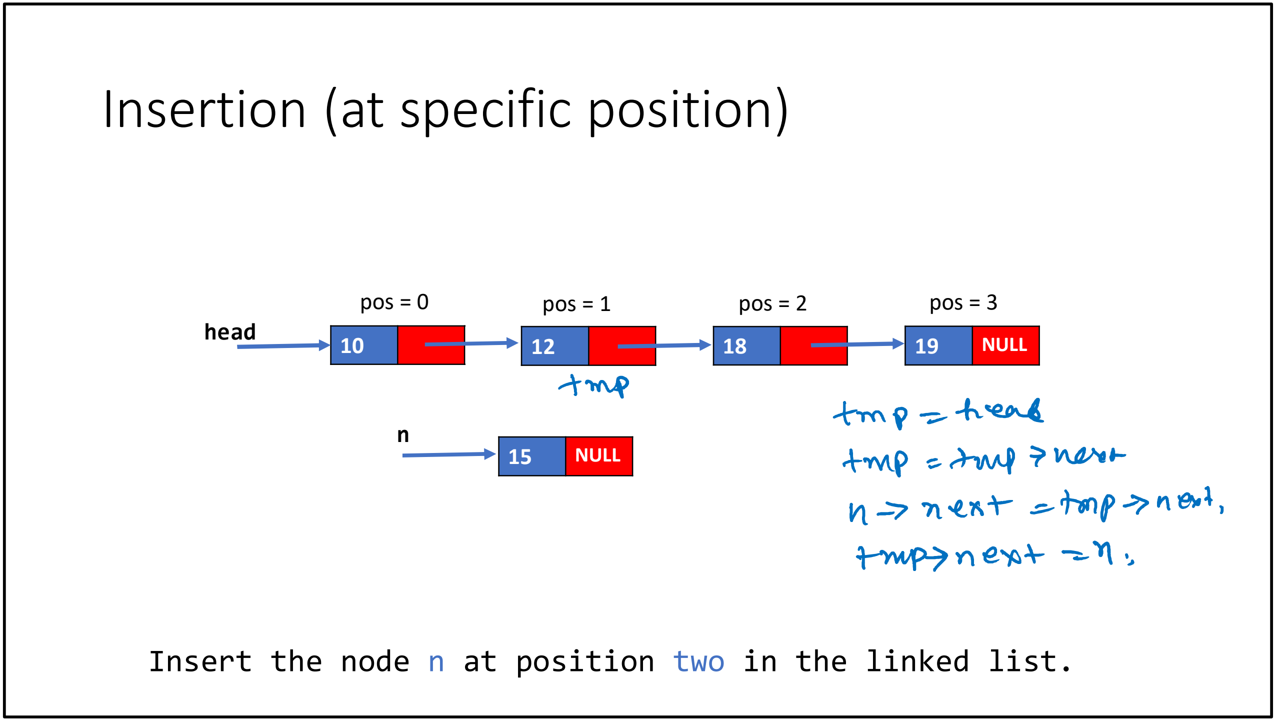
- Before insertion



- After inserting the new node n at position two



The position of the first node is 0, the second node is 1, and so on. After inserting a new node at a specific position i, the node at position i-1 will point to the new node, and the new node will point to the node that was previously at position i. In other words, the positions of all the nodes starting from i will be increased by one in the final list after the insertion.



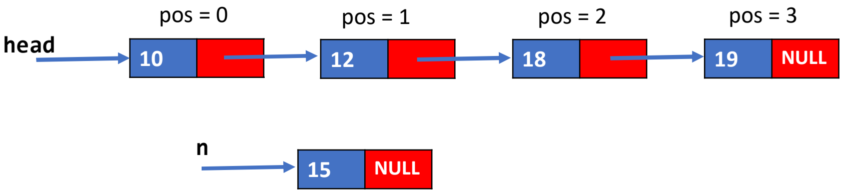
Insertion (at specific position)

```
tmp = head
tmp = tmp -> next
n -> next = tmp -> next;
tmp -> next = n;
```

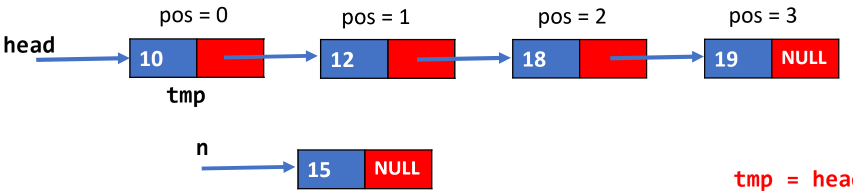
Insert the node n at position two in the linked list.

To insert a node at position two, we first need to initialize a temporary variable tmp with the head of the linked list. Now we node to update tmp with the address of the next node in a loop until we reach the node at position i-1. At this point, we need to first store the address of the ith node in the address field of n, followed by storing the address of n in the address field of tmp. This will give us the desired list.

Insertion (at specific position)

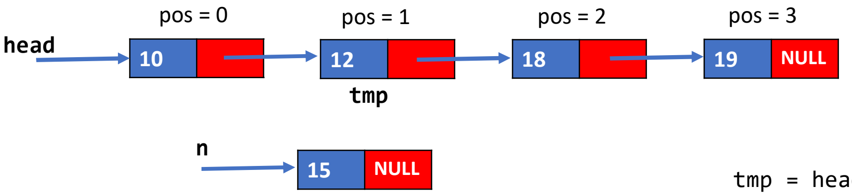


Insertion (at specific position)



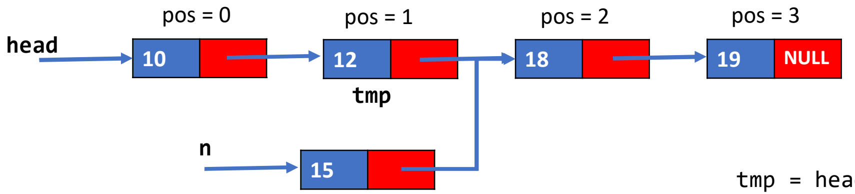
```
tmp = head
pos = 0
```

Insertion (at specific position)



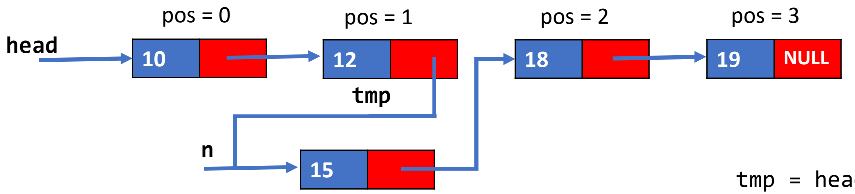
```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
```

Insertion (at specific position)



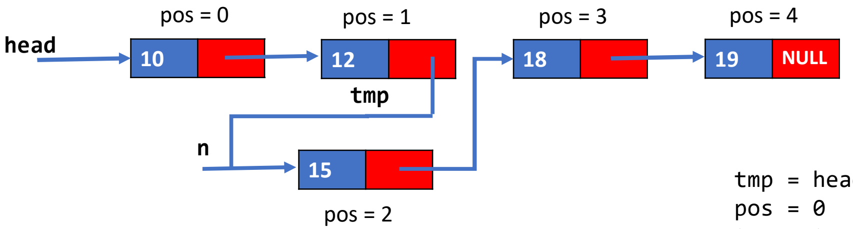
```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
```

Insertion (at specific position)



```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
```
`tmp->next = n`

Insertion (at specific position)



```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
tmp->next = n
```

```c
// insert the node n at position pos, returns the new head
// the insertion is successful if 0 <= pos <= N,
// where N  is the total number of elements in the list
struct node* insert_pos(struct node *head, struct node *n, int pos) {
  if (pos == 0) {
    n->next = head;
    return n;
  }

  struct node *tmp = head;
  int i = 0;

  while (i != pos - 1 && tmp != NULL) {
    tmp = tmp->next;
    i = i + 1;
  }

  if (tmp != NULL) {
    n->next = tmp->next;
    tmp->next = n;
  }
  return head;
}
```

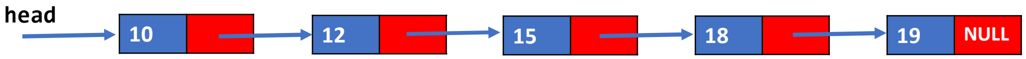
Insertion at specific position

Time complexity: $O(n)$

If we insert at pos=0, it is the same as inserting at the front. If we insert at pos=n, it is the same as inserting at the rear. We need to be careful when pos>n because, in that case, we can’t insert the element. Notice that an insert operation may take O(n) operations in the worst case. insert_pos returns the new head of the linked list.

Search

Search



```
Search a linked list node that contains 18 in the value field.
If the node exists, the search routine returns the node; otherwise, it
returns NULL. The search operation doesn’t modify the linked list.
```

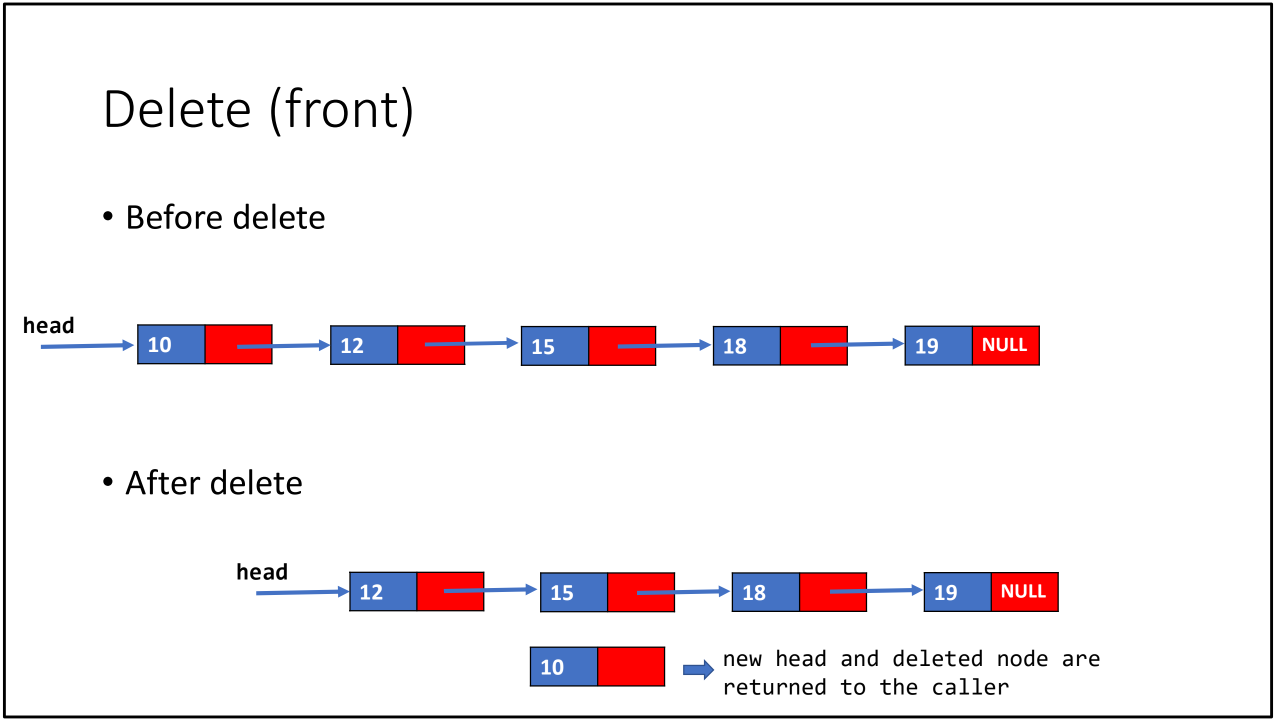
Time complexity: $O(n)$

We can iterate all the elements in the linked list as we have done so far, and if any node contains the value we are searching for, we can return the address of that node.

Search

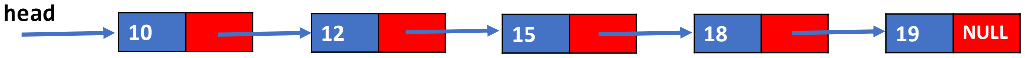
```
// Returns a linked list node that contains
// the input argument val

struct node* search(struct node *head, int val) {
  struct node *tmp = head;
  while (tmp != NULL) {
    if (tmp->val == val) {
      return tmp;
    }
    tmp = tmp->next;
  }
  return NULL;
}
```

Deleting the front node

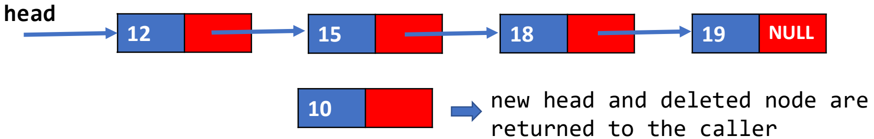


Delete (front)

- Before delete



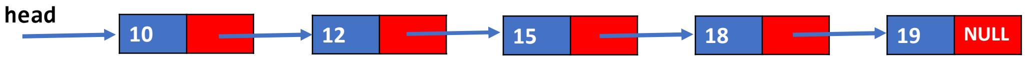
- After delete



new head and deleted node are returned to the caller

After deleting the first node, the head of the linked list will be changed. The delete procedure returns two values: the new head of the linked list and the address of the deleted node.

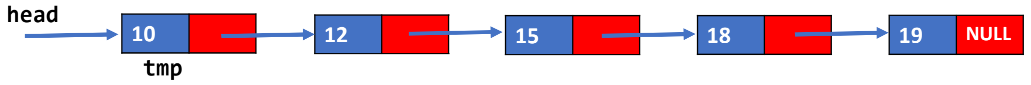
Delete (front)



tmp = head
head = head->next;

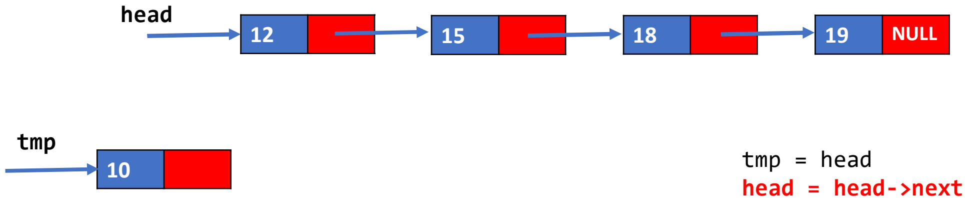
If the list is not empty, the deleted node is head, and the new head is head->next.

Delete (front)

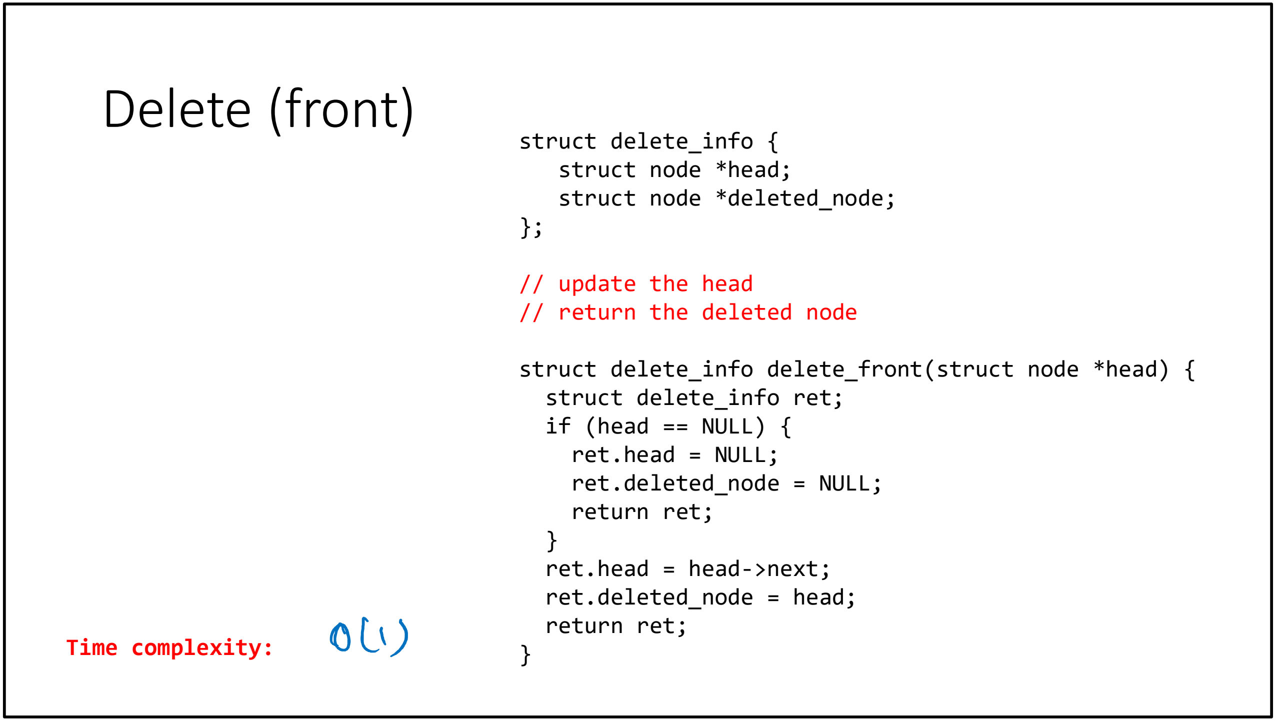


```
tmp = head
```

Delete (front)



```
tmp = head
head = head->next
```



Delete (front)

```
struct delete_info {
   struct node *head;
   struct node *deleted_node;
};

// update the head
// return the deleted node

struct delete_info delete_front(struct node *head) {
  struct delete_info ret;
  if (head == NULL) {
    ret.head = NULL;
    ret.deleted_node = NULL;
    return ret;
  }
  ret.head = head->next;
  ret.deleted_node = head;
  return ret;
}
```

Time complexity: $O(1)$

We can use a structure as we did earlier to return two values. The time complexity of delete_front is O(1).