

Today's class

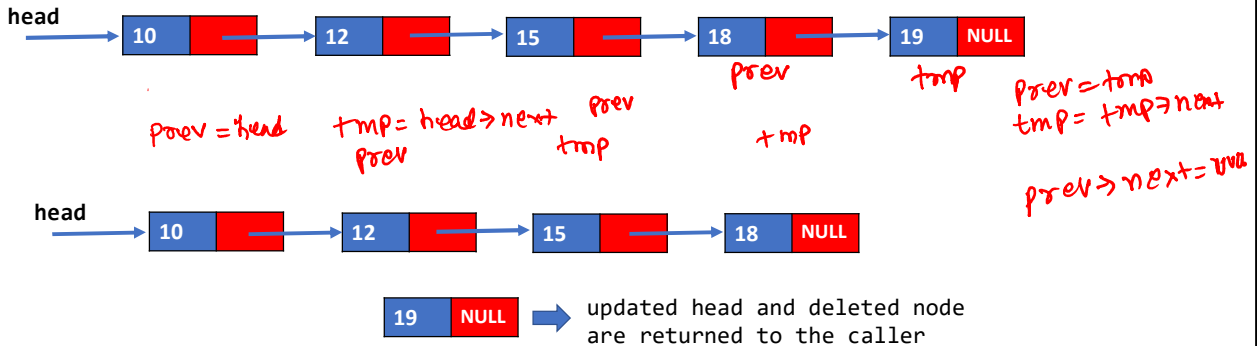
- Linked lists
- Stack
- Queue

Extra class

- There will be an extra class today at 3 pm
 - The venue is the same C101

Deleting the rear node

Delete (rear)

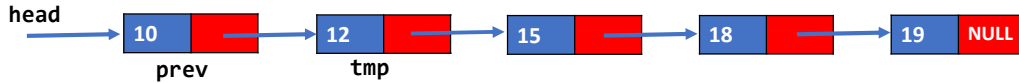


To delete the last node, we need to iterate all the nodes until we reach the last node. However, we also need a reference to the second last node to delete the last node. To achieve this, we can iterate using two temporary variables, `prev` and `tmp`, in such a way that if at a given point during the iteration `tmp` points to the node at position `i`, `prev` will point to the node at position `i-1`. When `tmp` reaches the last node (i.e., `tmp->next == NULL`), at this point, we can simply delete the last node by setting `prev->next` to `NULL`.

Delete (rear)

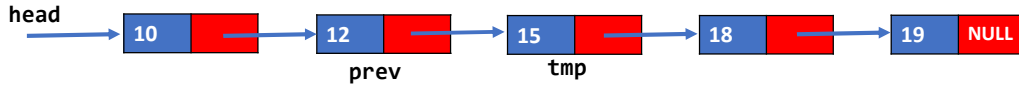


Delete (rear)



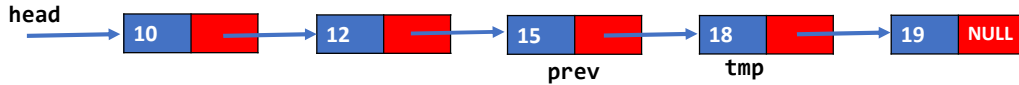
prev = head
tmp = head->next

Delete (rear)



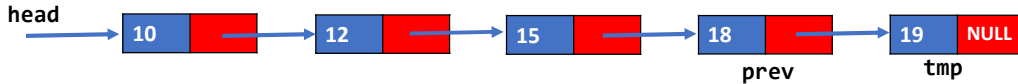
```
prev = head  
tmp = head->next  
prev = tmp  
tmp = tmp->next
```

Delete (rear)



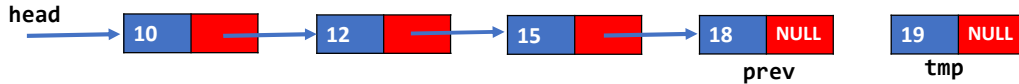
```
prev = head  
tmp = head->next  
prev = tmp  
tmp = tmp->next  
prev = tmp  
tmp = tmp->next
```


Delete (rear)



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

Delete (rear)



```
prev = head  
tmp = head->next  
prev = tmp  
tmp = tmp->next  
prev = tmp  
tmp = tmp->next  
prev = tmp  
tmp = tmp->next  
prev->next = NULL
```

```

1. struct delete_info delete_rear(struct node *head) {
2.     struct delete_info ret;
3.     if (head == NULL) {
4.         ret.head = NULL;
5.         ret.deleted_node = NULL;
6.         return ret;
7.     }
8.     else if (head->next == NULL) {
9.         ret.head = NULL;
10.        ret.deleted_node = head;
11.        return ret;
12.    }
13.    struct node *prev = head;
14.    struct node *tmp = head->next;
15.    while (tmp->next != NULL) {
16.        prev = tmp;
17.        tmp = tmp->next;
18.    }
19.    prev->next = NULL;
20.    ret.head = head;
21.    ret.deleted_node = tmp;
22.    return ret;
23.}

```

Deletion at rear

Time complexity: $O(n)$

The condition at line-3 handles the case when the input list is empty. The condition at line-8 is for the case when there is only one element in the list; in this case, the head will also change. The rest of the code iterates the list using variables tmp and prev, as discussed before.

Deleting a given node

Search

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

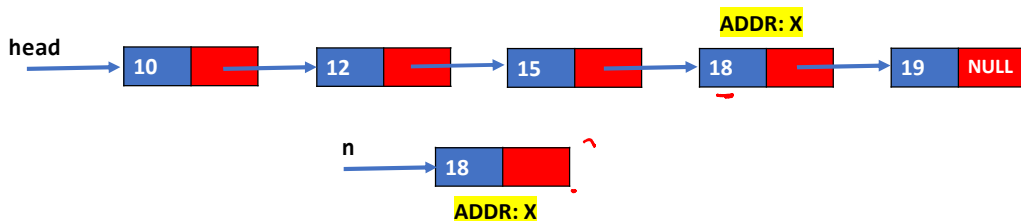
struct node* search(struct node *head, int val) {
    struct node *tmp = head;
    while (tmp != NULL) {
        if (tmp->val == val) {
            return tmp;
        }
        tmp = tmp->next;
    }
    return NULL;
}

// In many cases, we want to delete the node return by the search procedure
// Next, we will implement the delete procedure that takes a linked list node
// and deletes it from the linked list
```

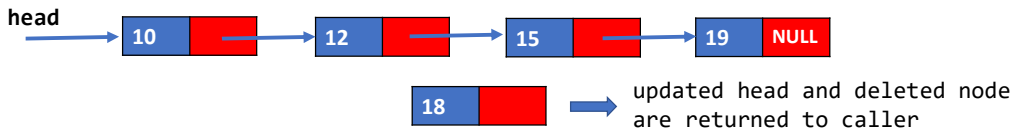
Notice that the search routine returns the address of a node that contains the value being searched. It is possible that the client of the linked-list library may want to delete that node at some point. To facilitate this, we will discuss an API that takes the address of the node that we want to delete and removes the node from the linked list.

Delete a given node

- Before deletion

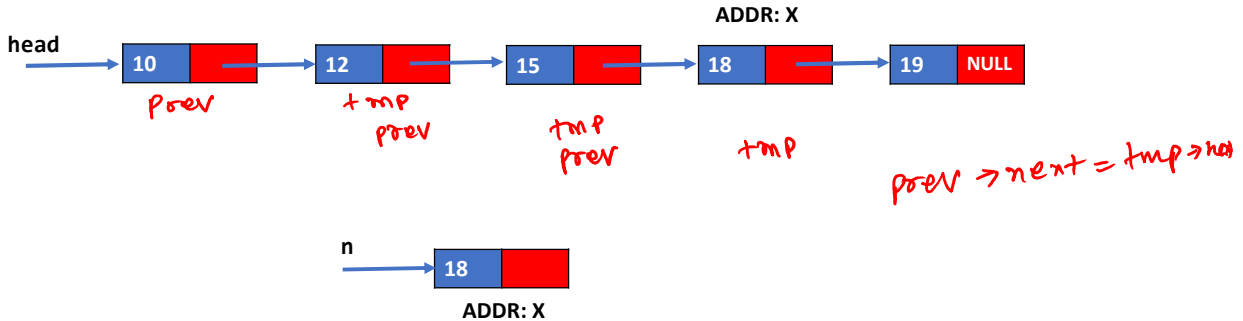


- After deletion



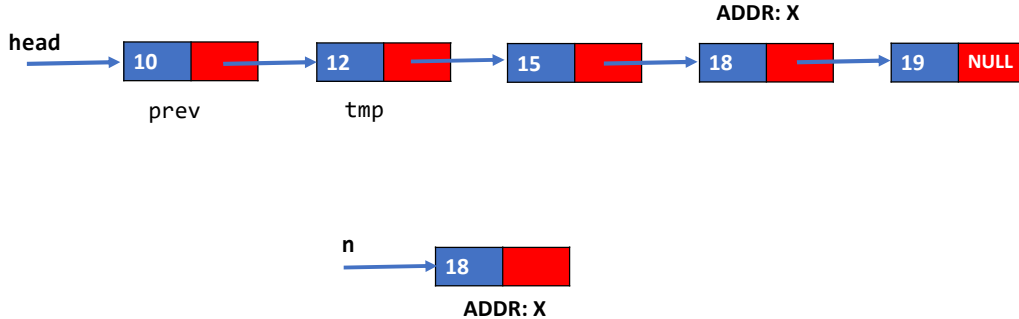
After the removal of a node, the removed node should not be reachable via the head of the linked list.

Delete a given node

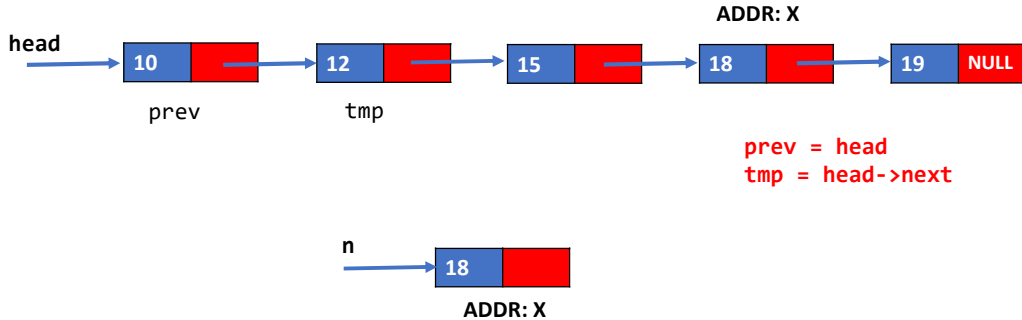


We can use the same trick that we used to delete the last node. We can iterate the list using two variables `prev` and `tmp`, and we can stop when `tmp == n`, where `n` contains the address of the node we want to delete. To delete the node, we can set the next field in the `prev` to `tmp->next`, resulting in the removal of `n` from the list. Notice that the next field in `n` still contains the address of the next node (the node that contains 19 in this example), but this is not an issue because there is no way we can reach `n` using `head` after the “`prev->next = tmp->next`” operation.

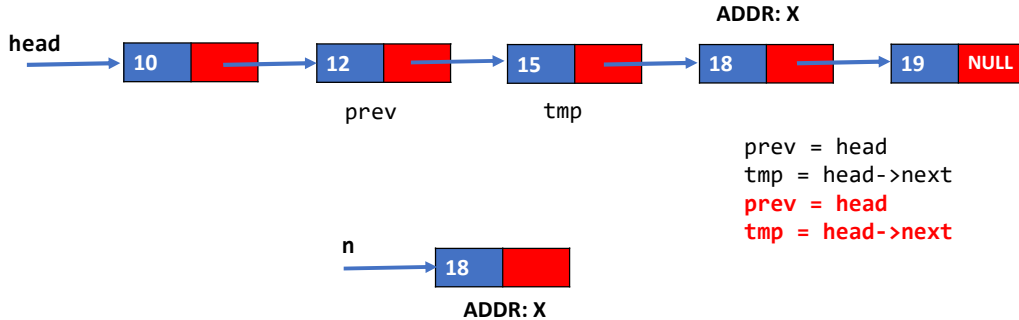
Delete a given node



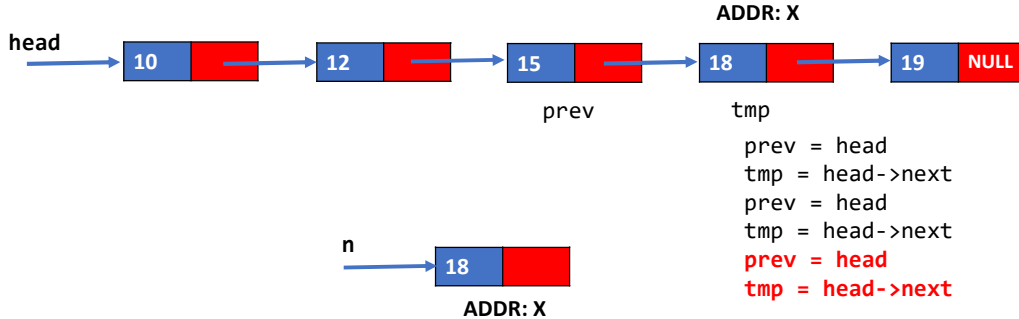
Delete a given node



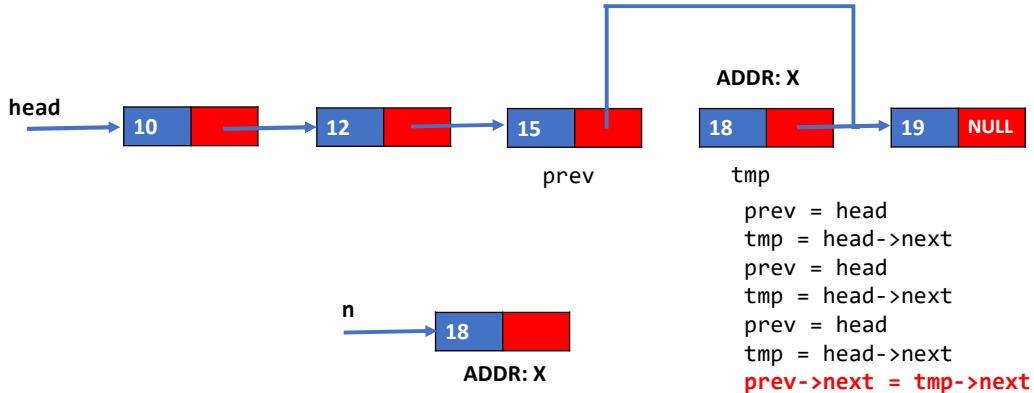
Delete a given node



Delete a given node



Delete a given node



```

1. struct delete_info delete_node(struct node *head, struct node *n) {
2.     struct delete_info ret;
3.     if (head == NULL) {
4.         ret.head = NULL;
5.         ret.deleted_node = NULL;
6.         return ret;
7.     }
8.     else if (head == n) {
9.         ret.head = head->next;
10.        ret.deleted_node = head;
11.        return ret;
12.    }
13.    struct node *prev = head;
14.    struct node *tmp = head->next;

15.    while (tmp != NULL && tmp != n) {
16.        prev = tmp;
17.        tmp = tmp->next;
18.    }
19.    if (tmp != NULL) {
20.        prev->next = tmp->next;
21.    }
22.    ret.head = head;
23.    ret.deleted_node = tmp;
24.    return ret;
25.}

```

```

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

```

Deleting a given node

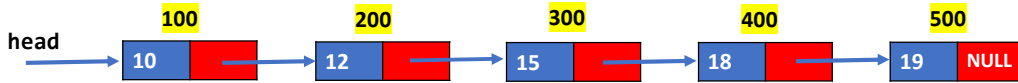
Time complexity:
 $O(N)$

The condition at line-3 handles the case when the list is empty. The condition at line-8 handles the case when we want to delete the first node; in this case, the head will change. Otherwise, we are iterating the list using tmp and prev to find n. Notice that, unlike the case of deleting the last node, in this case, it is possible that the node we are searching is not present in the list. Therefore, the condition at line-19 updates the prev node only if node n is present in the list.

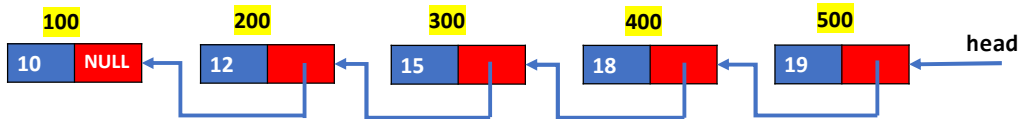
Reversing a linked list

Reversing a linked list

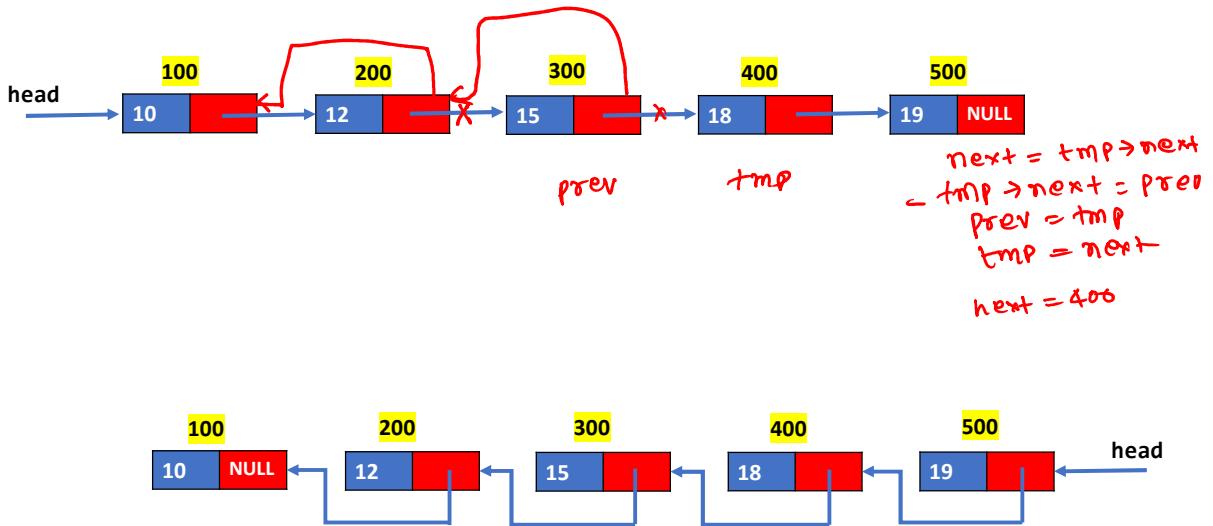
- Before reversing



- After reversing

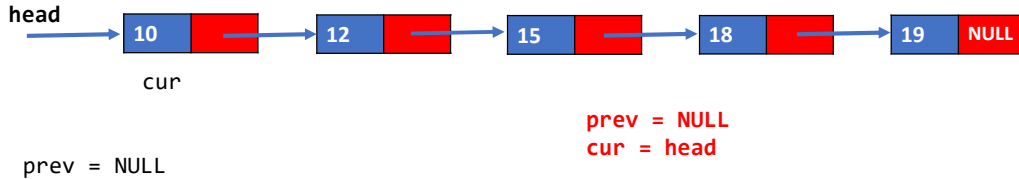


Reversing a linked list

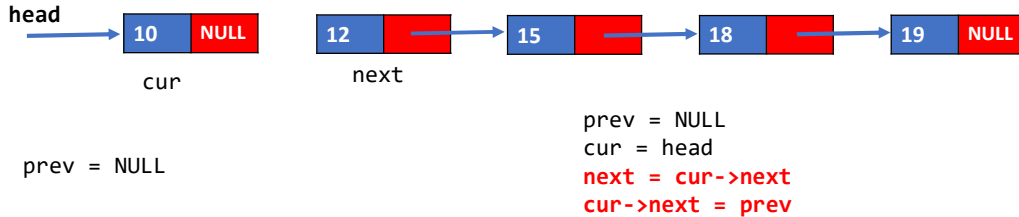


If in the original linked list, node-1 points to node-2, node-2 points to node-3, and so on, in the reversed list, node-1 will point to NULL, node-2 will point to node-1, node-3 will point to node-2, and so on. Here, node-1, node-2, etc., are the position of the node in the original list. We can iterate the list using temporary variables and update the next fields in the nodes to get the reversed list after iterating all the nodes. Let's consider the case when there is more than one node. `prev` points to node-1, and `tmp` points to node-2. Now, if we want node-2 to point to node-1 instead of node-3, we can simply set the next field of node-2 (`tmp->next`) to the value in `prev`. But this will create an issue because if we change the next field of node-2 without saving its previous value, we will lose the reference to the rest of the list (i.e., node-3 onwards). Therefore, we need to first save the value of node-3 (`tmp->next`) in another variable, say `next`, before updating `tmp->next` to `prev`. After that, we can make `prev` point to the second node (using `prev = tmp`), and `tmp` point to the third node (using `tmp = next`). We can do this in a loop until we have reversed the entire list.

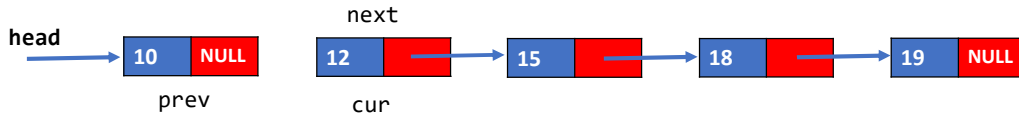
Reversing a linked list



Reversing a linked list

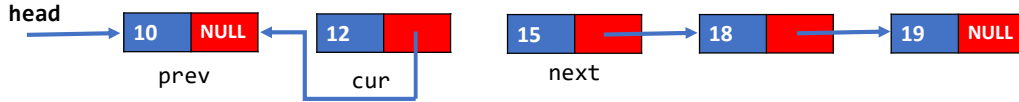


Reversing a linked list



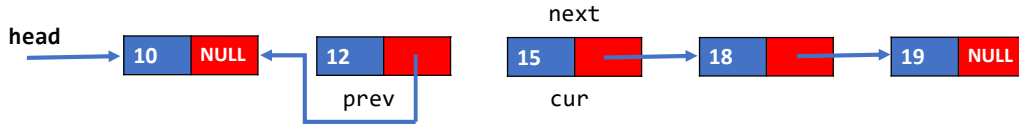
```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
```

Reversing a linked list



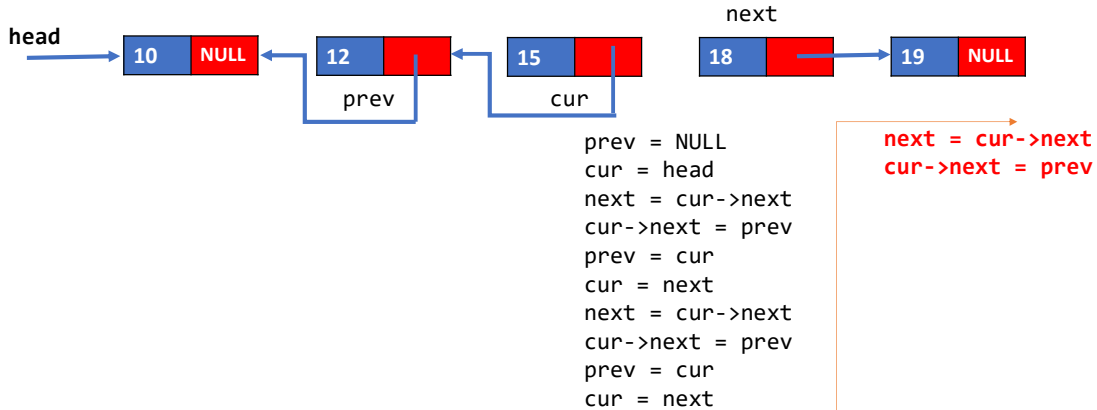
```
prev = NULL  
cur = head  
next = cur->next  
cur->next = prev  
prev = cur  
cur = next  
next = cur->next  
cur->next = prev
```

Reversing a linked list

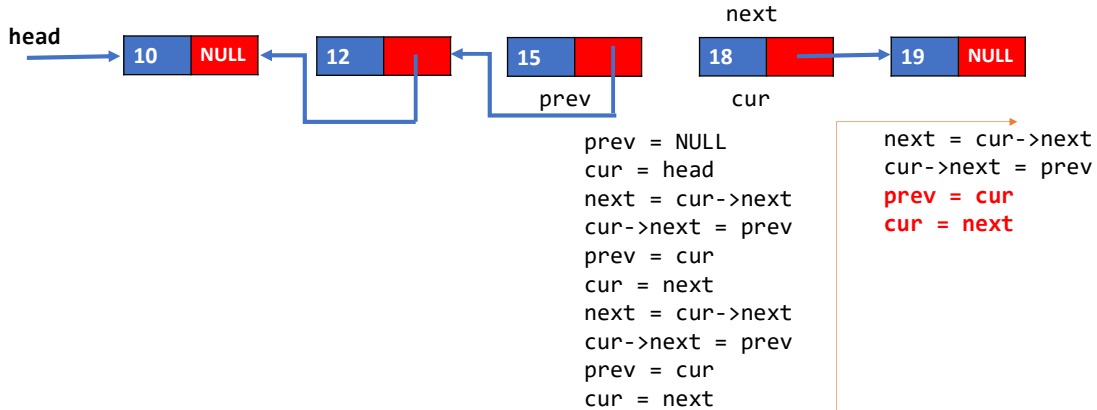


```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

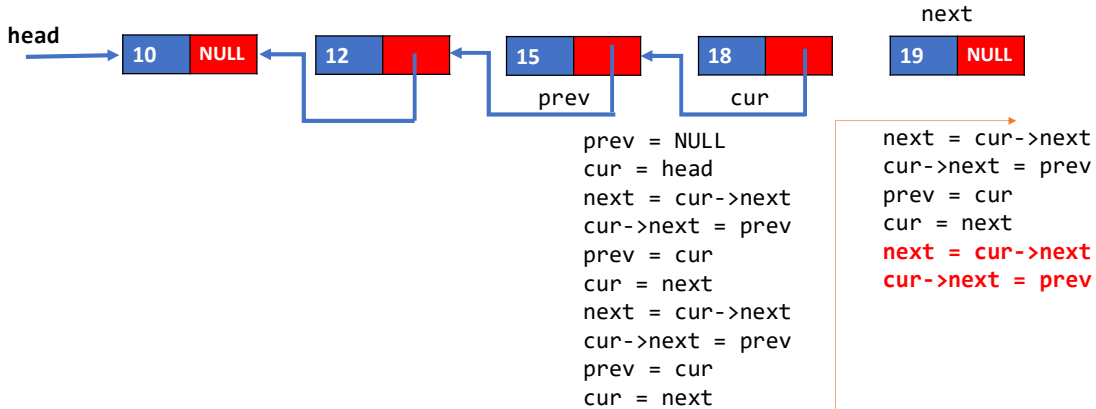
Reversing a linked list



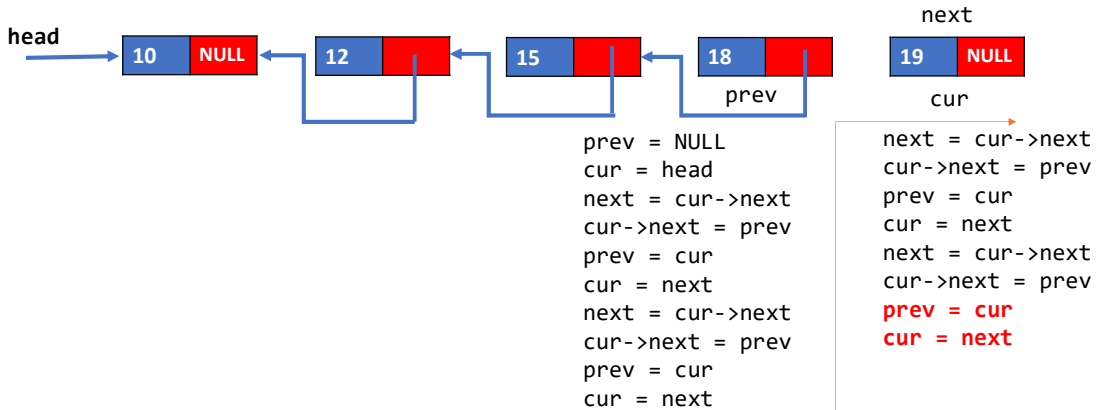
Reversing a linked list



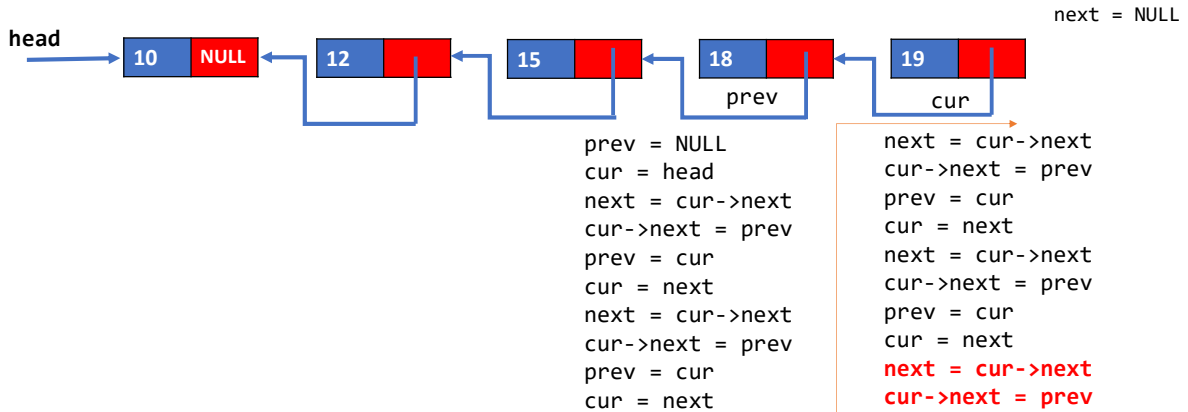
Reversing a linked list



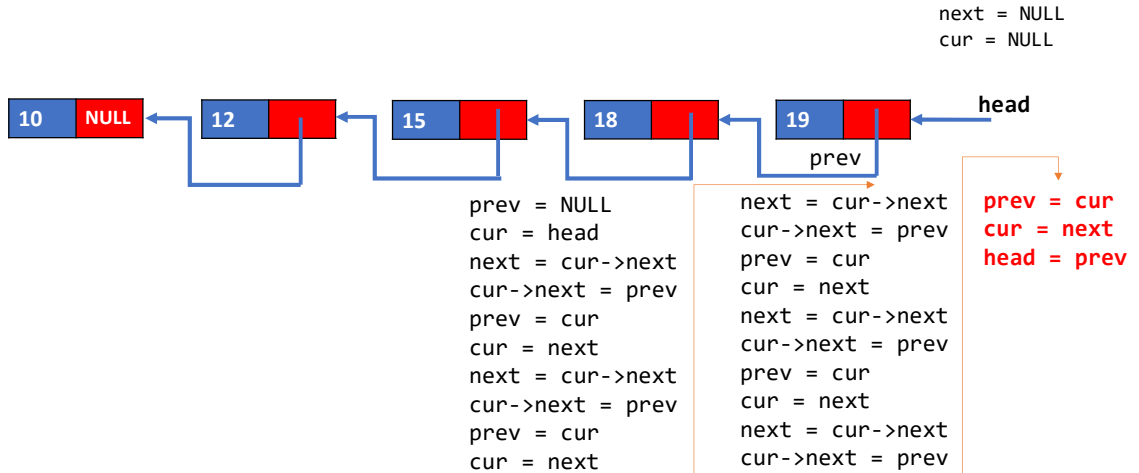
Reversing a linked list



Reversing a linked list



Reversing a linked list



Reversing a linked list

```
// returns the new head
struct node *reverse_list(struct node *head) {
    struct node *prev = NULL;
    struct node *cur = head;
    struct node *next;

    while (cur != NULL) {
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    }
    return prev;
}
```

In this code, we are using `cur` instead of `tmp`, as we used in our earlier discussion. Instead of initially setting `prev` and `cur` to the first and the second node, starting from `prev = NULL` and `cur = head` will also set the `next` field in the first node to `NULL`. After exiting from the while loop, `prev` contains the value of the last node in the original list, which is the head of the reversed list.

Applications of linked list

Polynomial operations using linked list

- Storing polynomial in a linked list

$$5x^3 + 3x^2 + 1$$

Polynomial operations using linked list

- Storing polynomial in a linked list

$$5x^3 + 3x^2 + 1$$

```
struct node {  
    int coeff;  
    int exp;  
    struct node *next;  
};
```

We can store coefficients and exponents in the linked list nodes



Type

Type

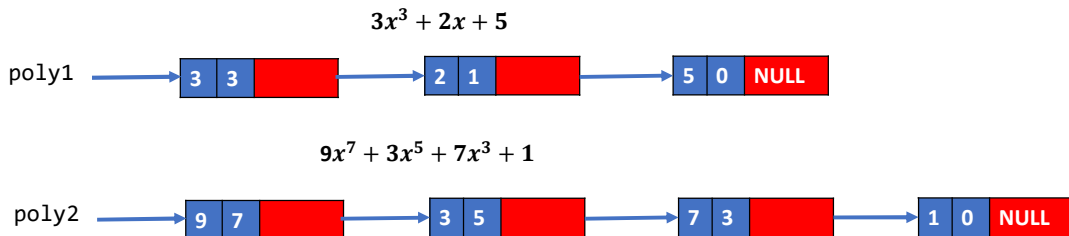
```
struct poly_node {  
    int coefficient;  
    int exponent;  
    struct poly_node *next;  
};
```

Adding two polynomials

$$\begin{array}{r} 3x^3 + 2x + 5 \\ 9x^7 + 3x^5 + 7x^3 + 1 \end{array}$$

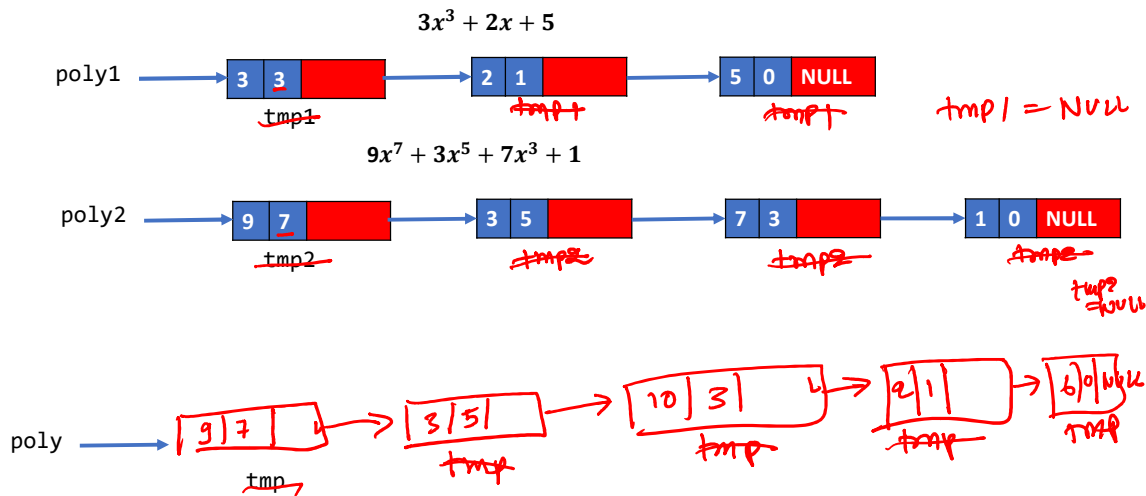
Let's say that the linked list nodes are sorted in the descending order of exponent values.

Adding two polynomials



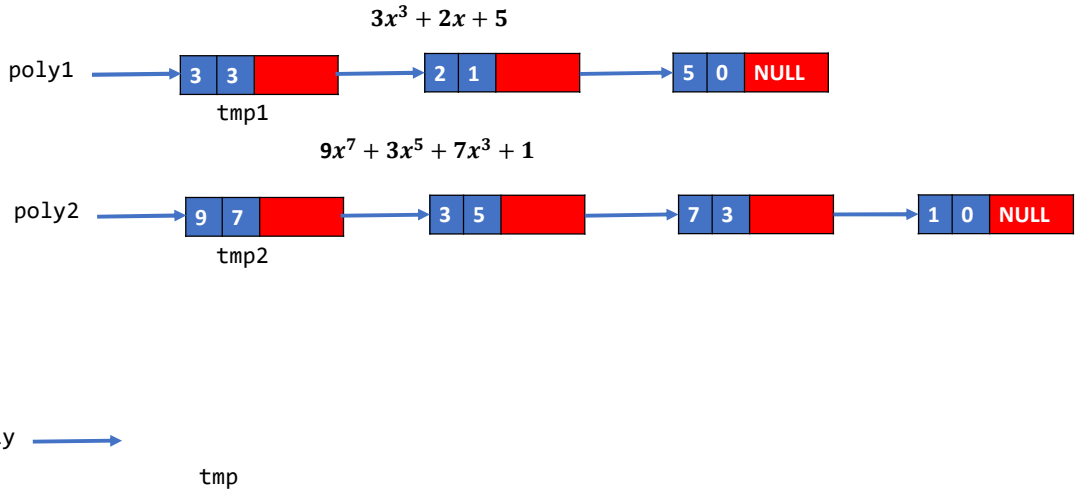
Create a new linked list that represents the sum of both the polynomials.

Adding two polynomials

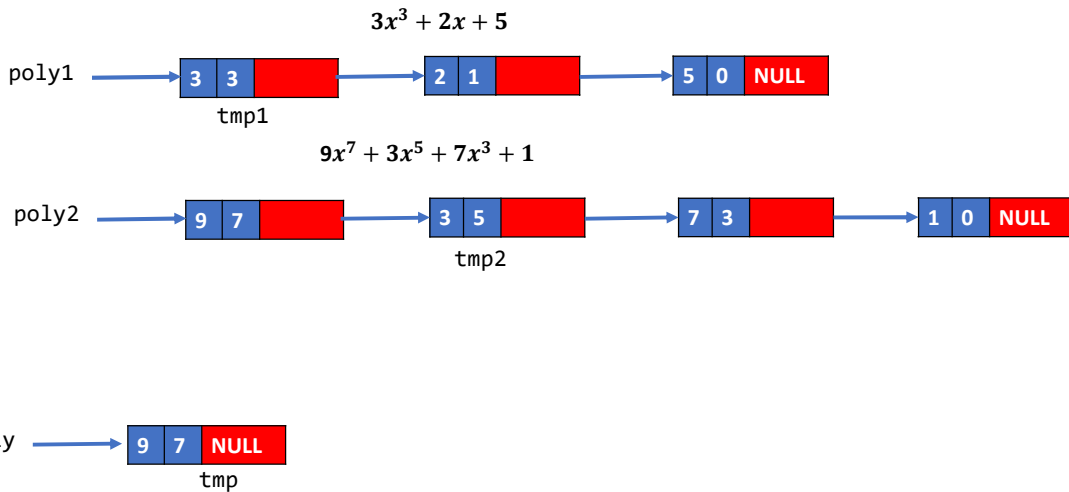


The goal here is to create a new list, L, that stores the sum of polynomials 1 and 2. The individual items of the polynomials are stored in the linked lists in the sorted order of their exponents. Initially, tmp1 and tmp2 point to the heads of polynomial1 and polynomial2. tmp points to the rear end of L (which is initially empty). If the exponent values of the nodes pointed by tmp1 or tmp2 are not the same, in this case, we allocate a new node, n, copy the value of the node with the larger exponent to n, insert n at the rear end of L, and update tmp to point the new rear in L. Afterwards, we skip the node that was copied by advancing either tmp1 or tmp2 to the next node. If the exponents at tmp1 and tmp2 are equal, in this case, we allocate a new node n, add the coefficients of nodes pointed by tmp1 and tmp2, store the sum in the coefficient field of n, copy the exponent value of tmp1 to n, insert n at the rear end of L, make tmp point to the new rear in L, and advance both tmp1 and tmp2. If either tmp1 or tmp2 reaches NULL, copy the rest of the nodes in the other polynomial to L.

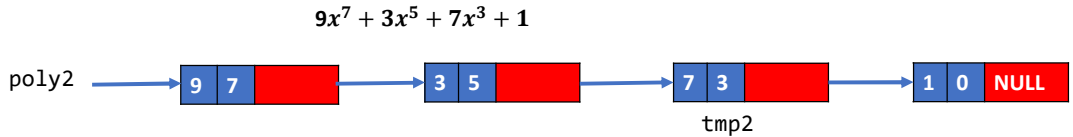
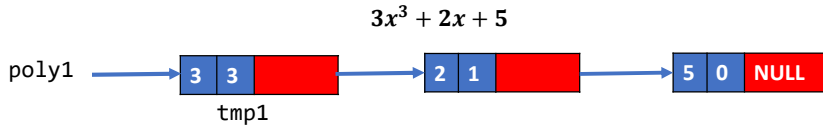
Adding two polynomials



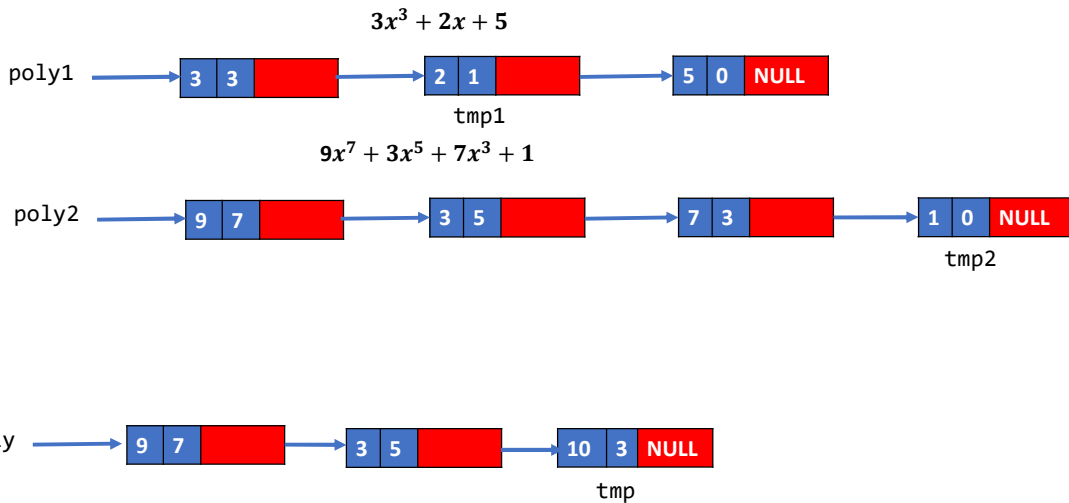
Adding two polynomials



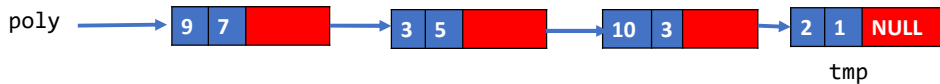
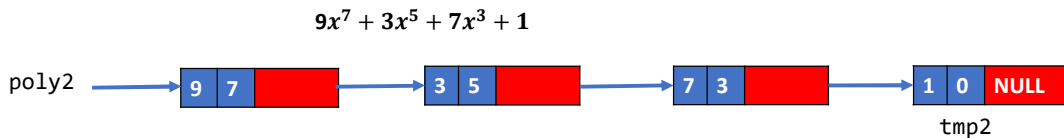
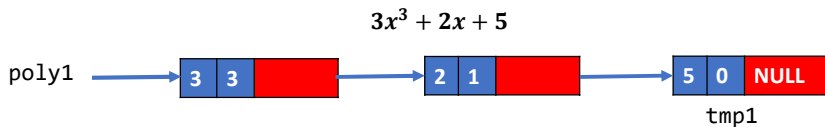
Adding two polynomials



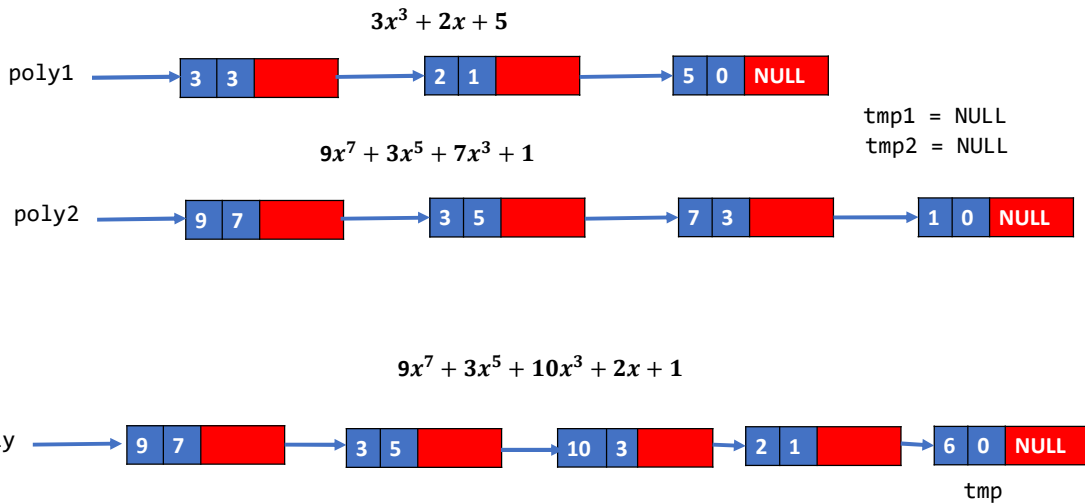
Adding two polynomials



Adding two polynomials



Adding two polynomials



Singly linked list

Singly linked list

- The linked list that we discussed so far is also called a **singly linked list**
- The address of the first node of a singly linked list is stored in a variable **head**
- Linked list can only be traversed in the forward direction starting from the **head**
- An empty linked list is represented using **NULL**

Singly linked list

- Linked list can also visualize as a recursive structure
 - Every node in a singly linked list contains a reference to the singly linked list that starts from the next node
 - The last node of the singly linked list contains a reference to the empty list

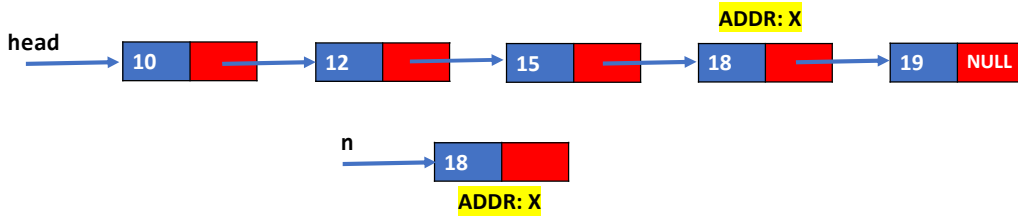
Doubly linked list

Deleting a given node

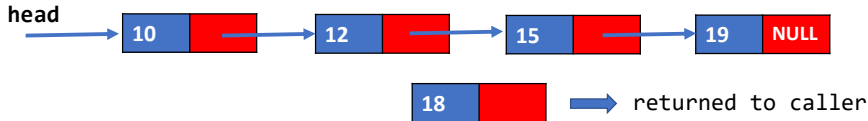
Delete a given node

How can we delete a given node n from a linked list?

- Before deletion



- After deletion

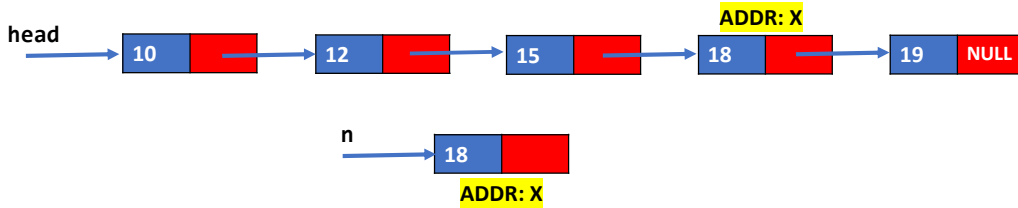


Delete a given node

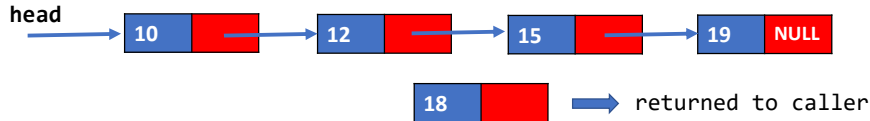
How can we delete a given node n from a linked list?

- Before deletion

Can we store additional information in linked list nodes to prevent iteration?



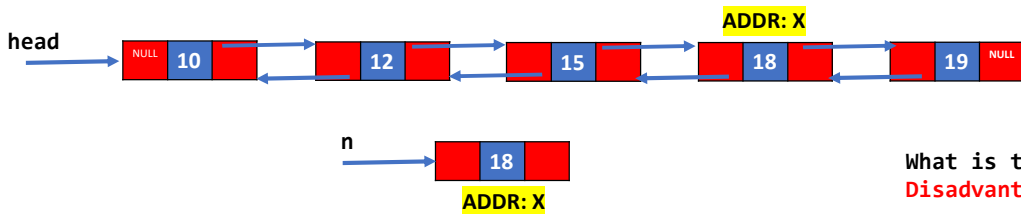
- After deletion



Delete a given node

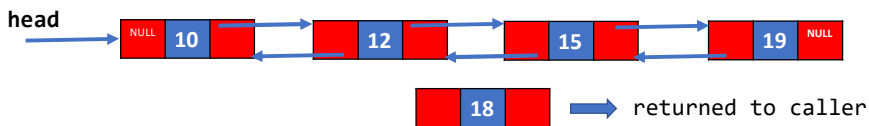
In addition to the address of the next node, we can also store the address of the previous node in a linked list node.

- Before deletion



What is the type?
Disadvantage?

- After deletion



The disadvantage of storing an additional field is the additional memory overhead.

Type

- Doubly linked list node

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

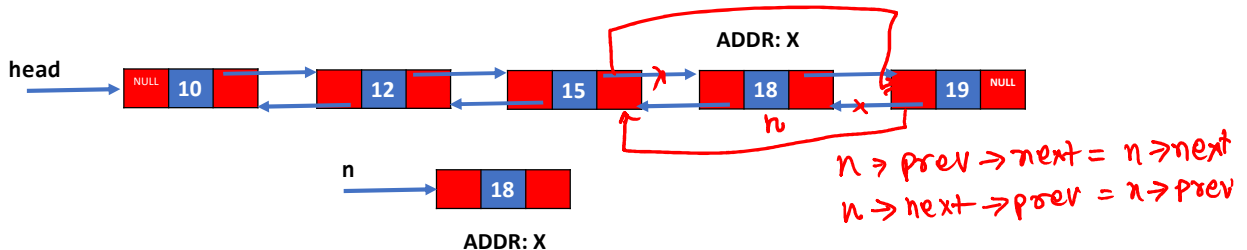
- head

```
struct node *
```

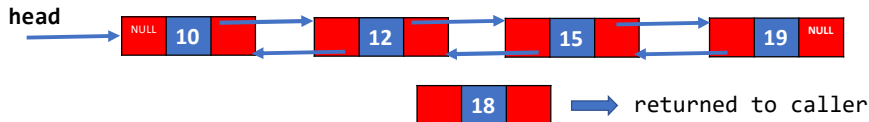
Delete a given node

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

- Before deletion



- After deletion

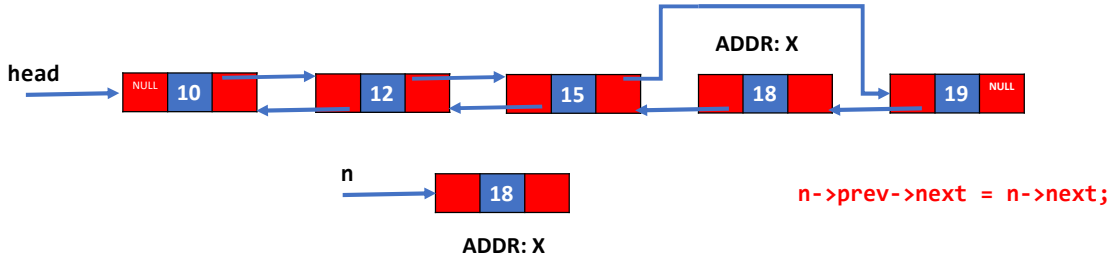


To delete node n , we can simply set the next field of the previous node to the next node after n and the prev field of the next node to the previous node before n . This can be done in $O(1)$ because $n \rightarrow prev$ contains the address of the preceding node, and $n \rightarrow next$ contains the address of the following node.

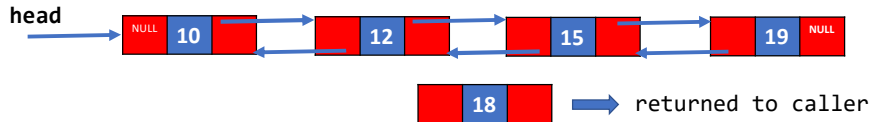
Delete a given node

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

- Before deletion



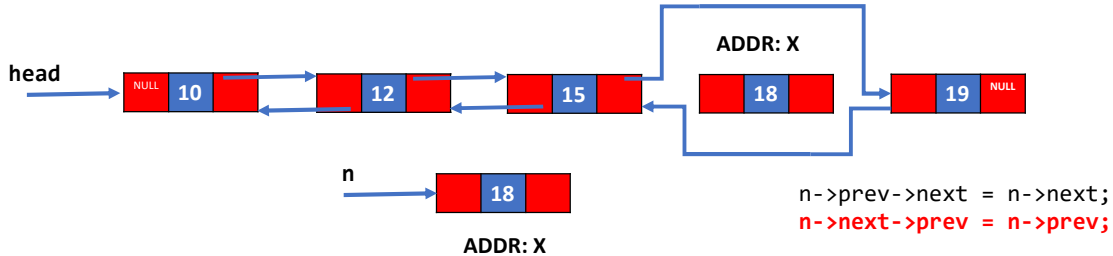
- After deletion



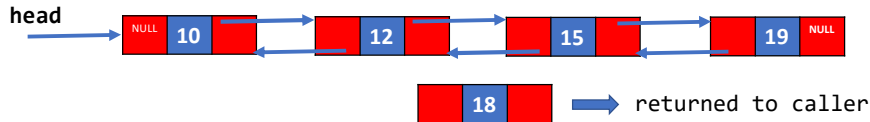
Delete a given node

- Before deletion

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



- After deletion



Delete a given node

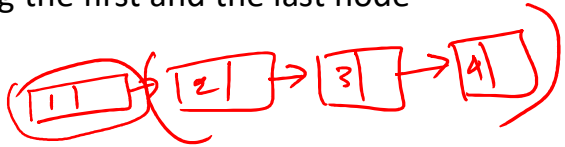
- You need to be careful while deleting the first and the last node

- Deleting the first node

- `head` will change
- `prev` is NULL

- Deleting the last node

- `next` is NULL

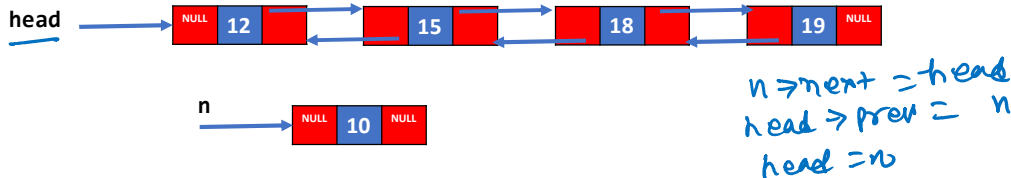


Insert (front)

Insert (front)

- Before insertion

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



- After insertion

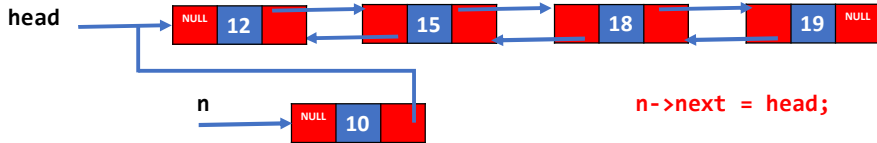


To insert a new node n at the front of a doubly linked list, we need to ensure that $n \rightarrow next$ points to head and $head \rightarrow prev$ points to n before making n the new head.

Insert (front)

- Before insertion

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



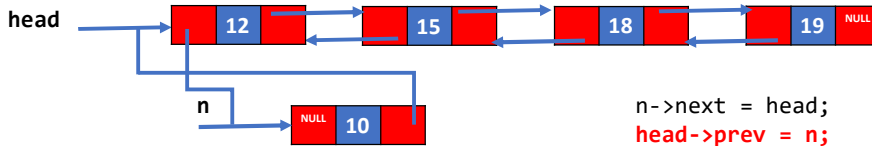
- After insertion



Insert (front)

- Before insertion

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



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

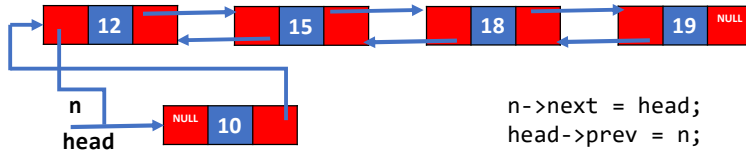
- After insertion



Insert (front)

- Before insertion

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



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

- After insertion



Doubly linked list

- You can try to implement all operations (we did for the singly linked list) for the doubly linked list yourself

Doubly linked list

- In a doubly linked list, **head** points to the first node of the linked list (similar to a singly linked list)
- At any node at position **i**, we can traverse in both directions, i.e., the node at positions **i-1** and **i+1** can be reached in one step
- This data structure efficiently handles the case when given a node we want to access nearby nodes in both directions or delete the node
 - e.g., **Redo/Undo** operations, Playlist with **next** and **previous** buttons

Doubly linked list

- Every node in a doubly linked list contains two references, **prev** and **next**, corresponding to the previous and next node
- The **prev** and **next** pointers of the first and last nodes point to the empty list (**NULL**)

Circular linked list

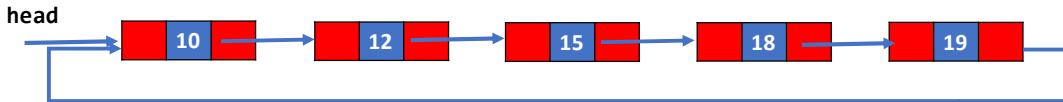
Circular linked list

- Circular linked list can be both singly and doubly
- The **prev** pointer of the first node points to the last node
- The **next** pointer of the last node points to the first node
- In a doubly circular linked list, we can efficiently reach the last node from the first node and vice-versa in just one step

Singly circular linked list

Printing singly circular list

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



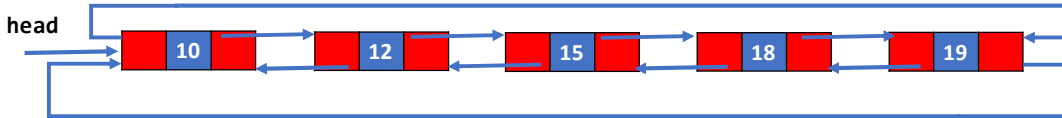
```
if (head == NULL)  
    return;  
print ("d", head->val);  
tmp = head->next;  
while (tmp != head) {  
    printf("d", tmp->val);  
    tmp = tmp->next;  
}
```

In a singly circular linked list, the last node stores a reference to the first node in the next field. To print the values of all nodes, we need to iterate the linked list until we encounter the head again.

Doubly circular linked list

Printing doubly circular list

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

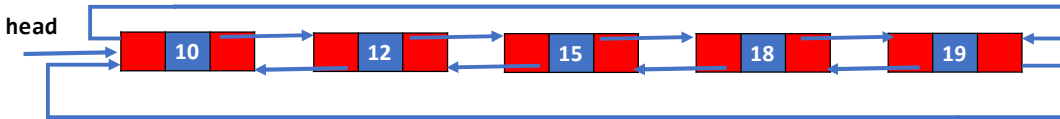


Same as singly linked list

In a singly circular linked list, the last node stores a reference to the first node in the next field, and the first node stores the reference to the last node in the prev field. We can print the doubly list similar to the singly linked list.

Inserting at a given position

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

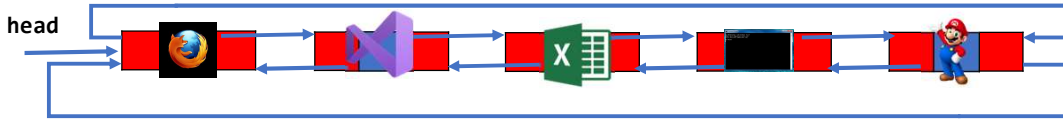


We can insert a node at a given position slightly more efficiently than the singly linked list because we can walk in both directions. If the position is closer to the rear end, we can start walking from the rear end in the reverse direction to reach the node at the desired position; otherwise, we can walk in the forward direction, similar to what we did in the case of singly linked list. To infer which end is closer, we also need to keep track of the total number of elements in the list.

Circular linked list

- You can try to implement all operations (we did for the singly linked list) for the circular linked list yourself

Management of active applications



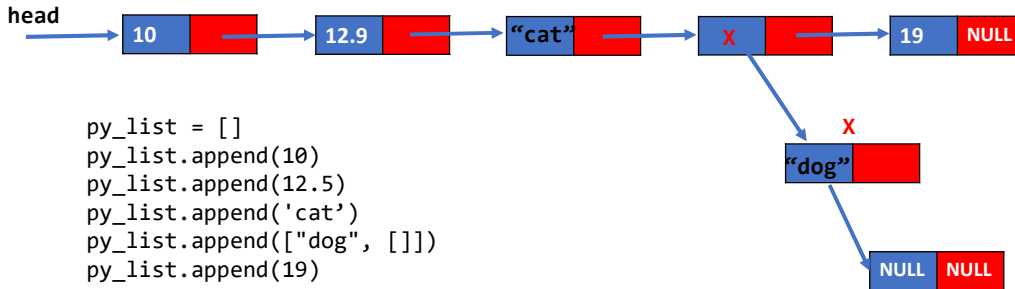
Other application

- Playing a list of songs in a loop

Lists

- In our discussion so far, all the values in the list are of the same type
- How can we support values of different types in the list?

List with values of different type



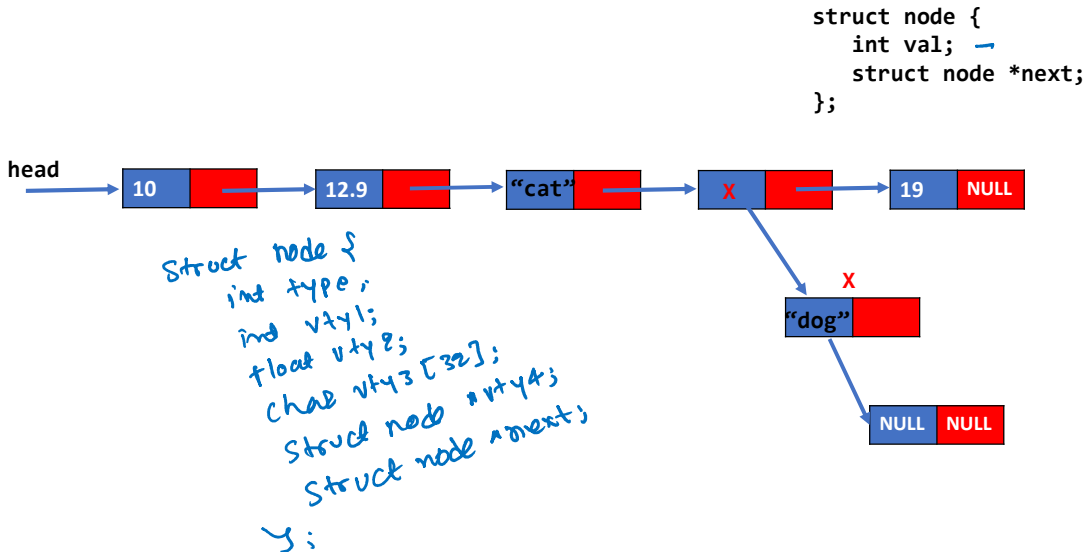
```
py_list = []  
py_list.append(10)  
py_list.append(12.5)  
py_list.append('cat')  
py_list.append(["dog", []])  
py_list.append(19)
```

```
v = py_list[2] / 1  
print(v)
```

What will be the output of this program?

If we compare our lists with the list in Python, we can say that the Python lists are more powerful because we can store elements of different types in the same list. However, the problem with that approach is that if we do some operation on the linked list element that is not allowed by the language, we get runtime errors. For example, the code listed on this slide tries to divide “cat” by one, which will result in a runtime error. These kinds of errors are not possible for the linked list in C. In other words, lists in C are more reliable than lists in Python. Secondly, Python defers these type-checking decisions (whether an operation is allowed on a given value) at runtime, which adds an additional overhead of type-checking during the program execution, making it very inefficient compared to C. On the bright side, it’s easy to build a prototype in Python because you don’t need to worry about the types while writing the program.

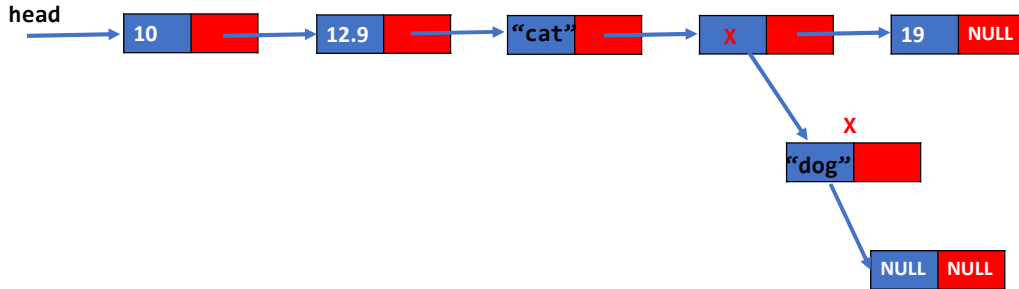
List with values of different type



We can implement lists in C that can support different types in the following way. A node in the linked list can also contain an additional field corresponding to the value type. We also need to insert fields corresponding to all possible types we want to store in the list. When we want to insert a value of a given type, first, we need to set the type field to the type of the value, and then we need to set the field that can store a value of that particular type. Similarly, during the read operation, we first need to read the type from the type field and then the value from the field corresponding to that type.

List with values of different type

```
struct node {  
    int type;  
    int vty1;  
    float vty2;  
    char vty3[32];  
    struct node *vty4;  
    struct node *next;  
};
```



Exercise

- Write an algorithm to find the median of all values in a linked list
- Write an algorithm to check if two lists are identical
- Write an algorithm to reverse a doubly linked list recursively
- Write an algorithm to reverse a doubly linked list iteratively
- Write an algorithm to remove duplicates from an unsorted list
- Write an algorithm to sort a doubly linked list

Stack

References

- Read chapter-10.1 from CLRS
- Read chapter-3.3 from Mark Allen Weiss

Stack

- A stack is a collection of items
- Insertion and deletion are performed at the same end, called the top
- Implements last-in, first-out strategy

Stack of plates



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

Stack ADT

- $\text{STACK-EMPTY}(S) \Rightarrow$ returns true if the stack S is empty
- $\text{PUSH}(S, x) \Rightarrow$ insert an item x on the top of the stack S
- $\text{POP}(S) \Rightarrow$ remove and return the item at the top of the stack S
- $\text{TOP}(S) \Rightarrow$ return the item at the top of the stack S without removing

- ADT hides the actual implementation from the clients
 - Internally, a stack can be implemented using a linked list or an array

Example

Operations	Output	Stack Contents
push(5)		(5)
push(3)		(5 3)
pop()	3	(5)
push(7)		(5 7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	"underflow"	()
top()	"underflow"	()
stack_empty()	TRUE	()
push(9)		(9)
push(7)		(9 7)
push(7)		(9 7 7)
pop()	7	(9 7)
stack_empty()	FALSE	(9 7)

Stack using a singly linked list

Stack

head = NULL

Stack is empty.

Push



head points to the top of the stack.

push(S, 10)

Push



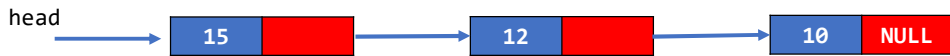
Which list operation is suitable for the push operation?

insert front

push(S, 10)

push(S, 12)

Push



push(S, 10)

push(S, 12)

push(S, 15)

Pop

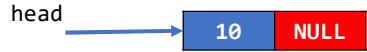


Which list operation is suitable for the pop operation?

delete front()

v = pop(S); // 15

Pop



```
v = pop(S);    // 15  
v = pop(S);   // 12
```

Pop

head = NULL

v = pop(S); // 15

v = pop(S); // 12

v = pop(S); // 10


Pop

head = NULL

The pop operation
on an empty stack
causes underflow.

```
v = pop(S);    // 15  
v = pop(S);    // 12  
v = pop(S);    // 10  
v = pop(S);    // underflow
```

Type of stack



```
struct stack {  
    struct node *head;  
};
```

Creating stack

```
[ struct stack {  
    struct node *head;  
};  
  
// returns a handle to stack  
// all stack operations APIs take this handle as input  
struct stack* create_stack() {  
    struct stack *S = (struct stack*)malloc(sizeof(struct stack));  
    if (S == NULL) {  
        printf("failed to allocate memory!\n");  
        return NULL;  
    }  
    S->head = NULL;  
    return S;  
}
```

The `create_stack` API allocates a node of type “struct stack”. “struct stack” stores all the metadata corresponding to the stack. For a linked list implementation, we just need the head of the list to implement push and pop operations. However, you can also add more fields, e.g., the number of elements in the stack, to implement other APIs. Initially, the head of the linked list is set to NULL, which represents an empty stack.

Stack-empty

```
// returns 1 if stack is empty
// returns 0 if stack is not empty
int stack_empty(struct stack *S) {
    if (S->head == NULL)
        return 1;

    return 0;
}
```

```
struct stack {
    struct node *head;
};
```


Stack-empty

```
// returns 1 if stack is empty
// returns 0 if stack is not empty
int stack_empty(struct stack *S) {
    if (S->head == NULL) {
        return 1;
    }
    return 0;
}
```

Push

```
struct node* allocate_node(int val);  
struct node* insert_front(struct node *head, struct node *n);  
struct stack {  
    struct node *head;  
};
```

```
// allocate a new node with value val  
// insert the new node at the  
// front of the linked list
```

```
void push(struct stack *S, int val) {  
    struct node *n = allocate_node(val);  
    S->head = insert_front(S->head, n);  
}
```

Push

```
struct delete_info {  
    struct node *head;  
    struct node *deleted_node;  
};  
struct delete_info delete_front(struct node *head);  
void free(void *ptr);
```

```
// allocate a new node with value val  
// insert the new node at the  
// front of the linked list
```

```
void push(struct stack *S, int val) {  
    struct node *n = allocate_node(val);  
    S->head = insert_front(S->head, n);  
}
```

Read the insert_front and allocate_node APIs from the previous lectures.

Pop

```
struct delete_info {  
    struct node *head;  
    struct node *deleted_node;  
};  
struct delete_info delete_front(struct node *head);  
void free(void *ptr);
```

```
// delete the first node of the linked list and  
// return its value
```

```
int pop(struct stack *S) {  
    if (stack_empty(S) == 1) {  
        printf("Underflow"),  
        exit(0);  
    }  
    struct delete_info ret = delete_front(S->head);  
    int ret_val = ret.deleted_node->val;  
    free(ret.deleted_node);  
    S->head = ret.head;  
    return ret_val;  
}
```

Pop

```
    struct delete_info {
        struct node *head;
        struct node *deleted_node;
    };
    struct delete_info delete_front(struct node *head);
    void free(void *ptr);

// delete the first node of the linked list and
// return its value

int pop(struct stack *S) {
    if (stack_empty(S)) {
        printf("stack underflow\n");
        exit(0);
    }
    struct delete_info ret = delete_front(S->head);
    struct node *deleted_node = ret.deleted_node;
    int retval = deleted_node->val;
    free(deleted_node);
    S->head = ret.head;
    return retval;
}
```

Read the delete_front API from the previous lectures.

Client (stack ADT)

The client is not expected to access the fields in **S** (returned from an stack ADT API).

The handle **S** is invalid after `dispose_stack`.

```
int main()
{
    struct stack *S = create_stack();
    int i;

    assert(stack_empty(S));
    for (i = 0; i < 10; i++) {
        push(S, i);
        print_stack(S);
    }
    assert(!stack_empty(S));
    for (i = 0; i < 10; i++) {
        pop(S);
        print_stack(S);
    }
    assert(stack_empty(S));
    dispose_stack(S);
    return 0;
}
```

This slide shows a client program that uses the stack ADT. The `create_stack` routine returns a handle to the stack. The other APIs in the stack ADT simply take that handle as input. The client is not supposed to change the values inside the handle. `dispose_stack` deletes all the memory allocated for the stack, i.e., it frees all nodes in the linked list and the memory allocated for the handle.

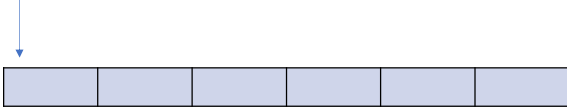
Exercise

- Try to implement the `top(S)` API yourself
- Extend the stack ADT with a new API `size(S)`, which returns the number of elements in the stack
- Implement the `dispose_stack(S)` and `print_stack(S)` APIs from the previous slide

Stack using a fixed size array

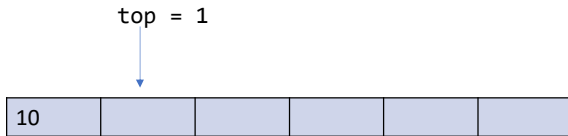
Stack

top = 0



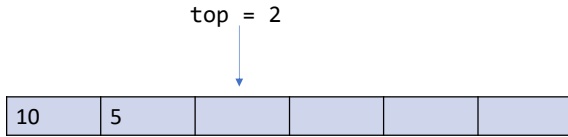
Stack is empty.
Stack size = 6

Stack



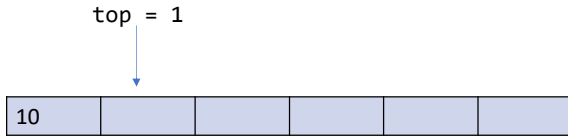
push(10)

Stack



push(10)
push(5)

Stack



```
push(10)  
push(5)  
pop() = 5
```

Stack

top = 0



```
push(10)  
push(5)  
pop() = 5  
pop() = 10
```

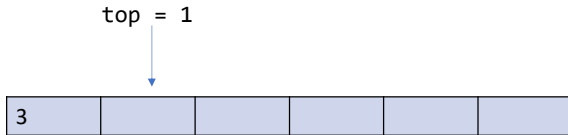
Stack

top = 0



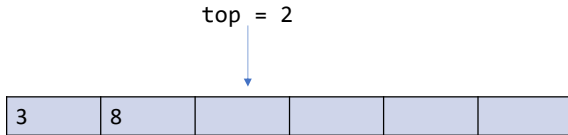
```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
```

Stack



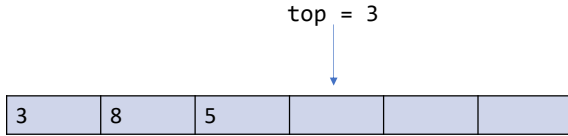
```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
```

Stack



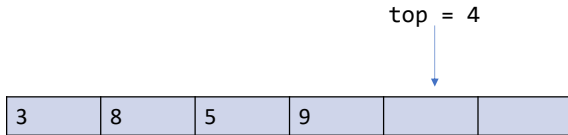
```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
```


Stack



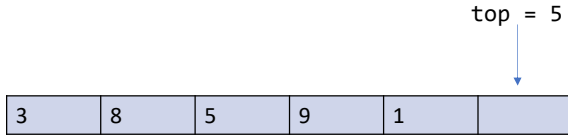
```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
```

Stack



```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
push(9)
```

Stack



```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
push(9)
push(1)
```

Stack

3	8	5	9	1	2
---	---	---	---	---	---

top = 6



```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
```

Stack

3	8	5	9	1	2
---	---	---	---	---	---

top = 6



No **overflow** in the linked list based implementation!

```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) "overflow"
```

Notice that an overflow can happen if we want to push elements beyond the fixed size of the array.

Type of stack

```
struct stack {  
    int top;  
    int capacity;  
    int *arr;  
};
```

Type of stack

```
struct stack {  
    int top;  
    int capacity;  
    int *arr;  
};
```

Initialization

```
struct stack* create_stack(int max_elements) {  
    struct stack *S = (struct stack*)malloc(sizeof(struct stack));  
    if (S == NULL) {  
        printf("failed to allocate memory\n");  
        return NULL;  
    }  
    S->arr = (int*)malloc(max_elements * sizeof(int));  
    if (S->arr == NULL) {  
        printf("failed to allocate memory\n");  
        return NULL;  
    }  
    S->top = 0;  
    S->capacity = max_elements;  
    return S;  
}
```

For the stack using a fixed-size array, the `create_stack` API additionally takes the maximum number of elements in the stack as an argument and allocates an array of the corresponding size. Initially, the top is set to zero, which signifies an empty stack.

Stack-empty

```
// returns 1, if the stack is empty
// returns 0, if the stack is not empty
int stack_empty(struct stack *S) {
    if (S->top == 0)
        return 1;
    return 0;
}
```

```
struct stack {
    int top;
    int capacity;
    int *arr;
};
```

Stack-empty

```
// returns 1, if the stack is empty
// returns 0, if the stack is not empty
int stack_empty(struct stack *S) {
    if (S->top == 0) {
        return 1;
    }
    return 0;
}
```

```
struct stack {
    int top;
    int capacity;
    int *arr;
};
```

Stack-full

```
// returns 1, if the stack is full  
// returns 0, if the stack is not full  
int stack_full(struct stack *S) {
```

```
    if (S->top == S->capacity)  
        return 1;  
    return 0;
```

```
}
```

```
struct stack {  
    int top;  
    int capacity;  
    int *arr;  
};
```

Stack-full

```
// returns 1, if the stack is full
// returns 0, if the stack is not full
int stack_full(struct stack *S) {
    if (S->top == S->capacity) {
        return 1;
    }
    return 0;
}
```

```
struct stack {
    int top;
    int capacity;
    int *arr;
};
```

Push

```
// insert a value (val) at the top of the stack  
void push(struct stack *S, int val) {
```

```
    if ( stack_full(S) == 1 ) {  
        printf ( "Overflow" );  
        exit(1);  
    }  
    S->arr[S->top] = val;  
    S->top ++;
```

```
}
```

```
struct stack {  
    int top;  
    int capacity;  
    int *arr;  
};
```

Push

```
// insert a value (val) at the top of the stack
void push(struct stack *S, int val) {
    if (stack_full(S)) {
        printf("Stack overflow\n");
        exit(0);
    }
    S->arr[S->top] = val;
    S->top += 1;
}
```

```
struct stack {
    int top;
    int capacity;
    int *arr;
};
```

Pop

```
// remove an element from the top of the stack
```

```
// at returns its value
```

```
int pop(struct stack *S) {
```

```
    if (stack_empty(S) == 1) {  
        printf("Underflow");  
        exit(0);  
    }
```

```
    int retval = S->arr[S->top - 1];  
    S->top --;  
    return retval;  
}
```

```
struct stack {  
    int top;  
    int capacity;  
    int *arr;  
};
```

Pop

```
// remove an element from the top of the stack
// at returns its value
int pop(struct stack *S) {
    if (stack_empty(S)) {
        printf("Stack underflow\n");
        exit(0);
    }
    int ret = S->arr[S->top - 1];
    S->top -= 1;
    return ret;
}
```

```
struct stack {
    int top;
    int capacity;
    int *arr;
};
```


Eliminating stack overflow

Eliminating stack overflow

- How can we eliminate the overflow of a stack?

Eliminating stack overflow

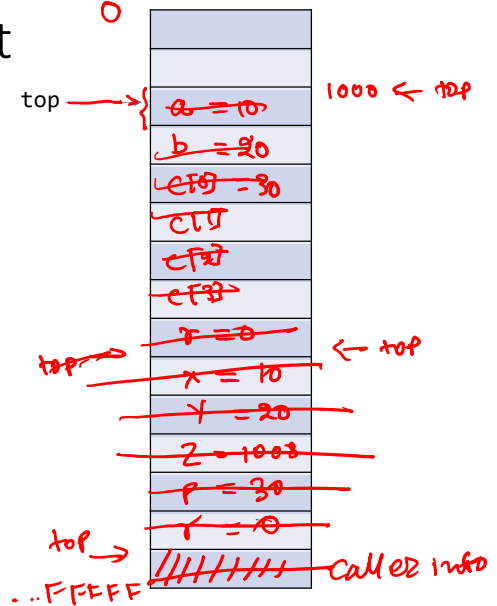
- How can we eliminate the overflow of a stack?
 - Use a dynamic array instead of a fixed-size array

Applications of stack

0

top

Let's say each slot is four bytes long.



Local variables management

- Have you ever noticed that the value of some other variable is changed when you do some out-of-bound array access?

Because the variables are allocated consecutively on the stack, an out-of-bounds memory access can change the values of the other variables.

Local variables management

- Have you ever encountered a stack overflow error while running your program?

Notice that the application uses a fixed-size stack for local variable allocation and deallocation. So, if you run a recursive routine for an input that may cause a very large depth of the recursion, then the variables of all callers remain allocated on the stack until we reach the base case. This may cause a stack overflow situation when the memory usage exceeds the size of the stack. For a similar reason, we should avoid allocating large arrays on the stack because that may also cause an overflow. These bugs are hard to detect because you will not get any warning during the allocation even if the allocation fails.

Local variables management

- Have you ever encountered a stack overflow error while running your program?
 - The application stack is implemented using a fixed-size array
 - The array size is big enough to support most applications
 - You can change the default size of the stack using some compiler option

Eliminating stack overflow

- Can we use a dynamic array to eliminate stack overflow?

This will be a homework exercise.

Matching symbols

Matching symbols

- Three kinds of symbols
 - Parentheses: “(” and “)”
 - Braces: “{” and “}”
 - Brackets: “[” and “]”
- Goal: check if each opening symbol matches the corresponding closing symbol

Matching symbols

- Correct: $(a + b) + (e * (d - \{c + \{a / b\}\})) + [c * d] \{x + [y - z]\}$
- Incorrect: $) (a + b)($
- Incorrect: $\{a + (b + c)\}$

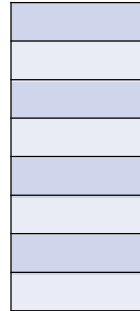
Basic idea

- Scan the string from left to right
- If an opening symbol is encountered, push it to stack
- If a closing symbol is encountered
 - If the stack is empty, return false
 - pop a symbol from the stack
 - if the popped symbol is not equal to the corresponding closing symbol, return false
- Skip other characters
- If the end of input is reached
 - If the stack is not empty, return false; otherwise, return true

Matching symbols

$$(a + b) * \{[c + d]\} * z$$

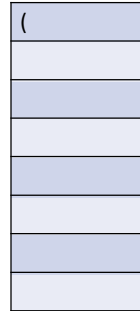
$\text{pop}() = '('$
 $\text{pop}() = '['$
 $\text{pop}() = '\{'$



Matching symbols

$$(a + b) * \{ [c + d] \} * z$$

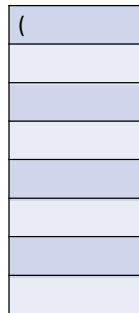
push (



Matching symbols

$$(a + b) * \{[c + d]\} * z$$

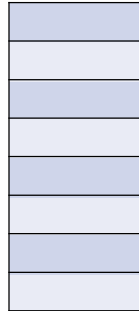
push (
skip a
skip +
skip b



Matching symbols

$(a + b) * \{ [c + d] \} * z$

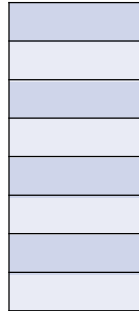
push (
skip a
skip +
skip b
pop == (



Matching symbols

$(a + b) * \{ [c + d] \} * z$

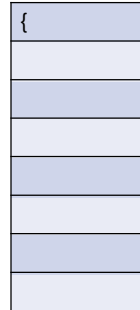
push (
skip a
skip +
skip b
pop == (
skip *



Matching symbols

$$(a + b) * \{[c + d]\} * z$$

```
push (
skip a
skip +
skip b
pop == (
skip *
push {
```



Matching symbols

$$(a + b) * \{[c + d]\} * z$$

```
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
```

[illegible]

Matching symbols

$$(a + b) * \{ [c + d] \} * z$$

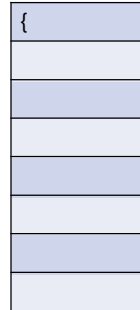
```
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
skip c
skip +
skip d
```

[illegible]

Matching symbols

$$(a + b) * \{[c + d]\} * z$$

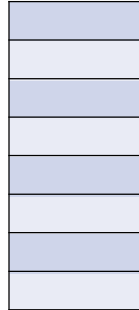
```
push (
  skip a
  skip +
  skip b
pop == (
  skip *
push {
  push [
  skip c
  skip +
  skip d
pop == [
```



Matching symbols

$(a + b) * \{ [c + d] \} * z$

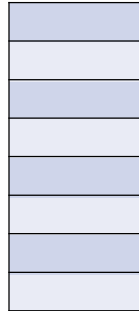
```
push (  
skip a  
skip +  
skip b  
pop == (  
skip *  
push {  
push [  
skip c  
skip +  
skip d  
pop == [  
pop == {
```



Matching symbols

$(a + b) * \{ [c + d] \} * z$

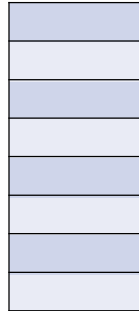
```
push (  
  skip a  
  skip +  
  skip b  
  pop == (  
    skip *  
    push {  
      push [  
        skip c  
        skip +  
        skip d  
      pop == [  
        pop == {  
          skip *  
        skip z  
      stack_empty == TRUE
```



Matching symbols

)(a + b)

*stack_empty()
 return false*

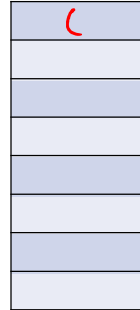


Matching symbols

 $((a + b)$

POPC() = '('

at the end stack not empty
return false



Pseudo-code

Pseudo-code

- Read “Pseudocode conventions” from page-20 of the CLRS book

Pseudo-code

- Pseudo-code is a mixture of programming language constructs and natural language
 - In pseudo-code, we can mix C semantics with a little bit of natural language for brevity
- There is no precise definition of a pseudo-code because of its dependence on the natural language
- Next, we are going to define the syntax of pseudo-code that we will follow in this course

Pseudo-code

- **for** and **while** statements are mostly similar to C
 - `for i = 2 to n` is the same as `for (i = 2; i <= n; i++)`
 - Use **indentation** instead of **braces** for a block of statements inside a loop (similar to Python)
 - Terminating semicolon can be omitted
 - A function can return multiple values

```
sum = 0
fact = 1
for i = 1 to n
    sum = sum + i
    fact = fact * i
return sum, fact
```

```
sum = 0
fact = 1
i = 1
while i < n
    sum = sum + i
    fact = fact * i
return sum, fact
```

Pseudo-code

- If the types of the variables can be inferred using their use, there is no need to declare the variable

```
sum = 0
fact = 1
for i = 1 to n
    sum = sum + i
    fact = fact * i
return sum, fact
```


Pseudo-code

- Function declaration: `Name(arg1, arg2, ...)`
 - followed by the description of input parameters and the return values
 - Use `//` for comments

`SUM_FACTORIAL(A, n)`

`// input A is an array of integers`

`// n is an integer`

`// returns the sum of numbers from 1 to n and factorial n`

Pseudo-code

- Conditional statements are the same as C
 - Use **indentation** instead of **braces** for a block of statements inside a condition (similar to Python)
 - Assignment (=) and comparison (==, <=, <, >=, >, !=) operators are the same as C
 - Array indexing is the same as C

```
if (A[i] > A[j])
    t = A[i]
    A[i] = A[j]
    A[j] = t
else if (A[i] == A[j])
    A[i] = A[j] = 0
else
    A[i] = A[j] - A[i]
```

Pseudo-code

- Boolean operator “and” is used in place of “&&” from C
- Boolean operator “or” is used in place of “||” from C
- You can also use TRUE and FALSE directly in the pseudo-code

```
if (A[i] > A[j] and A[j] < A[k])
    t = A[i]
    A[i] = A[j]
    A[j] = t
else if (A[i] == A[j] or A[j] == A[k] or t == TRUE)
    A[i] = A[j] = 0
else
    A[i] = A[j] - A[i]

return FALSE
```

Pseudo-code

- Function calls are the same as C
- A function can return multiple values

```
s, fact = SUM_FACTORIAL(A, 20)  
// s contains the first return value  
// fact contains the second return value
```

Pseudo-code

- Structure accesses are the same as C

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

```
SUM_LIST(head)  
// head is pointer to the input list of type struct Node  
// returns the sum of all elements of the list
```

```
sum = 0  
while (head != NULL)  
    sum = sum + head->val  
    head = head->next  
return sum
```

Pseudo-code

- You can also define a function using a set of *assume* statements

```
assume(factorial(0) == 1)
```

```
assume(factorial(n) == n * factorial(n-1))
```

```

MatchingSymbols(Str, n):
    // Str is an array of n characters
    // returns TRUE if all symbols are matching; otherwise, returns FALSE
    S = create_stack()
    assume(rev('(')) == '(')
    assume(rev('{')) == '{')
    assume(rev('[')) == '[')
    for i = 0 to n - 1
        if (Str[i] == '(' || Str[i] == '{' || Str[i] == '[')
            S.push(Str[i])
        else if (Str[i] == ')' || Str[i] == '}' || Str[i] == ']')
            if S.isEmpty()
                dispose_stack(S)
                return FALSE
            else if S.pop() != rev(Str[i])
                dispose_stack(S)
                return FALSE
    if S.isEmpty()
        dispose_stack(S)
        return TRUE
    else
        dispose_stack(S)
        return FALSE

```

Matching symbols

Queues

References

- Read section-10.1 from the CLRS book
- Read section-3.4 from Mark Allen Weiss

Queue



[This Photo](#) by Unknown Author is licensed under [CC BY](#)

In real life, most of the lines you stand in are queues.

Queue

- Queue is the collection of items
- Queue has two ends **head** and **tail**
- **Insert** items at the **head** of the queue
- **Remove** items from the **tail** of the queue
- Implements first-in, first-out strategy

Queue

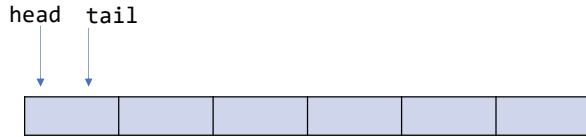
- Follow first-in, first-out (FIFO) policy
- Queue ADT:
 - QUEUE-EMPTY => returns true if the queue is empty
 - ENQUEUE => insert an item at the tail of the queue
 - DEQUEUE => remove and return the item at the head of the queue
 - FIRST => return the first item in the queue without removing

Example

Operations	Output	Queue contents
enqueue(5)		(5)
enqueue(3)		(5 3)
dequeue()	5	(3)
enqueue(7)		(3 7)
dequeue()	3	(7)
first()	7	(7)
dequeue()	7	()
dequeue()	"underflow"	()
first()	"underflow"	()
queue_empty()	TRUE	()
enqueue(9)		(9)
enqueue(7)		(9 7)
enqueue(7)		(9 7 7)
dequeue()	9	(7 7)
queue_empty()	FALSE	(7 7)

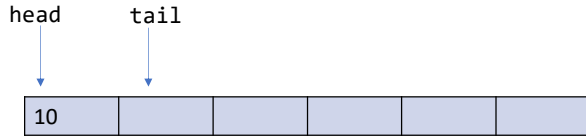
Queue (using array)

Queue



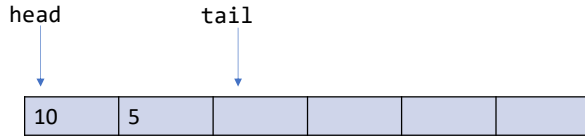
Queue is empty.
Queue size = 6

Queue



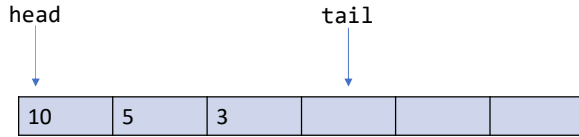
enqueue(10)

Queue



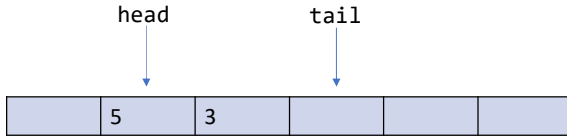
enqueue(10)
enqueue(5)

Queue



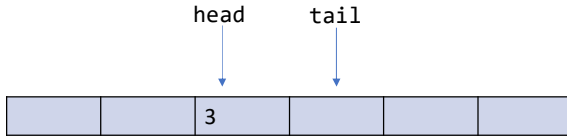
```
enqueue(10)  
enqueue(5)  
enqueue(3)
```

Queue



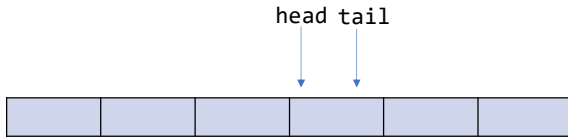
```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
```

Queue



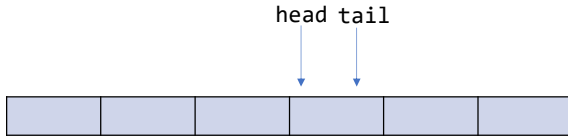
```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
```

Queue



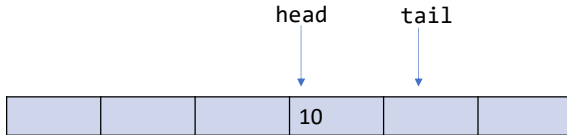
```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
```

Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

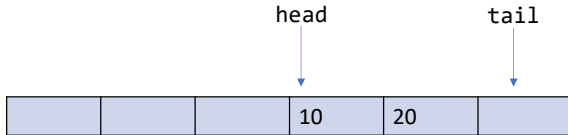
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

enqueue(10)

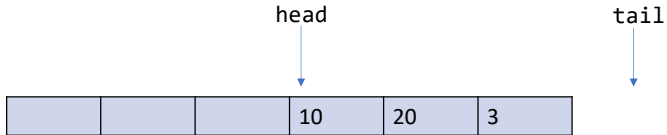
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
```

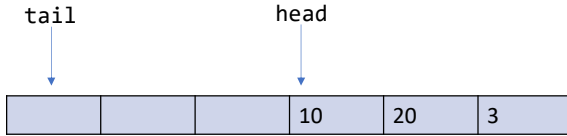

Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
```

Queue

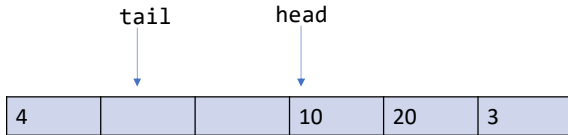


After an enqueue operation, if the value of the tail reaches the end of the array, it is wrapped around to the zeroth index.

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
```

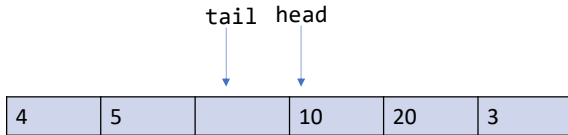
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
```

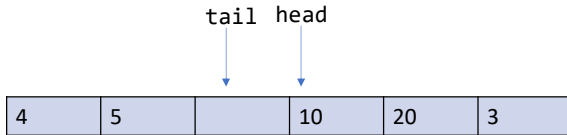
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
```

Queue

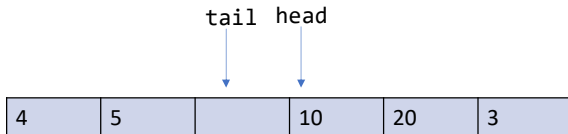


Should we allow the tail to move further?

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
enqueue(6)
```

Queue



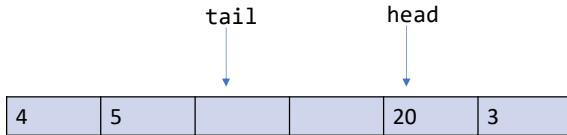
```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
enqueue(6) = "Overflow"
```

Should we allow the tail to move further?

No, because in this case, the tail will be equal to the head, which satisfies the condition for an empty queue.

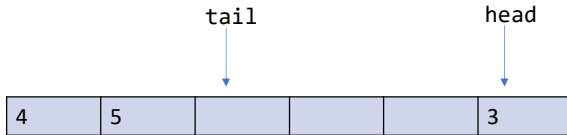
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
enqueue(6) = "Overflow"
dequeue() = 10
```

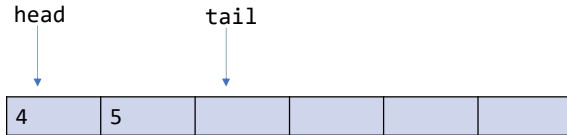
Queue



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
enqueue(6) = "Overflow"
dequeue() = 10
dequeue() = 20
```


Queue



After an dequeue operation, if the value of the head reaches the end of the array, it is wrapped around to the zeroth index.

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(4)
enqueue(5)
enqueue(6) = "Overflow"
dequeue() = 10
dequeue() = 20
```

```
dequeue() = 3
```

Type of queue

```
struct queue {  
    int head;  
    int tail;  
    int capacity;  
    int arr;  
};
```

Type of queue

```
struct queue {  
    int head;  
    int tail;  
    int capacity;  
    int *arr;  
};
```

Initialize queue

```
struct queue* create_queue(int max_elements) {  
    struct queue *Q = (struct queue*)malloc(sizeof(struct queue));  
    if (Q == NULL) {  
        printf("Failed to allocate memory!\n");  
        return NULL;  
    }  
    Q->arr = (int*)malloc(max_elements * sizeof(int));  
    if (Q->arr == NULL) {  
        printf("Failed to allocate memory!\n");  
        return NULL;  
    }  
    Q->head = Q->tail = 0;  
    Q->capacity = max_elements;  
    return Q;  
}
```

Queue-empty

```
// returns 1, if the queue is empty
// returns 0, if the queue is full
int queue_empty(struct queue *Q) {
    if (Q->head == Q->tail)
        return 1;
    return 0;
}
```

```
struct queue {
    int head; -
    int tail; -
    int capacity; -
    int *arr;
};
```

Queue-empty

```
// returns 1, if the queue is empty
// returns 0, if the queue is full
int queue_empty(struct queue *Q) {
    if (Q->head == Q->tail) {
        return 1;
    }
    return 0;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

Next-head

```
// returns the value of new value head after a  
// successful dequeue operation  
int next_head(struct queue *Q)
```

```
{  
    int next_head = Q->head + 1;  
    if (next_head == Q->capacity)  
        return 0;  
    return next_head;  
}
```

```
struct queue {  
    int head;  
    int tail;  
    int capacity;  
    int *arr;  
};
```

Next-head

```
// returns the value of new value head after a
// successful dequeue operation
int next_head(struct queue *Q)
{
    if (Q->head == Q->capacity - 1) {
        return 0;
    }
    return Q->head + 1;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

The next_head routine returns the possible value of head after a dequeue operation. This routine doesn't modify the value of head.

Next-tail

```
// returns the value of new value tail after a  
// successful enqueue operation  
int next_tail(struct queue *Q)
```

```
{
```

```
    if (Q->tail == Q->capacity - 1)  
        return 0;
```

```
    return Q->tail + 1;
```

```
}
```

```
struct queue {  
    int head;  
    int tail;  
    int capacity;  
    int *arr;  
};
```

Next-tail

```
// returns the value of new value tail after a
// successful enqueue operation
int next_tail(struct queue *Q)
{
    if (Q->tail == Q->capacity - 1) {
        return 0;
    }
    return Q->tail + 1;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

The next_tail routine returns the possible value of tail after an enqueue operation. This routine doesn't modify the value of tail.

Queue-full

```
// returns 1, if the queue is full
// returns 0, if the queue is not full
int queue_full(struct queue *Q) {
    if (next_tail(Q) == Q->head)
        return 1;
    return 0;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

Queue-full

```
// returns 1, if the queue is full
// returns 0, if the queue is not full
int queue_full(struct queue *Q) {
    if (next_tail(Q) == Q->head) {
        return 1;
    }
    return 0;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

The queue is full when the next value of tail after an enqueue is equal to the head, which is the condition for an empty queue. We are keeping one buffer element to distinguish between empty and full queues.

Enqueue

```
// insert a new value (val) in the queue  
void enqueue(struct queue *Q, int val) {
```

```
    if (queue_full(Q) == 1) {  
        printf("Overflow");  
        exit(0);  
    }
```

```
    Q->arr[Q->tail] = val;  
    Q->tail = next_tail(Q);  
}
```

```
struct queue {  
    int head;  
    int tail;  
    int capacity;  
    int *arr;  
};
```

Enqueue

```
// insert a new value (val) in the queue
void enqueue(struct queue *Q, int val) {
    if (queue_full(Q)) {
        printf("Queue overflow\n");
        exit(0);
    }
    Q->arr[Q->tail] = val;
    Q->tail = next_tail(Q);
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

After inserting the value at index `Q->tail`, we need to update `Q->tail` with the return value of `next_tail`.

Deque

```
// remove and return the oldest value from the queue
int dequeue(struct queue *Q) {
    if (Queue_empty(Q)) {
        printf("Underflow");
        exit(0);
    }
    int retval = Q->arr[Q->head];
    Q->head = next_head(Q);
    return retval;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

Deque

```
// remove and return the oldest value from the queue
int dequeue(struct queue *Q) {
    if (queue_empty(Q)) {
        printf("Queue underflow\n");
        exit(0);
    }
    int ret = Q->arr[Q->head];
    Q->head = next_head(Q);
    return ret;
}
```

```
struct queue {
    int head;
    int tail;
    int capacity;
    int *arr;
};
```

After saving the value at index `Q->head` in `ret`, we need to update `Q->head` with the return value of `next_head`.

Example

```
struct queue {  
    int head;  
    int tail;  
    int size;  
    int *arr;  
};
```

```
int main() {  
    int i;  
    struct queue *Q = create_queue(10);  
  
    assert(queue_empty(Q));  
    for (i = 0; i < 9; i++) {  
        enqueue(Q, i);  
        print_queue(Q); // prints all elements  
    }  
    assert(!queue_empty(Q));  
    for (i = 0; i < 9; i++) {  
        dequeue(Q);  
        print_queue(Q);  
    }  
    assert(queue_empty(Q));  
    dispose_queue(Q); // free all memory  
    return 0;  
}
```

Exercise

- Implement `print_queue` and `dispose_queue` APIs from the previous example
- Implement `FIRST(Q)` API from the queue ADT

Overflow

- Can we prevent overflow in the array implementation of a queue?

Overflow

- Can we prevent overflow in the array implementation of a queue?
 - Yes, we can use dynamic arrays

Queue using linked list

Queue

head = NULL

Empty queue

Enqueue



`enqueue(12)`

Enqueue



enqueue(12)
enqueue(15)

Enqueue

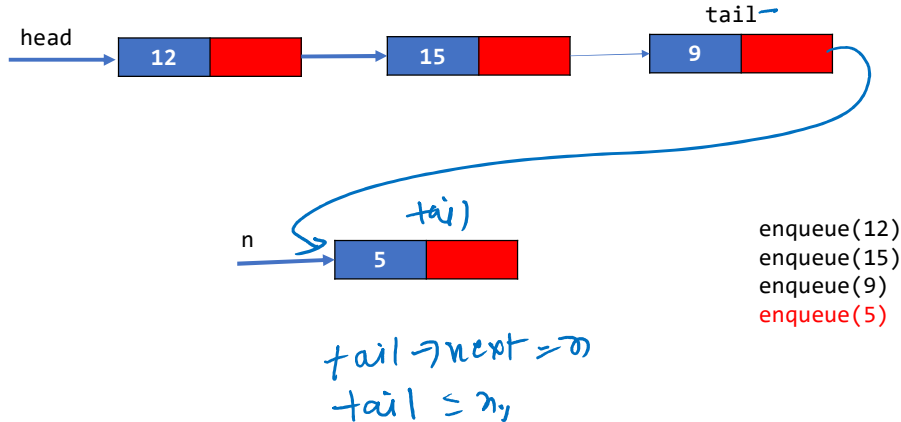


Which list operation is suitable for enqueue?

insert_at_end

enqueue(12)
enqueue(15)
enqueue(9)

Enqueue



To efficiently implement insert_rear, we can keep another variable tail that stores the address of the last node. To insert a new node n at the rear end, first, we need to set tail->next to n , followed by storing the value of n in tail. This is needed because the next element will be inserted after n .

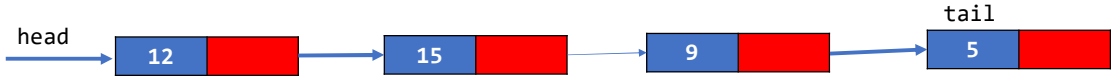
Enqueue



`tail->next = n`
`tail = n`

`enqueue(12)`
`enqueue(15)`
`enqueue(9)`
`enqueue(5)`

Enqueue



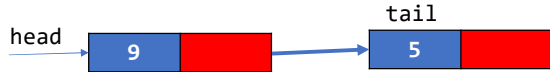
```
enqueue(12)  
enqueue(15)  
enqueue(9)  
enqueue(5)
```

Deque



```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
```

Deque



Which list operation is suitable for deque?

delete front

```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
dequeue() = 15
```

Exercise

- Implement Queue ADT using a linked-list

Application of queues

- Train or flight booking software
- Buffer in devices, e.g., NIC, keyboard, etc.
- Scheduling of applications
- Access to shared resources
- Queue of network packets or disk blocks inside the OS