Question 1

The operator used for dereferencing is :

a) *

b) &

c) ->

d) –>>

Question 2

int *a = new int;

// how would you assign a value 5 to the memory allocated ?

*a = 5

int *a = new int(5)

Question 3

What will happen in this code?

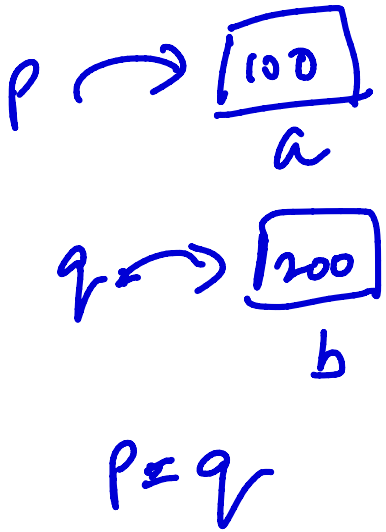
```
int a = 100, b = 200;
int *p = &a, *q = &b;
 p = q;
```

a) b is assigned to a

b) p now points to b

c) a is assigned to b

d) q now points to a

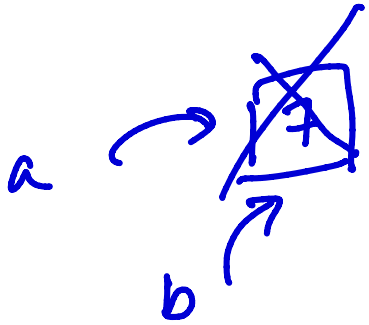


Question 4

```
int *a = new int;

*a = 7;

int *b = a;

delete a;
```

What would b point to now?

undefined behaviour



Question 5

Consider a structure:

```
struct Book {

string name;

};
```

Book b;
(if you want to create an instance of this struct statically)

```
Book* b = new Book;

b->name = "book1";

cout << b->name << endl;

cout << (*b).name << endl;
```

} Same output

Question 6

Consider the linked list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL

Head pointing to the first node with value 1.

The following operation is performed:

```
printLinkedList(head); //Assume this is a function which prints out the linked list
head = head -> next -> next -> next;
printLinkedList(head);
```

What would be the output ?

Question 7

Push the linked list keys into the Stack

Consider a linked list:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL

Consider a Stack whose maximum capacity is 5;

```
Stack s;
Node* temp = head;
while (temp != NULL) {
    if (!isFull(s)) {
        s.push(temp->data);
    }
    temp = temp->next
}
```

What would be the top of the stack? Is there something missing in between the code lines?

5

Question 8

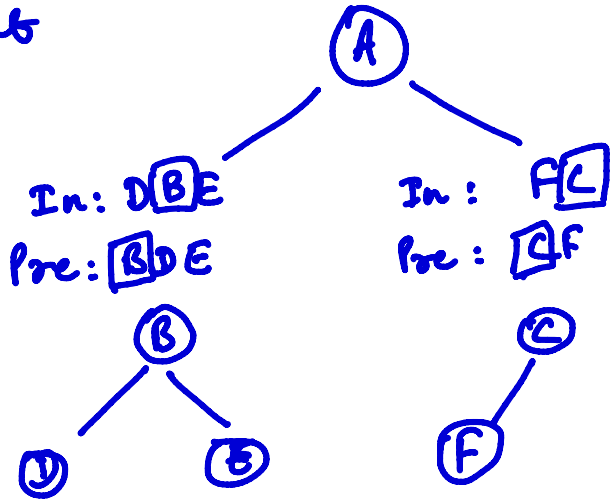
How to reconstruct a tree from the following Inorder and Preorder traversals?

In → Left Root Right

Pre → Root Left Right

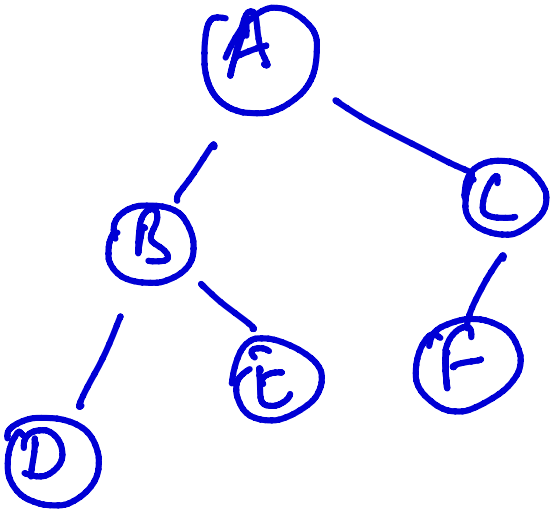
Inorder : DBEAFC

Preorder : ABDECF



Question 8

Can you give me a Postorder traversal of the reconstructed tree?



Left Right Root

D E B F C A

Question 9

Let’s write a C++ Program to check for balanced parentheses in an expression using stack.

Given an expression as string comprising of opening and closing characters of parentheses - (), curly braces - {} and square brackets - [], we need to check whether symbols are balanced or not.

Ex → ({ [ ] })]

() [] {} ∈ Pairs

Question 9 ..

S = "[{()}]"

For Example -

Is this balanced? - "[()]{}{[()()]()}"

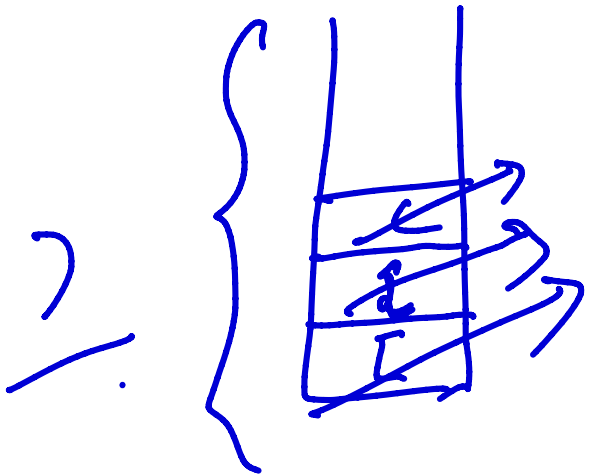
Yes

open → {{{(

closed →)}]

Is this ? - "[(])"

No



```
bool AreParanthesesBalanced(string exp)
{
        stack<char>  S;
        for(int i =0;i<exp.length();i++)
        {
                if(exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
                        S.push(exp[i]);
                else if(exp[i] == ')' || exp[i] == '}' || exp[i] == ']')
                {
                        if(S.empty() || !ArePair(S.top(),exp[i]))
                                return false;
                        else
                                S.pop();
                }
        }
        return S.empty() ? true:false;
}
```

A ? B : C

⇓

It's value is B if A is true and C if A is false

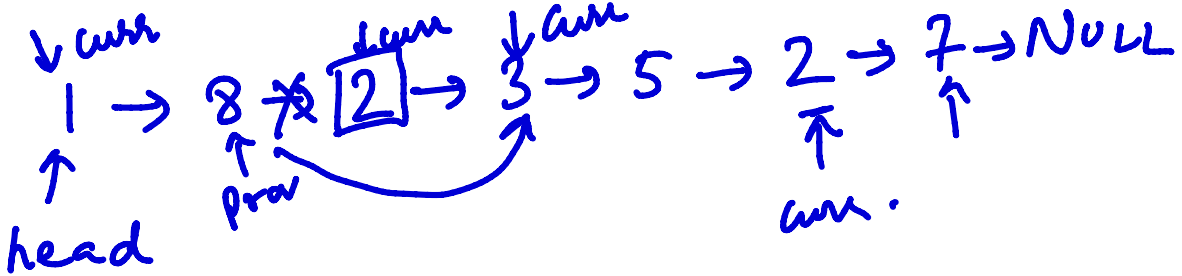
Question 10

Delete all occurrences of a given key in a linked list.

Input: 1 -> 8 -> 2 -> 3 -> 5-> 2 -> 7

Key to delete = 2

Output: 1 -> 8 -> 3 -> 5 -> 7




```
prev = NULL
current = head
while (curr != NULL) {
    while (current -> data != 2)
    { prev = curr
      curr = curr -> next
    }
    prev -> next =
        curr -> next
    delete (curr)
    curr = prev -> next
}
```

this is just a pseudo code, corner cases not handled

Check out this function in next slide!

```
void LinkedList::deleteKey(int key)
{
    // Store head node
    Node* temp = head;
    Node *prev = NULL;

    // If head node itself holds the key or multiple occurrences of key
    while (temp != NULL && temp->data == key)
    {
        head = temp->next;   // Changed head
        delete(temp);               // free old head
        temp = head;          // Change Temp
    }

    // Delete occurrences other than head
    while (temp != NULL)
    {
        // Search for the key to be deleted, keep track of the
        // previous node as we need to change 'prev->next'
        while (temp != NULL && temp->data != key)
        {
            prev = temp;
            temp = temp->next;
        }

        // If key was not present in linked list
        if (temp == NULL) return;

        // Unlink the node from linked list
        prev->next = temp->next;

        delete(temp);  // Free memory

        //Update Temp for next iteration of outer loop
        temp = prev->next;
    }
}
```

Comments are self-explanatory!

"deletekey" is a function inside the LinkedList class (as we created in our exercise)