- **Ques 9**

```cpp
// Function to reverse the first K nodes in the doubly linked list
Node* reverseKNodes(Node* head, int K) {
    if (!head) return nullptr;
    Node* current = head;
    Node* next = nullptr;
    Node* prev = nullptr;
    int count = 0;
    // Check if there are at least K nodes in the list
    Node* check = head;
    for (int i = 0; i < K; ++i) {
        if (!check) return head; // If fewer than K nodes, return head
        check = check->getNext();
    }

    // Reverse the first K nodes
    while (current && count < K) {
        next = current->getNext();
        current->setNext(prev);
        current->setPrev(next);
        prev = current;
        current = next;
        count++;
    }
    // Now head is the last node in the reversed block, connect it to the next K+1 node
    if (next) {
        head->setNext(reverseKNodes(next, K));
        if (head->getNext()) head->getNext()->setPrev(head);
    }
    // prev is now the new head of the reversed block
    return prev;
}
```

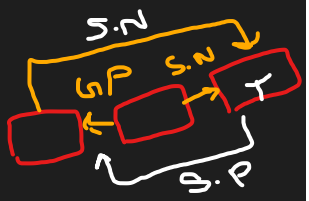
```
//    // Function to add a node to the end of the list
//    void append(int data) {
//        Node* newNode = new Node(data);
//        if (head == nullptr) {
//            head = newNode;  // If the list is empty, new node becomes the head
//        }
//        else {
//            Node* temp = head;
//            while (temp->getNext() != nullptr) {
//                temp = temp->getNext();  // Traverse to the last node
//            }
//            temp->setNext(newNode);  // Add the new node at the end
//        }
//    }
```

Merging Sort

```cpp
// Function to merge two sorted linked lists into a third list
void mergeSortedLists(LinkedList& list1, LinkedList& list2, LinkedList& list3) {
    Node* ptr1 = list1.head;  // Pointer to traverse list1
    Node* ptr2 = list2.head;  // Pointer to traverse list2

    // Traverse both lists and insert nodes in sorted order
    while (ptr1 != nullptr && ptr2 != nullptr) {
        if (ptr1->getData() <= ptr2->getData()) {
            list3.append(ptr1->getData());  // Add the smaller value to list3
            ptr1 = ptr1->getNext();         // Move to the next node in list1
        }
        else {
            list3.append(ptr2->getData());  // Add the smaller value to list3
            ptr2 = ptr2->getNext();         // Move to the next node in list2
        }
    }

    // If there are remaining nodes in list1, add them to list3
    while (ptr1 != nullptr) {
        list3.append(ptr1->getData());
        ptr1 = ptr1->getNext();
    }

    // If there are remaining nodes in list2, add them to list3
    while (ptr2 != nullptr) {
        list3.append(ptr2->getData());
        ptr2 = ptr2->getNext();
    }
}
```

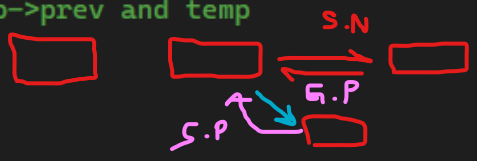
- **Ques 4**
- **Delete second last node Circular Doubly**

```cpp
// 2. Delete the second last node of the list
void deleteSecondLast() {
    // If the list is empty or has less than 2 nodes, return
    if (head == nullptr || head->getNext() == head) {
        cout << "Cannot delete second last node, not enough nodes." << endl;
        return;
    }

    Node* secondLast = tail->getPrev();     // Second last node
    // If the list has only two nodes, we delete the head
    if (secondLast == head) {
        head = tail;                    // Head becomes the tail
        head->setNext(head);            // Update the circular link
        head->setPrev(head);
        delete secondLast;              // Delete the original head
    }
    else {
        secondLast->getPrev()->setNext(tail);  // Link previous node to tail
        tail->setPrev(secondLast->getPrev());  // Link tail back to previous node
        delete secondLast;                // Delete the second last node
    }
}
```



```cpp
// 1. Insert a node while maintaining sorted order
void insertSorted(int data) {
    Node* newNode = new Node(data);  // Create a new node with the given data

    // Case 1: List is empty
    if (head == nullptr) {
        head = tail = newNode;       // The new node is both head and tail
        head->setNext(head);         // Point to itself (circular)
        head->setPrev(head);         // Circular doubly linked list
    }
    // Case 2: Insert at the beginning (new data is smaller than head)
    else if (data <= head->getData()) {
        newNode->setNext(head);      // New node points to current head
        newNode->setPrev(tail);      // New node points to current tail
        head->setPrev(newNode);      // Head points back to new node
        tail->setNext(newNode);      // Tail points to new node (circular link)
        head = newNode;              // Update head to new node
    }
    // Case 3: Insert at the end (new data is greater than tail)
    else if (data >= tail->getData()) {
        newNode->setNext(head);      // New node points to head (circular)
        newNode->setPrev(tail);      // New node points to current tail
        tail->setNext(newNode);      // Tail points to new node
        head->setPrev(newNode);      // Head points back to new node
        tail = newNode;              // Update tail to new node
    }
    // Case 4: Insert in the middle
    else {
```

```cpp
    // Case 4: Insert in the middle
    else {
        Node* temp = head;
        // Traverse the list to find the correct position to insert
        while (temp->getData() < data) {
            temp = temp->getNext();
        }
        // Insert the new node between temp->prev and temp
        newNode->setNext(temp);
        newNode->setPrev(temp->getPrev());
        temp->getPrev()->setNext(newNode);
        temp->setPrev(newNode);
    }
}
```



- **Ques 3**

```
// Function to delete all occurrences of a specific value from the list
void deleteAllOccurrences(int item) {
    Node* temp = getHead();
    // Traverse the list and delete matching nodes
    while (temp != nullptr) {
        if (temp->getData() == item) {
            Node* nodeToDelete = temp;  // Store the node to delete

            // If the node is at the head
            if (temp == getHead()) {
                setHead(getHead()->getNext());
                if (getHead() != nullptr) getHead()->setPrev(nullptr);
            }
            // If the node is at the tail
            else if (temp == getTail()) {
                setTail(getTail()->getPrev());
                if (getTail() != nullptr) getTail()->setNext(nullptr);
            }
            // If the node is in the middle
            else {
                temp->getPrev()->setNext(temp->getNext());
                temp->getNext()->setPrev(temp->getPrev());
            }

            temp = temp->getNext();   // Move to the next node
            delete nodeToDelete;      // Delete the current node
        }
        else {
            temp = temp->getNext();   // Move to the next node if no match
        }
    }
}
```

```
// Function to delete the second last node of the list
void deleteSecondLast() {
    // Check if the list has less than 2 nodes
    if (getHead() == nullptr || getHead() == getTail()) {
        cout << "List is too short to delete the second last node." << endl;
        return;
    }

    Node* secondLast = getTail()->getPrev();  // Find the second last node

    // If the list has only two nodes, delete the head
    if (secondLast == getHead()) {
        setHead(getTail());  // Update head to point to the last node
        delete secondLast;  // Delete the second last node
    }
    // Otherwise, unlink and delete the second last node
    else {
        secondLast->getPrev()->setNext(getTail());
        getTail()->setPrev(secondLast->getPrev());
        delete secondLast;  // Delete the second last node
    }
}
```

- **Ques 2**

Singly with Circular

```
// Function to reverse the list
void reverseList() {
    if (getHead() == nullptr || getHead()->getNext() == getHead()) {
        cout << "Exception: List is too small or empty!" << endl;
        return;
    }

    Node* prev = nullptr;
    Node* current = getHead();
    Node* next = nullptr;

    Node* tail = getHead();

    do {
        next = current->getNext();
        current->setNext(prev);
        prev = current;
        current = next;
    } while (current != getHead());

    getHead()->setNext(prev);
    setHead(prev);
}
```

```
void deleteSecondLastNode() {
    if (!getHead() || getHead()->getNext() == getHead()) {
        cout << "Exception: List has less than 2 nodes." << endl;
        return;
    }

    Node* temp = getHead();
    Node* prev = nullptr;

    // Traverse the list to find the second last node
    do {
        prev = temp;
        temp = temp->getNext();
    } while (temp->getNext()->getNext() != getHead());

    if (prev == getHead() && getHead()->getNext() == getHead()) {
        setHead(temp);
        getHead()->setNext(getHead());
    }
    else {
        prev->setNext(temp->getNext());
    }

    delete temp;
}
```

```
// Function to append data to the list
void append(int data) {
    Node* newNode = new Node(data);
    if (getHead() == nullptr) {
        setHead(newNode);
        getHead()->setNext(getHead()); // Circular link
    }
    else {
        Node* temp = getHead();
        while (temp->getNext() != getHead()) {
            temp = temp->getNext();
        }
        temp->setNext(newNode);
        newNode->setNext(getHead()); // Maintain circular structure
    }
}
```

```cpp
void swapNodes(int val1, int val2) {
    if (val1 == val2) {
        cout << "The nodes have the same value, no need to swap." << endl;
        return;
    }

    Node* prev1 = nullptr, * curr1 = getHead();
    Node* prev2 = nullptr, * curr2 = getHead();

    // Find the first node with value val1
    while (curr1 && curr1->getData() != val1) {
        prev1 = curr1;
        curr1 = curr1->getNext();
    }

    // Find the second node with value val2
    while (curr2 && curr2->getData() != val2) {
        prev2 = curr2;
        curr2 = curr2->getNext();
    }

    if (curr1 == nullptr || curr2 == nullptr) {
        cout << "Exception: One or both nodes not found!" << endl;
        return;
    }

    if (prev1) prev1->setNext(curr2);
    else setHead(curr2);

    if (prev2) prev2->setNext(curr1);
    else setHead(curr1);
```

```cpp
    if (curr1 == nullptr || curr2 == nullptr) {
        cout << "Exception: One or both nodes not found!" << endl;
        return;
    }

    if (prev1) prev1->setNext(curr2);
    else setHead(curr2);

    if (prev2) prev2->setNext(curr1);
    else setHead(curr1);

    Node* temp = curr1->getNext();
    curr1->setNext(curr2->getNext());
    curr2->setNext(temp);
}
```

Ques 1

```
    // 4. Reverse the list
    void reverseList() {
        Node* prev = nullptr;
        Node* current = getHead();
        Node* next = nullptr;
        while (current) {
            next = current->getNext();
            current->setNext(prev);
            prev = current;
            current = next;
        }
        setHead(prev);
    }
```

```
// 3. Swap two nodes
void swapNodes(int x, int y) {
    if (x == y) return;

    Node* prevX = nullptr, * currX = getHead();
    while (currX && currX->getData() != x) {
        prevX = currX;
        currX = currX->getNext();
    }

    Node* prevY = nullptr, * currY = getHead();
    while (currY && currY->getData() != y) {
        prevY = currY;
        currY = currY->getNext();
    }

    if (!currX || !currY) {
        cout << "Exception: One or both nodes not found." << endl;
        return;
    }

    if (prevX) prevX->setNext(currY);
    else setHead(currY);

    if (prevY) prevY->setNext(currX);
    else setHead(currX);

    Node* temp = currX->getNext();
    currX->setNext(currY->getNext());
    currY->setNext(temp);
}
```

```cpp
// 1. Delete the second node
void deleteSecondNode() {
    if (!getHead() || !getHead()->getNext()) {
        cout << "Exception: List has less than 2 nodes." << endl;
        return;
    }
    Node* second = getHead()->getNext();
    getHead()->setNext(second->getNext());
    delete second;
}

// 2. Delete the second last node
void deleteSecondLastNode() {
    if (!getHead() || !getHead()->getNext()) {
        cout << "Exception: List has less than 2 nodes." << endl;
        return;
    }
    Node* temp = getHead();
    Node* prev = nullptr;
    while (temp->getNext() && temp->getNext()->getNext()) {
        prev = temp;
        temp = temp->getNext();
    }
    if (!prev) {
        // Only 2 nodes
        setHead(getHead()->getNext());
    }
    else {
        prev->setNext(temp->getNext());
    }
    delete temp;
}
```

Simple insertion and deletion of Singly with Circular

```
// Function to insert a new node at the end of the list
void insert(int data) {
    Node* newNode = new Node(data);

    // If the list is empty, insert the first node
    if (getHead() == nullptr) {
        setHead(newNode);
        newNode->setNext(getHead()); // Point to itself (circular)
    }
    else {
        Node* temp = getHead();
        // Traverse to the last node
        while (temp->getNext() != getHead()) {
            temp = temp->getNext();
        }
        temp->setNext(newNode);
        newNode->setNext(getHead()); // Maintain the circular nature
    }
}
```

```
// Function to delete a node with a specific value
void deleteNode(int value) {
    if (getHead() == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return;
    }

    Node* temp = getHead();
    Node* prev = nullptr;

    // Case: the head node needs to be deleted
    if (temp->getData() == value) {
        // If there's only one node in the list
        if (temp->getNext() == getHead()) {
            delete temp;
            setHead(nullptr);
            return;
        }

        // Otherwise, find the last node
        while (temp->getNext() != getHead()) {
            temp = temp->getNext();
        }
        Node* toDelete = getHead();
        setHead(getHead()->getNext());
        temp->setNext(getHead());
        delete toDelete;
        return;
    }
```

```
        return;
    }

    // Case: deleting a node other than the head
    do {
        prev = temp;
        temp = temp->getNext();
    } while (temp != getHead() && temp->getData() != value);

    // If the node was found
    if (temp->getData() == value) {
        prev->setNext(temp->getNext());
        delete temp;
    }
    else {
        cout << "Node with value " << value << " not found." << endl;
    }
}
```

Simple insertion and deletion of Doubly with Circular

```
// Function to insert a new node at the end of the list
void insert(int data) {
    Node* newNode = new Node(data);

    // If the list is empty, insert the first node
    if (getHead() == nullptr) {
        setHead(newNode);
        newNode->setNext(newNode);  // Point next to itself (circular)
        newNode->setPrev(newNode);  // Point prev to itself (circular)
    }
    else {
        Node* tail = getHead()->getPrev();  // Get the last node (tail)
        tail->setNext(newNode);  // Last node's next points to new node
        newNode->setPrev(tail);  // New node's prev points to the last node
        newNode->setNext(getHead());  // New node's next points to head
        getHead()->setPrev(newNode);  // Head's prev points to the new node
    }
}
```

```
void deleteNode(int value) {
    if (getHead() == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return;
    }

    Node* temp = getHead();

    // Case: the node to delete is the head
    if (temp->getData() == value) {
        // If there's only one node in the list
        if (temp->getNext() == getHead()) {
            delete temp;
            setHead(nullptr);
            return;
        }

        Node* tail = temp->getPrev();  // Get the last node (tail)
        setHead(temp->getNext());  // Update head to the next node
        getHead()->setPrev(tail);  // Update the new head's prev pointer
        tail->setNext(getHead());  // Last node's next points to the new head
        delete temp;  // Delete the old head
        return;
    }

    // Case: deleting a node other than the head
    do {
        if (temp->getData() == value) {
            Node* prevNode = temp->getPrev();
            Node* nextNode = temp->getNext();
            prevNode->setNext(nextNode);  // Bypass the node to delete
            nextNode->setPrev(prevNode);  // Set the next node's prev pointer
            delete temp;  // Delete the node
            return;
        }
        temp = temp->getNext();
    } while (temp != getHead());
```

No issues found

```
        temp = temp->getNext();
    } while (temp != getHead());

    cout << "Node with value " << value << " not found." << endl;
}
```