SORTING ON LINKED LIST

BUBBLE SORT

```
#include <iostream>
Using namespace std;
class Node {
public:
   int data;
   Node* next;

   Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
private:
   Node* head;

public:
   LinkedList() : head(nullptr) {}

   // Insert a new node at the front of the list
   void insert(int val) {
      Node* newNode = new Node(val);
      newNode->next = head;
      head = newNode;
   }
```

```
// Bubble sort function for the linked list
void bubbleSort() {
    if (!head) return;

    bool swapped;
    do {
        swapped = false;
        Node* current = head;

        while (current && current->next) {
            if (current->data > current->next->data) {
                // Swap data directly
                int temp = current->data;
                current->data = current->next->data;
                current->next->data = temp;
                swapped = true;
            }
            current = current->next;
        }
    } while (swapped);
}

// Function to print the linked list
void printList() const {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
```

```
            current = current->next;
        }
        cout <<  endl;
    }
};

int main() {
    LinkedList list;
    list.insert(5);
    list.insert(1);
    list.insert(4);
    list.insert(2);
    list.insert(3);

     cout << "Before sorting: ";
    list.printList();

    list.bubbleSort();

     cout << "After sorting: ";
    list.printList();

    return 0;
}
```

OUTPUT

```
Before sorting: 3 2 4 1 5
After sorting: 1 2 3 4 5
```

SELECTION SORT

```
#include <iostream>
Using namespace std;
class Node {
public:
   int data;
   Node* next;

   Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
public:
   Node* head;

   LinkedList() : head(nullptr) {}

   // Function to insert a new node at the end
   void insert(int data) {
      Node* newNode = new Node(data);
      if (!head) {
         head = newNode;
      } else {
         Node* temp = head;
         while (temp->next) {
            temp = temp->next;
```

```
        }
        temp->next = newNode;
    }
}

// Function to print the list
void printList() {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout <<  endl;
}

// Function to perform selection sort
void selectionSort() {
    if (head == NULL || head->next == NULL) return;

    Node* current = head;
    while (current) {
        Node* minNode = current;
        Node* temp = current->next;

        // Find the node with the minimum value in the unsorted part of the list
        while (temp) {
            if (temp->data < minNode->data) {
```

```
                minNode = temp;
            }
            temp = temp->next;
        }

        // Instead of swapping nodes, swap their values
        if (minNode != current) {
            int tempData = current->data;
            current->data = minNode->data;
            minNode->data = tempData;
        }

        // Move to the next node in the list
        current = current->next;
    }
  }
};

int main() {
  LinkedList list;

  list.insert(30);
  list.insert(3);
  list.insert(4);
  list.insert(20);
  list.insert(5);
```

```
    cout << "Original list: ";
    list.printList();

    list.selectionSort();

    cout << "Sorted list: ";
    list.printList();

    return 0;
}
```

OUTPUT

```
Original list: 30 3 4 20 5
Sorted list: 3 4 5 20 30
```

INSERTION SORT

```
#include <iostream>
using namespace std:
class Node {
public:
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
public:
```

```
Node* head;

LinkedList() : head(nullptr) {}

// Function to insert a new node at the end
void insert(int data) {
   Node* newNode = new Node(data);
   if (!head) {
      head = newNode;
   } else {
      Node* temp = head;
      while (temp->next) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}

// Function to print the list
void printList() {
   Node* temp = head;
   while (temp) {
       cout << temp->data << " ";
      temp = temp->next;
   }
    cout <<  endl;
}
```

```
// Function to perform insertion sort
void insertionSort() {
    if (head == NULL || head->next == NULL) return; // If the list is empty or has only one element, it's already sorted.

    // Initialize the sorted part of the list with the first node.
    Node* sorted = nullptr;
    Node* current = head;

    while (current) {
        Node* nextNode = current->next;  // Save the next node

        // If the sorted list is empty or the current node should be placed at the beginning
        if (sorted == NULL || current->data <= sorted->data) {
            current->next = sorted;  // Insert the current node at the start of sorted
            sorted = current;
        } else {
            // Find the correct position to insert current in the sorted part
            Node* temp = sorted;
            while (temp->next && temp->next->data < current->data) {
                temp = temp->next;
            }

            // Insert the current node after temp
            current->next = temp->next;
            temp->next = current;
```

```
        }

        // Move to the next node in the original list
        current = nextNode;
      }

      // Update the head to point to the sorted list
      head = sorted;
   }
};

int main() {
   LinkedList list;

   list.insert(30);
   list.insert(100);
   list.insert(4);
   list.insert(20);
   list.insert(5);

    cout << "Original list: ";
   list.printList();

   list.insertionSort();

    cout << "Sorted list: ";
   list.printList();
```

```
    return 0;
}
```

OUTPUT

```
Original list: 30 100 4 20 5
Sorted list: 4 5 20 30 100
```

RADIX SORT

```
#include <iostream>


class Node {
public:
    int data;
    Node* next;


    Node(int val) : data(val), next(nullptr) {}
};


class LinkedList {
public:
    Node* head;


    LinkedList() : head(nullptr) {}


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

```
    if (!head) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to print the list
void printList() {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout <<  endl;
}

// Function to perform Radix Sort
void radixSort() {
    if (!head || !head->next) return; // If the list is empty or has only one element, it's already sorted.

    int maxVal = getMaxValue();  // Get the maximum value to know how many digits we need to sort
```

```
    // Perform counting sort for each digit (ones, tens, hundreds, etc.)
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        head = countingSortByDigit(exp);
    }
  }

private:
  // Function to get the maximum value in the list
  int getMaxValue() {
    Node* temp = head;
    int maxVal = head->data;
    while (temp) {
        if (temp->data > maxVal) {
            maxVal = temp->data;
        }
        temp = temp->next;
    }
    return maxVal;
  }

  // Perform counting sort based on the digit represented by exp (1 for ones, 10 for tens, etc.)
  Node* countingSortByDigit(int exp) {
    Node* buckets[10] = {nullptr};  // Array of heads of linked lists for digits 0-9
    Node* bucketTails[10] = {nullptr};  // Array to store the tails of linked lists for digits 0-9

    Node* current = head;
```

```
// Place nodes in corresponding buckets based on the current digit
while (current) {
    int digit = (current->data / exp) % 10;  // Extract the digit

    if (!buckets[digit]) {
        // First node for this digit
        buckets[digit] = current;
        bucketTails[digit] = current;
    } else {
        // Append the node to the corresponding bucket
        bucketTails[digit]->next = current;
        bucketTails[digit] = current;
    }

    current = current->next;
}

// Rebuild the list by concatenating all the buckets in order
Node* newHead = nullptr;
Node* newTail = nullptr;

for (int i = 0; i < 10; ++i) {
    if (buckets[i]) {
        if (!newHead) {
            newHead = buckets[i];
            newTail = bucketTails[i];
```

```
            } else {
                newTail->next = buckets[i];
                newTail = bucketTails[i];
            }
        }
    }

    // Terminate the list
    if (newTail) {
        newTail->next = nullptr;
    }

    return newHead;
  }
};

int main() {
  LinkedList list;

  list.insert(170);
  list.insert(45);
  list.insert(75);
  list.insert(90);
  list.insert(802);
  list.insert(24);
  list.insert(2);
  list.insert(66);
```

```
    cout << "Original list: ";
    list.printList();

    list.radixSort();

    cout << "Sorted list: ";
    list.printList();

    return 0;
}
```

OUTPUT

```
Original list: 170 45 75 90 802 24 2 66
Sorted list: 2 24 45 66 75 90 170 802
```