Linked List Operations

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}

temp->next = newNode;
```

3. Insert at the Middle

- Allocate memory and store data for new node.

- Traverse to node just before the required position of new node.
- Change next pointers to include new node in between.

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;

struct node *temp = head;

for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

- Point head to the second node.

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

2. Delete from end

- Traverse to second last element.
- Change its next pointer to null.

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle

- Traverse to element before the element to be deleted.
- Change next pointers to exclude the node from the chain.

```
for(int i=2; i< position; i++) {
  if(temp->next!=NULL) {
    temp = temp->next;
  }
}

temp->next = temp->next->next;
```

Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding `item` on a linked list.

- Make `head` as the `current` node.
- Run a loop until the `current` node is `NULL` because the last element points to `NULL`.
- In each iteration, check if the key of the node is equal to `item`. If it the key matches the item, return `true` otherwise return `false`.

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
    if (current->data == key) return true;
      current = current->next;
  }
  return false;
}
```

Sort Elements of a Linked List

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the `head` as the `current` node and create another node `index` for later use.
2. If `head` is null, return.
3. Else, run a loop till the last node (i.e. `NULL`).
4. In each iteration, follow the following step 5-6.
5. Store the next node of `current` in `index`.
6. Check if the data of the current node is greater than the next node. If it is greater, swap `current` and `index`.

Check the article on bubble sort for better understanding of its working.

```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
    return;
  } else {
    while (current != NULL) {
      // index points to the node next to current
      index = current->next;

    while (index != NULL) {
        if (current->data > index->data) {
          temp = current->data;
          current->data = index->data;
          index->data = temp;
        }
        index = index->next;
    }
    current = current->next;
    }
  }
}
```