Shazaib Dawood
10/28/2024
CS 211

- concept of a linked list in comparison to an array
    - Array
        - fixed size
        - continuous memory block
        - accessed by index
    - linked list
        - dynamic size
        - memory not stored in a large block
        - elements accessed by pointer
- pros and cons of linked lists in comparison to arrays
    - pros
        - can change size of linked list
        - O(1) time complexity for inserting/removing an element
    - cons
        - extra memory for pointers
- most important aspects about working with linked lists in C vs. other languages
    - C
        - Pointers are manually managed
    - other languages
        - have built in libraries and garbage collection to free memory
- defining a node struct; declaring a linked list as a node pointer

```
struct Node {
   int data;
   struct Node* next;
};
struct Node* head = NULL;
```

- common linked list functions - brief explanation & example(s)
    - traversal (e.g. to display all elements, to perform a global calculation/operation such as a sum or maximum, searching for specific elements, etc.)
        - accesses each element of the list

```
void printList(struct Node* head) {
   struct Node* current = head;
   while (current != NULL) {
      printf("%d, ", current->data);
      current = current->next;
   }
   printf("NULL\n");
```

```
}
```

- prepend (i.e. insert to front)
    - inserts new node at the beginning of the lsit

```
void prepend(struct Node** head, int data) {
   struct Node* newNode = malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = *head;
   *head = newNode;
}
```

- append (i.e. insert to back)
    - adds new node at the end of the list

```
void append(struct Node** head, int data) {
   struct Node* newNode = malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;

   if (*head == NULL) {
      *head = newode; //empty list
      return;
   }

   struct Node* last = *head;
   while (last->next != NULL) {
      last = last->next;
   }
   last->next = newNode;
}
```

- insert at specified index

```
void insertAtIndex(struct Node** head, int index, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;

   if (index == 0) {
      newNode->next = *head;
      *head = new_node;
      return;
   }

   struct Node* current = *head;
   for (int i = 0; i < index - 1 && current != NULL; i++) {
```

```c
        current = current->next;
    }

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

- delete from front
    - removes the head node

```c
void deleteFromFront(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

- delete from back
    - removes the tail node

```c
void deleteFromBack(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }

    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }

    struct Node* second_last = *head;
    while (second_last->next->next != NULL) {
        second_last = second_last->next;
    }

    free(second_last->next);
    second_last->next = NULL;
}
```

        - delete specified index
            - removes node at specific index

```
void deleteAtIndex(struct Node** head, int index) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = *head;

    if (index == 0) {
        *head = temp->next;
        free(temp);
        return;
    }

    for (int i = 0; i < index - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        printf("Index out of bounds\n");
        return;
    }

    struct Node* next = temp->next->next;
    free(temp->next);
    temp->next = next;
}
```

- potential extensions of the classic singly-linked list (e.g. doubly-linked list, circularly linked list, etc.) and when they would be useful
    - Doubly linked list
        - each node points to previous and next element
        - can traverse backwards
    - Circularly linked list
        - last node points to the first node