Linked Lists

Prons and Cons

- Common property for all array-like random structures: consecutive storing of elements
    - Advantage
        - Quick random access `a[3]`
            - Compiler knows: location of array `a` + type of element
                - e.g. if type is double --> 8 bytes, then compiler just does `1000 + 8*k` for `a[k]`
            - trivial calculation
    - Disadvantage
        - Costly insertion/removal for ordered elements
            - Exception: insertion/removal at the end
                - delete/add one, change size -> cheap operation
                - do this when your items do not have to be ordered
            - If you want to add a new item between consecutive elements -> shift all elements to make room for insertion/removal
            - Insertion/removal near the beginning much more costly than end

Singly-Linked List

Linked List Class Implementation

```
struct Node
{
    string value;
    Node *next;
}
```

The only member variable we need is a head pointer

```
class LinkedList
{
    public:
        LinkedList();
        void addToFront(string v);
```

```
        void addToRear(string v);
        void deleteItem(string v);
        bool findItem(string v);
        void printItems();
        ~LinkedList();
    private:
        Node *head;
};
```

Constructor: create an empty list (head pointer to nullptr)

```
LinkedList()
{
    head = nullptr;
}
```

Print Items in a List: loop through each of the nodes and print out values, starting with node pointed to by `head`

Linked List Traversal

```
void printItems()
{
    //Use a node pointer
    Node *p;
    p = head; //p points to 1st node

    while (p != nullptr)
    {
        cout << p->value << endl;
        p = p->next;
    }
}
```

Add an Item to a List

Add at top

```
void addToFront(string v)
{
    //allocate new node
    Node *p;
    p = new Node;

    p->value = v; //set value

    //Link new node to current top node
```

```
    p->next = head;

    //Update head pointer to new top node
    head = p;
}
```

Add at rear

- Two cases:
    - Empty list: same as adding new node to front
    - Non-empty list: traverse down links until we find the current last node
        - Use a `temp` variable to traverse to current last node
        - Allocate a new node, set value in node
        - Link the current last node to new node
        - Link last node to nullptr

```
void addToRear(string v)
{
    if (head == nullptr)
    {
        addToFront(v);
    }
    else
    {
        Node *p;
        p = head; //start at top node
        while (p->next != nullptr)
        {
            p = p->next;
        }
        Node *n = new Node;
        n->value = v;
        p->next = n;
        n->next = nullptr;
    }
}
```

Add Anywhere

- Several Cases:
    - If empty list: `addToFront()`
    - If new node belongs at top of list (sorted etc.): `addToFront()`
    - If new node belongs in middle of list

        - Use a traversal loop to find the node just ABOVE where you want to insert your new item
        - Allocate and fill new node
        - Link new node into list right after the ABOVE node

```
void AddItem(string newItem)
{
    if (head == nullptr)
    {
        AddToFront(newItem);
    }
    else if (/*decide if new item belongs at the top*/)
    {
        AddToFront(newItem);
    }
    else //new node belongs somewhere in the middle
    {
        Node *p = head;
        while (p->next != nullptr)
        {
            if (/*p points just above where to insert*/)
            {
                break;
            }
            p = p->next;
        }
        Node *latest = new Node;
        latest->value = newItem;
        latest->next = p->next;
        p->next = latest;
    }
}
```

Delete Item from the List

- Two cases:
    - Check if list is empty first -> if then return
    - Deleting the first node
        - If value is value of first node
        - Set node to delete = address of top node
        - Update head to point to the second node in list
        - Delete target node
        - Return
    - Deleting interior or last node

        - Traverse down the list until find node ABOVE the one to delete (so we can relink)
            - If p->next is not a nulltpr and is p->next->value is the value we want
        - If found target node
            - killMe = addr of target node
            - Link node above the node below
            - Delete target node

```
void deleteItem(string v)
{
    if (head == nullptr) { return; }

    if (head->value == v)
    {
        Node *killMe = head;
        head = killMe->next;
        delete killMe;
        return
    }

    Node*p = head;
    while (p != nullptr)
    {
        if (p->next != nullptr && p->next->value == v)
        {
            break; //p points to node above
        }
        p = p->next;
    }
    if (p != nullptr) //found our value
    {
        Node *killMe = p->next;
        p->next = killMe->next;
        delete killMe;
    }
}
```

Linked List Destruction

- Traverse the list with temp variable `p`
- Before we delete the node pointed to buy p
    - Save the location of the next node in a temp variable

```
~LinkedList()
{
```

```
    Node *p;
    p = head;
    while (p != nullptr)
    {
        Node *n = p->next;
        delete p;
        p = n;
    }
}
```

Head and Tail Pointers

Disadvantages of Linked Lists (Singly)

- Complex to implement compared to arrays
- Element accessing
  - To access the kth item, have to travese down k - 1 times from the head
- To add an item at the end -> traverse through all N existing nodes

Tail Pointers and Linked Lists

> A tail pointer is a pointer that always points to the last node of the list

- We can now add new items to the end of our list without traversing

```
class LinkedList
{
    public:
        LinkedList();
        void addToFront(string v);
        ...
    private:
        Node *head;
        Node *tail;
};
```

New `addToRear()` function

- No longer need traversal loop, tail pointer already points to last node

```
void addToRear(string v)
{
    if (head == nullptr)
    {
        addToFront(v);
    }
    else
```

```
    {
        Node *n = new Node;
        n->value = v;
        tail->next = n; //linked prev last node to curr
        n->next = nullptr;
        tail = n;
    }
}
```

Doubly-Linked List

A doubly-linked list has both `next` and `previous` pointers in every node

```
struct Node
{
    string value;
    Node* next;
    Node* prev;
};
```

- Everytime we insert a new node or delete an existing node -> update 3 sets of pointers
  - The new nodes's next and prev pointers
  - The previous node's next pointer
  - The following node's previous pointer

Circular-Linked List with a Dummy Node