Discussion #2

data structures,linked lists

Arrays

- fixed size: resizing is expensive
- insertion and deletion are inefficient: elements are usually shifted
- random access, i.e., efficient indexing
- no memory waste if array is full or almost
- sequential access is faster

Linked Lists

- dynamic size

Common Operations for Data Structures

- insertToFront(DataType d)
- insertToEnd(DataType d)
- insertAfter(DataType d, DataType toInsert)
- delete(DataType d)
- print()
- reverse()

```
// LinkedList.h

#ifndef LINKED_LIST_H
#define LINKED_LIST_H

class LinkedList {

public:

  LinkedList();
  ~LinkedList();
  bool insertToFront(int v);
  bool insertToEnd(int v);
  bool insertAfter(int find, int v);
  bool erase(int v);
  void print();
  void reverse();
  }

private:
  struct Node {
    int value;
    Node* next;
    Node(int x) {
      value = x;
      next = nullptr;
    }
  }

  Node *m_head;
  Node *m_tail;
  int m_size;
};
```

```
// LinkedList.cpp

#include "LinkedList.h"

LinkedList::LinkedList() {
    m_head = nullptr;
    m_tail = nullptr;
    m_size = 0;
}

LinkedList::~LinkedList() {
    Node* iterator = m_head;
    while (iterator != nullptr) {
        Node* temp = iterator->next;
        delete iterator;
        iterator = temp;
    }
}

bool LinkedList::insertToFront(int v) {
    if (m_head == nullptr) {    // first node
        m_head = new Node(v);

        m_tail = m_head;
        return true;
    } else {                  // not first node
        Node* toAdd = new Node(v);
        toAdd->next = m_head;
        m_head = toAdd;
        return true;
    }

    m_size++;
}

bool LinkedList::insertToEnd(int v) {
    if (m_tail == nullptr) {
        insertToFront(v);
    } else {
        Node* toAdd = new Node(v);
        m_tail->next = toAdd;
        m_tail = toAdd;
        m_size++;
    }
    return true;
}

```

```cpp
bool LinkedList::insertAfter(int find, int v) {
    Node* iterator = m_head;
    while (iterator != nullptr) {
        if (iterator->value == find)
            break;
        iterator = iterator->next;
    }
    if (iterator != nullptr) {
        Node* toAdd = new Node(v);
        toAdd->next = iterator->next;
        iterator->next = toAdd;
        m_size++;
        if (toAdd->next == nullptr)
            tail = toAdd;
        return true;
    }
    return false;
}

bool LinkedList::erase(int v) {
    if (head == nullptr)
        return false;

    Node* iterator = head;
    if (head->value == v) {
        Node* temp = head;
        head = head->next;
        if (head == NULL)
            tail = NULL;
        delete temp;
        size--;
        return true;
    }

    Node* iterator = head;
    while (iterator-> next != nullptr) {
        if (iterator->next->value == v)
            break;
        iterator = iterator->next;
    }

    if (iterator ->next != nullptr) {
        Node* toDel = iterator->next;
        iterator->next = iterator->next->next;
        if (iterator->next == nullptr)
            tail = iterator;
        delete toDel;
    }
}

void LinkedList::reverse() {
    Node* iterator = m_head;
    Node* prev = nullptr;

    while (iterator != nullptr) {
      Node* tempNext = iterator->next;
      iterator->next = prev;
```

```
        prev = iterator;
        iterator = tempNext;
    }

    Node* temp = m_tail;
    m_tail = m_head;
    m_head = temp;
}

void LinkedList::print() {
    Node* iterator = m_head;

    while (iterator != nullptr) {
        std::cout << iterator->value << "  ";
        iterator = iterator->next;
    }
}
```

```
// main.cpp

int main() {
  LinkedList ll;
  ll.print();
  ll.insertToFront(1);
  ll.print();
  ll.insertToFront(2);
  ll.print();
}
```