## **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()

```
1 // LinkedList.h
 2
 3 #ifndef LINKED_LIST_H
 4 #define LINKED_LIST_H
 5
 6 class LinkedList {
 7
 8 public:
 9
10
     LinkedList();
11
     ~LinkedList();
     bool insertToFront(int v);
12
13
     bool insertToEnd(int v);
     bool insertAfter(int find, int v);
14
15
     bool erase(int v);
     void print();
16
17
     void reverse();
     }
18
19
20 private:
21
     struct Node {
22
     int value;
      Node* next;
23
24
      Node(int x) {
```

```
25
         value = x;
26
         next = nullptr;
27
28
     }
29
30
     Node *m_head;
31
     Node *m_tail;
     int m_size;
32
33 };
1 // LinkedList.cpp
 3 #include "LinkedList.h"
 4
 5 LinkedList::LinkedList() {
       m_head = nullptr;
6
 7
       m_tail = nullptr;
8
       m_size = 0;
9 }
10
11 LinkedList::~LinkedList() {
12
       Node* iterator = m_head;
13
       while (iterator != nullptr) {
14
           Node* temp = iterator->next;
15
           delete iterator;
16
           iterator = temp;
17
18 }
19
20 bool LinkedList::insertToFront(int v) {
       if (m head == nullptr) {
                                    // first node
21
22
           m_head = new Node(v);
23
24
           m_tail = m_head;
25
           return true;
                                  // not first node
26
       } else {
27
           Node* toAdd = new Node(v);
28
           toAdd->next = m_head;
29
           m_head = toAdd;
30
           return true;
31
32
33
       m_size++;
34 }
35
36 bool LinkedList::insertToEnd(int v) {
       if (m_tail == nullptr) {
37
38
           insertToFront(v);
39
       } else {
40
           Node* toAdd = new Node(v);
41
           m_tail->next = toAdd;
42
           m_tail = toAdd;
43
           m_size++;
44
45
       return true;
46 }
47
```

```
48 bool LinkedList::insertAfter(int find, int v) {
 49
        Node* iterator = m_head;
        while (iterator != nullptr) {
 50
 51
            if (iterator->value == find)
 52
                break;
            iterator = iterator->next;
 53
 54
        }
        if (iterator != nullptr) {
 55
            Node* toAdd = new Node(v);
 56
            toAdd->next = iterator->next;
 57
 58
            iterator->next = toAdd;
 59
            m_size++;
 60
            if (toAdd->next == nullptr)
 61
                tail = toAdd;
 62
            return true;
        }
 63
 64
        return false;
 65 }
 66
 67 bool LinkedList::erase(int v) {
 68
        if (head == nullptr)
 69
            return false;
 70
        Node* iterator = head;
 71
 72
        if (head->value == v) {
 73
            Node* temp = head;
 74
            head = head->next;
 75
            if (head == NULL)
 76
                tail = NULL;
 77
            delete temp;
 78
            size--;
 79
            return true;
 80
        }
 81
 82
        Node* iterator = head;
        while (iterator-> next != nullptr) {
 83
 84
            if (iterator->next->value == v)
 85
                break;
            iterator = iterator->next;
 86
        }
 87
 88
 89
        if (iterator ->next != nullptr) {
 90
            Node* toDel = iterator->next;
 91
            iterator->next = iterator->next->next;
 92
            if (iterator->next == nullptr)
 93
                tail = iterator;
 94
            delete toDel;
 95
 96 }
 97
 98 void LinkedList::reverse() {
        Node* iterator = m_head;
 99
100
        Node* prev = nullptr;
101
        while (iterator != nullptr) {
102
103
          Node* tempNext = iterator->next;
104
          iterator->next = prev;
```

```
105
       prev = iterator;
106
        iterator = tempNext;
107
        }
108
109
        Node* temp = m_tail;
        m_tail = m_head;
110
        m_head = temp;
111
112 }
113
114 void LinkedList::print() {
115
        Node* iterator = m_head;
116
        while (iterator != nullptr) {
117
            std::cout << iterator->value << " ";</pre>
118
119
            iterator = iterator->next;
120
        }
121 }
  1 // main.cpp
  2
  3 int main() {
  4
      LinkedList ll;
  5
      ll.print();
  6
      ll.insertToFront(1);
  7
      ll.print();
  8
      ll.insertToFront(2);
  9
      ll.print();
 10 }
```