Lecture 6

Pointer, linked node, linked list

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 18th, 2024

IDA, Linköping University

6.1

Content

Contents

1 C++ Memory **1**

2 Pointer and linked node **2**
2.1 Introduction 2
2.2 Pointers 2
2.3 Object 4
2.4 Linked nodes 7

3 Linked list **9**

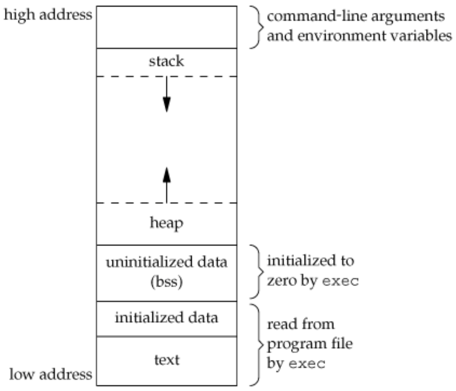
6.2

1 C++ Memory

C++ Memory

Four memory areas:

- Program code segment (text)
- Data segment initialised (constants, `"Hello␣world"`) and uninitialised (global variables)
- Heap (dynamically allocable memory **new** `std::string()`, `malloc(10)`...)
- Stack (local variables **int** `a`, function arguments...)



6.3

Stack vs heap

- Stack
  - holds return address of function call, arguments, local variables
  - fast (LIFO) but limited (one block allocated for each thread)
  - memory is free when function is returned
- Heap
  - dynamically allocated

- available space grows with the program
- require space management
- useful if size is unknown at compilation and for long term structure
- **dynamic memory needs to be freed!**

6.4

2 Pointer and linked node

2.1 Introduction

Arraybased data structure

- Many containers use arrays of elements for storage

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-3	17	9	0	0	0	0	0	0

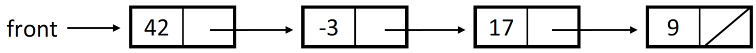
  - What are the benefits/drawbacks of using arrays?
    * Benefits: fast to add/remove at the end; fast to access elements
    * Drawbacks: slow to add/remove from the middle; array is wasting memory; need to increase capacity when full

6.5

2.2 Pointers

Linked data structure

- Other containers use **linked node object** to store data
  - Each node object stores a data element and a link to another object



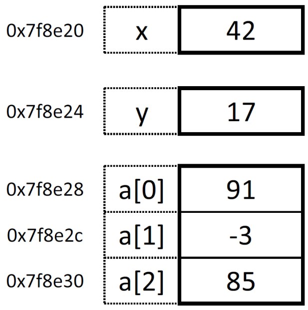
    * Benefits: quick to add/delete in all position
    * Drawbacks: slower access to certain part of the data structure
    * To understand link structure, we need to understand pointers. . .

6.6

Memory Address

- When a variable is declared, it is stored somewhere in memory
  - We can get the memory address with the `&`-operator
    * Memory address are usually written in hexadecimal form (base-16)
    * Many common data types use 4 bytes (32 bits) of memory

```
int x = 42;
int y = 17;
int a[3] = {91, -3, 85};
cout << x << endl;      // 42
cout << &x << endl;     // 0x7f8e20
cout << y << endl;      // 17
cout << &y << endl;     // 0x7f8e24
cout << &a[0] << endl;  // 0x7f8e28
cout << &a[1] << endl;  // 0x7f8e2c
cout << &a[2] << endl;  // 0x7f8e30
```

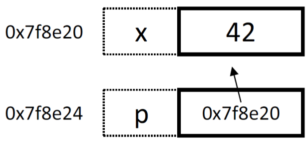


6.7

Pointer

- **pointer**: One variable that stores a memory address
  - Pointers are declared with * after the type
  - We can refer to the value at the address of the pointer using *-operator (also called dereferencing)

```
int x = 42;
int* p = &x;
cout << p << endl;  // 0x7f8e20
cout << *p << endl; // 42
*p = 99;
cout << x << endl;  // 99
```

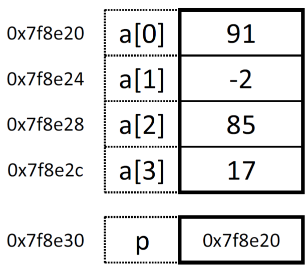


6.8

Pointer arithmetic

- We can modify a pointer with +, -, ++, --, etc.
  - Incrementing T* by 1 move the pointer to the next object of type T, in effect incrementing the address by the size of T
  - We can use the syntax [k] for random access. k is put after the pointer:
    - (An array variable is actually just a pointer pointing to the first element)

```
int a[4] = {91, -2, 85, 17};
int* p = a;              // p = &a[0];
p[1] = 5;                // a[1] = 5;
p++;                     // p = &a[1];
cout << *p << endl;      // 5
*(p + 2) = 26;           // a[3] = 26;
cout << p[2] << endl;    // 26
```

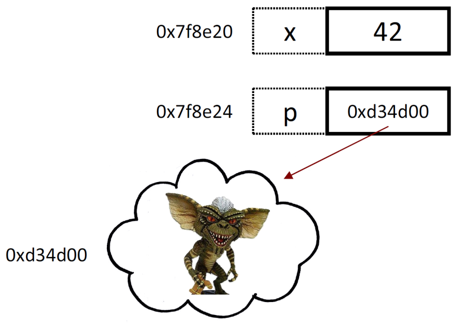


6.9

Dangling pointer

- If a pointer is pointing at arbitrary location in memory.
  - If we access the value of such a pointer, the program will likely crash!

```
int x = 42;
int* p;                  // dangling pointer
cout << p << endl;       // 0xd34d00
cout << *p << endl;      // KABOOM

int a[3] = {91, -2, 85};
int* p2 = a;
cout << p2[5] << endl;   // BOOM
```

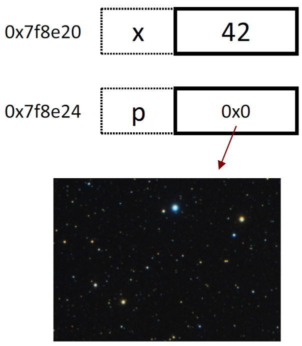


6.10

nullptr

- nullptr: Pointer literal which means “points to nothing” (points to address 0x0)
  - Meant to be used as a blank value to initialise the pointer
  - Dereferencing nullptr leads to an immediate crash!

```
int x = 42;
int* p = nullptr;
cout << p << endl;     // 0
cout << *p << endl;    // KABOOM

// Test for validity
if (p == nullptr) {...} // true
```



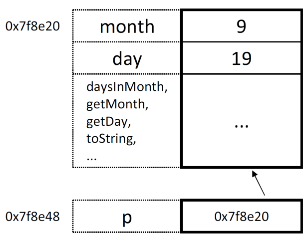
6.11

2.3 Object

Pointer to objects

- We can point to objects
  - To access a member of the object, write: pointer->member
    - Equivalent to (*pointer).member

```
Date d(9, 19);
Date* p = &d;
cout << p->daysInMonth() << endl; // 30
p->nextDay();
cout << *p << endl; // 20/9
```



6.12

Objects lifetime

- Items declared as variables live until the end of the block (end of function, end of if-statement)
  - This is called *static allocation* or *stack allocation*
  - If we return the value, the object is copied (or moved) and the original destroyed
    - How can we create an object that outlive a function?

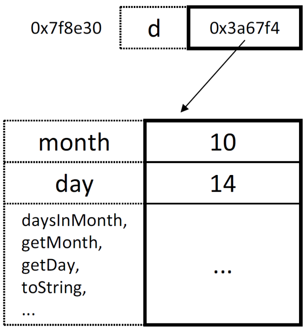
```
void foo() {
    int x = 42;
    Date d1(9, 19); ...
} // x, d1 förstörs

int main() {
    int a = 17;
    foo();
    Date d2(10, 14); ...
    return 0;
} // a, d2 förstörs
```

6.13

Dynamic allocation

- Operator **new** allocate "long term memory" for an object
  - Called *dynamic allocation* or *allocation on the heap*
  - Objects are kept alive even after the end of the function scope
    - The memory allocated for objects in the heap must be freed explicitly **delete**

```
void foo() {
    Date* d = new Date(10, 14);
    cout << d->daysInMonth() << endl; // 31
    d->nextDay();
    cout << d << endl;  // 0x3a67f4
    cout << *d << endl; // 15/10
    ...
    delete d;
}
```



6.14

const pointers

- What is the difference between **const** T*, T **const** * and T * **const**?
  - **const** T* and T **const** * are the same, it is a pointer to a constant value.

```
int a = 2;
const int* p = &a;
*p = 3; // Does not Compile!
++p; // Compile but dangerous!
```

  - T * **const** is a constant pointer to a non constant value.

```
int a = 2;
int* const p = &a;
*p = 3; // Compile!
++p; // Does not compile (and bad idea anyway)!
```

  - **const** T* **const** is a constant pointer to a constant value.

```
int a = 2;
const int* const p = &a;
*p = 3; // Does not Compile!
++p; // Does not compile (and bad idea anyway)!
```

6.15

references vs pointers

- What is the difference between a pointer (`T*`) and a reference (`T&`)?
- A pointer is a variable that contains a memory address
  - A pointer can (generally) be reassigned
  - A pointer supports arithmetic operation
  - To access the value, pointers need to be dereferenced (ie `*p` or `p->`)
- Reference are more aliases to a variable
  - Reference are bound to a specific variable at initialisation
  - Reference behaves like variable

```
int a = 2;
int&r = a;
int*p = &r;

// Prints 0x7ffc3c776de4 0x7ffc3c776de4 0x7ffc3c776de8 0x7ffc3c776de4
std::cout << &a << "␣" << &r << "␣" << &p << "␣" << &*p << std::endl;
```

6.16

references vs pointers

- References can be bound to temporaries

```
const int &x = 12; //legal C++
int *y = &12; //illegal to dereference a temporary.
```

- This makes **const**& safer for use in argument lists.

```
void f(std::vector<int>* v) {}
void g(const std::vector<int>& v) {}
void h(std::vector<int>& v) {}
void i(std::vector<int> v) {}
void j(std::vector<int>&& v) {}

f(&(std::vector<int>{10, 12, 13})); // Error: taking address of temporary
g(std::vector<int>{10, 12, 13}); // Works!
h(std::vector<int>{10, 12, 13}); // Error: invalid initialisation of
                                 // non-const reference on temporary
i(std::vector<int>{10, 12, 13}); // Works! But less efficient than g
j(std::vector<int>{10, 12, 13}); // Works! As performant as g,
                                 // but wrong meaning!
```

6.17

references vs pointers

- References may or may not have a memory location of their own!

```
int a = 2;
int&r = a;
int*p = &a;

// Print 4 4
std::cout << ((long long int)&p - (long long int)&a)
          << "␣" << sizeof(int) << std::endl;
```

- References are similar to `T* const` but may or may not be implemented as such, the standard does not specify!
- In many ways, references are similar to pointers. So why C++ have them?
- Mostly for convenience

6.18

references vs pointers

- Originally introduced for operator overloading. Imagine operators without references. You could use pointers to avoid copying:

```
MyClass add(MyClass*, MyClass* );
MyClass add(MyClass*, int );

MyClass a, b;
int c;

add(&a, MyClass()); // Does not Work
add(&a, &b); // Works but ugly
add(&a, c); // More confusion
```

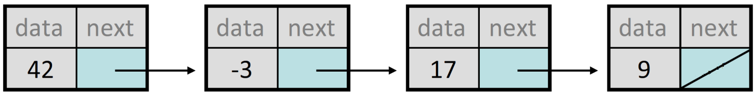
6.19

2.4 Linked nodes

Data structure for linked nodes

```
template<typename T>
struct ListNode {
    T data;
    ListNode<T>* next;
};
```

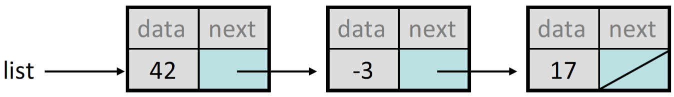
- Each object stores:
  - a value of type `T`
  - a pointer to an other node
- Linked nodes can be “linked” in a chain to form a list of values



6.20

Example of use of ListNode

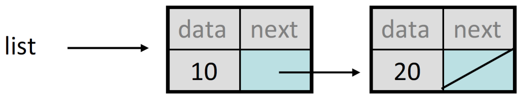
```
int main() {
    ListNode<int>* list = new ListNode<int>();
    list->data = 42;
    list->next = new ListNode<int>();
    list->next->data = -3;
    list->next->next = new ListNode<int>();
    list->next->next->data = 17;
    list->next->next->next = nullptr;
    cout << list->data << "␣"
    << list->next->data << "␣"
    << list->next->next->data << endl; // 42 -3 17
    ...
    return 0;
}
```



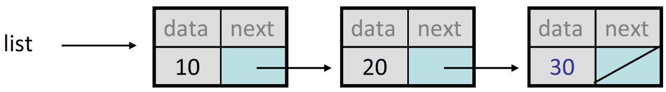
6.21

Linked node: problem 1

- What sequence of operations transforms this list:



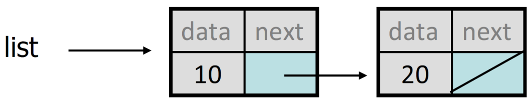
- into this one?



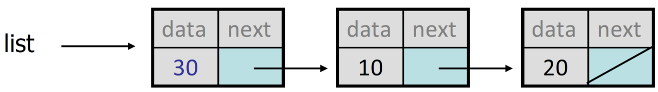
6.22

Linked node: problem 2

- What sequence of operations transforms this list:



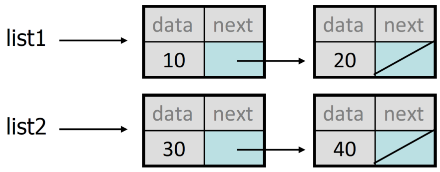
- into this one?



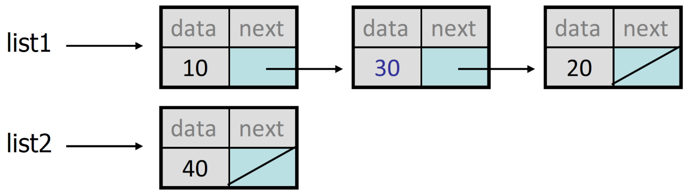
6.23

Linked node: problem 3

- What sequence of operations transforms these lists:



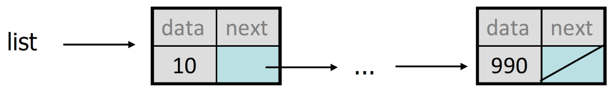
- into these ones?



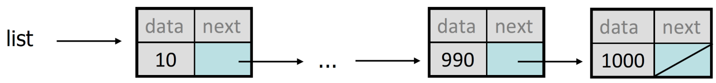
6.24

Linked node: problem 4

- What sequence of operations transforms this list:



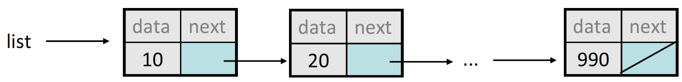
- into this one?



6.25

Linked node: problem 5

- Suppose we have to do a long chain:




```
ListNode<int>* list = new ListNode<int>(10);
list->next = new ListNode(20);
list->next->next = ...;
```

  – We don't know how long the chain should be

- How do we print all the values?

6.26

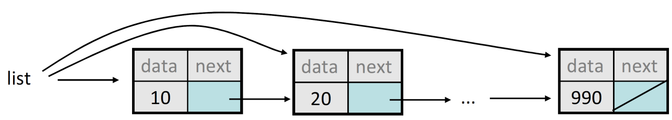
Traversing a list?

- A (bad) way of printing all the value in the list:

```
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next; // move to the next node
}
```

  – What is the problem with this solution?

    * We lose the list while printing it!



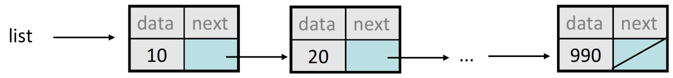
6.27

Traversing a list

- The right way to print all the values in the list:

```
ListNode* current = list;
while (current != nullptr) {
    cout << current->data << endl;
    current = current->next; // move to the next node
}
```

  - Changing `current` does not affect the list

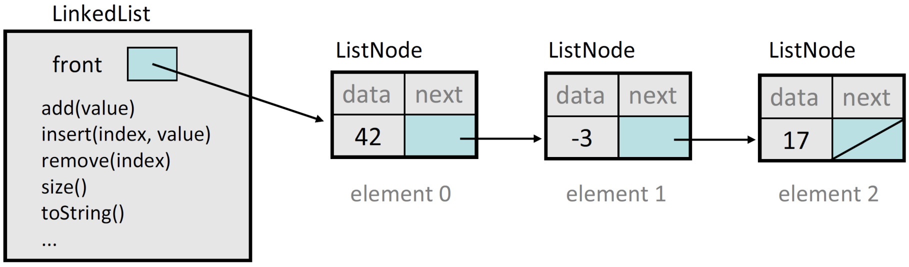


6.28

3 Linked list

`LinkedList`

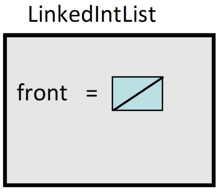
- Lets write a container class `LinkedList`
  - It has similar members to `ArrayList`
    - `add`, `clear`, `get`, `insert`, `isEmpty`, `remove`, `size`, `toString`
  - The list is internally represented as a chain of linked nodes
    - The list has a pointer to the first node in the list
    - `nullptr` marks the end of the list; the list is empty if the first node is `nullptr`



6.29

LinkedList.h

```
template<typename T>
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void add(const T& value);
    void clear();
    T get(int index) const;
    void insert(int index, const T& value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, const T& value);
    int size() const;
    string toString() const;
private:
    int m_size;
    ListNode<T>* m_front;
};
```

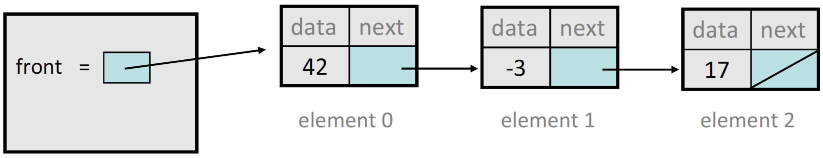


6.30

Implement add

```
// Appends the given value to the end of the list.
template<typename T>
void LinkedList<T>::add(const T& value) {
    ...
}
```

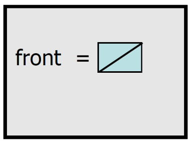
- How do we add a new node to the end of the list?



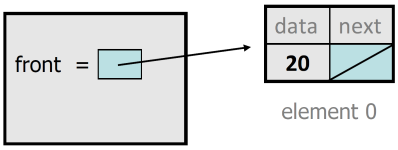
6.31

Adding to an empty list

- Before add(20):



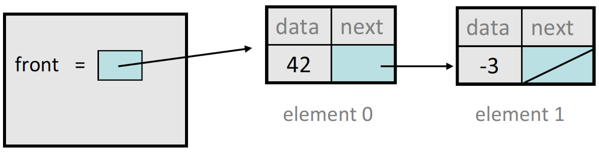
After:



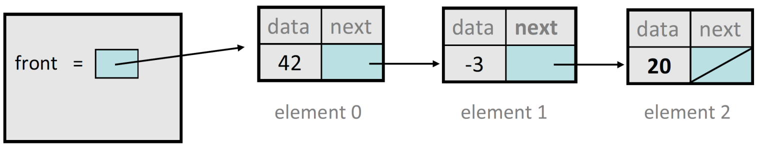
6.32

Add to a non-empty list

- Before add(20)



- After add(20)



6.33

Code for add

```
// Adds the given value to the end of the list.
template<typename T>
void LinkedList::add(const T& value) {
    if (m_front == nullptr) {
        // adding to an empty list
        m_front = new ListNode<T>(value);
    } else {
        // adding to the end of an existing list
        ListNode<T>* current = m_front;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = new ListNode<T>(value);
    }
}
```

6.34

Code for get

```
// Returns value in list at given index.
// Precondition: 0 <= index < size()
template<typename T>
T LinkedList::get(int index) {
    ListNode<T>* current = m_front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

6.35

Code for insert

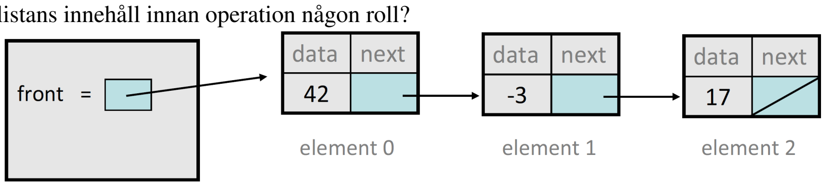
```
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
template<typename T>
void LinkedList<T>::insert(int index, const T& value) {
    if (index == 0) {
        // inserting at front of list
        m_front = new ListNode<T>(value, m_front);
    } else {
        // inserting in general position in list
        ListNode<T>* current = m_front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        current->next = new ListNode<T>(value, current->next);
    }
}
```

6.36

Implementing remove

```
// Removes value at given index from list.
// Precondition: 0 <= index < size
template<typename T>
void LinkedList<T>::remove(int index) {
   ...
}
```

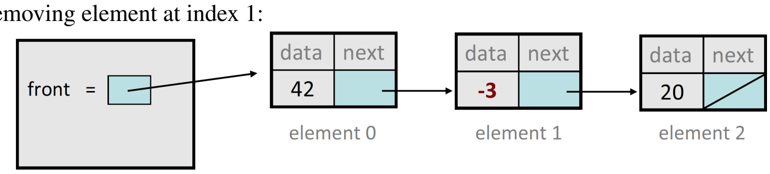
- How do we remove a node from a given position in a list?
- Spelar listans innehåll innan operation någon roll?



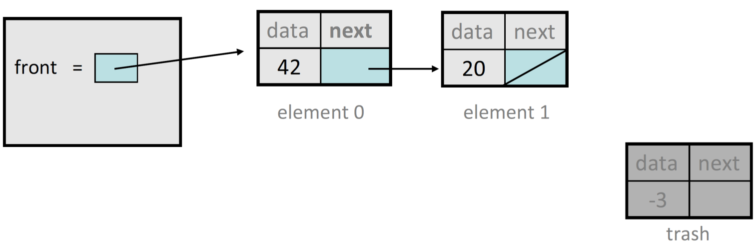
6.37

Remove from a list

- Before removing element at index 1:



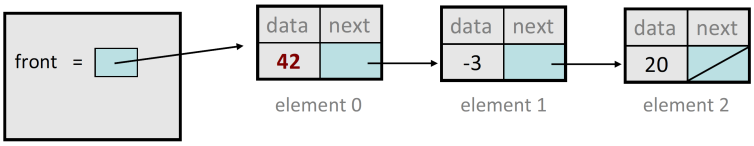
- After:



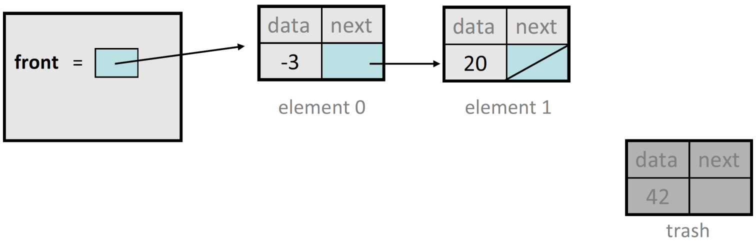
6.38

Remove at the beginning of the list

- Before removing element at index 0:



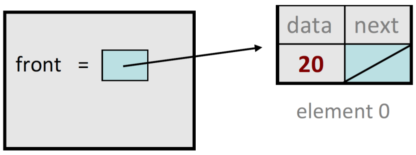
- After:



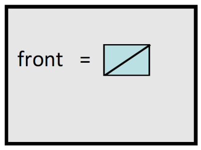
6.39

Removing with a single element

- Before:



After:



  - We must change `front` to be a `nullptr` instead of pointing to a node
  - Do we need a special case to deal with this?

6.40

Code for remove

```
// Removes value at given index from list.
// Precondition: 0 <= index < size()
template<typename T>
void LinkedList::remove(int index) {
    ListNode<T>* trash;
    if (index == 0) { // removing first element
        trash = m_front;
        m_front = m_front->next;
    } else { // removing elsewhere in the list
        ListNode<T>* current = m_front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        trash = current->next;
        current->next = current->next->next;
    }
    delete trash;
}
```

6.41

Is that the only use of Linked Lists?

6.42