Chapter 15: Inheritance

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Linked lists

- A linked list is a commonly used data structure (you should take SMC's data structures course!)
- It is similar to an array, except the memory is not stored in a contiguous block

Abstract data type

- C++ classes are Abstract Data Types (ADT)
    - Provides logical structure for information represented by the class
    - Provides operations to perform on this information

Linked List

- A collection of linked nodes
- Each node contains
    - Some kind of common data/information
    - Address of another node
- The collection grows and shrinks over time
- Nodes are accessed sequentially

Linked list structure

- The linked list class:
    - Logical structure:
        * Has a beginning (referred to as a "head")
        * Has an ending (referred to as a "tail")
        * May be empty
    - Operations
        * insert item
        * remove item
        * return length
        * position at head, tail, successor, predecessor

Linked list implementation

- We could implement a linked list with an array
  - The linked list would then be fixed in size (or we would need to manage resizing, similar to std::vector)
- We will instead implement a linked list with pointers
  - The list can grow and shrink in size easily

Node object

- The node object knows:
  - Its own data (information)
  - The address of the next node in the list
- The node object can:
  - Initialize itself
  - Return its information
  - Set the address of the next node
  - Give the address of the next node

Node class

```cpp
class ListNode; // forward declaration of ListNode class so we can
    typedef without a compile error
typedef ListNode* ListNodePtr;

class ListNode {
public:
  ListNode( const int& data_ = 0, ListNodePtr nextNode = nullptr );

  const int getData() const;
  void setNext( ListNodePtr nextNode );
  ListNodePtr getNext() const;

private:
  int data;
  ListNodePtr next;
};
```

Linked list of Nodes

- First, we'll declare a `typedef` of a pointer to a node: `typedef Node* NodePtr;`
- Next, we'll declare a head of the list and initialize it to `nullptr`: `NodePtr head = nullptr;`
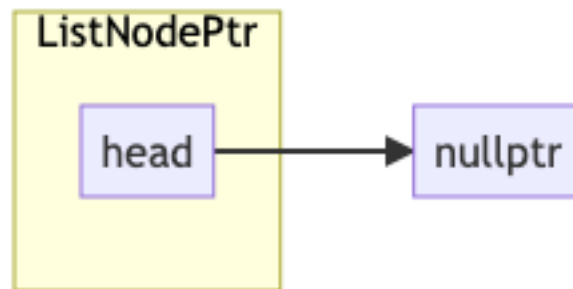

Figure 1: The start of a linked list. Head is empty and points to nullptr

- In the following diagrams, an arrow emitting from a `NodePtr` indicates the value the `NodePtr` is pointing to, while an arrow emitting from a `Node` indicates
- From here, we can create the first node with value 3

```
1  head = new Node(3);
```
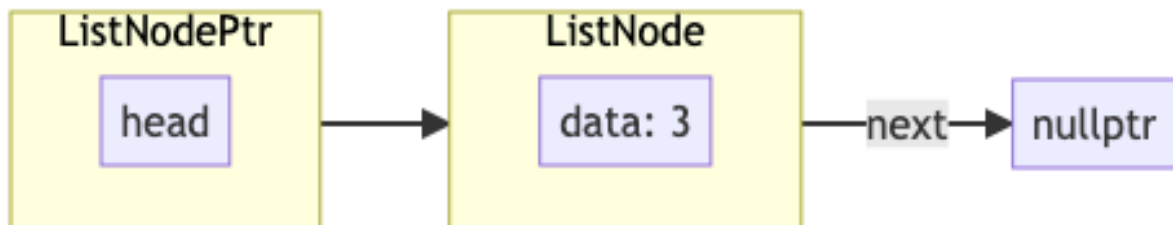
- Now we have the following structure:

Figure 2: The start of a linked list. Head is empty and points to nullptr

- Let's insert another node at the start of the list:

```
1  ListNodePtr newNode = new ListNode(5);
2  newNode->next = head;
3  head = newNode;
```
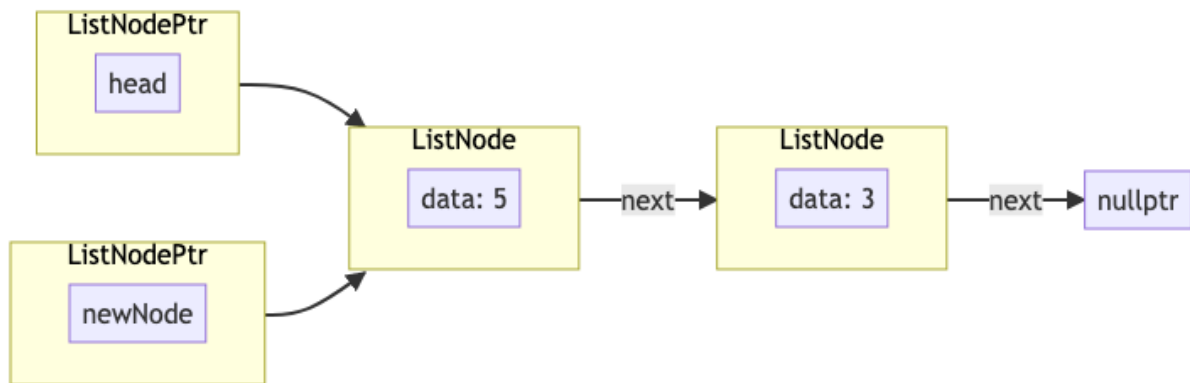
Figure 3: Linked list with two nodes

- Next, let's suppose we want to insert a node between the nodes with data 5 and 3 (i.e. after the node with data = 5)

```
1  newNode = new ListNode(4);
2  newNode->next = head->next;
3  head->next = newNode;
```
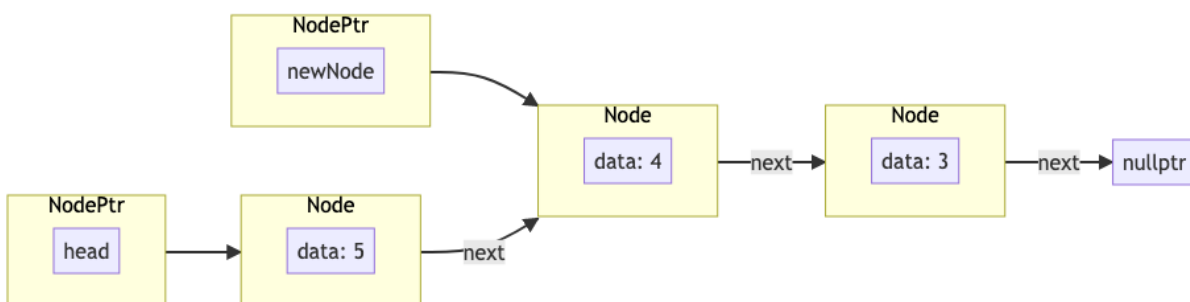

Figure 4: Linked list with three nodes after inserting in middle

Traversing the list

- To walk over all of the nodes in the list, we can do this:

```
1  void printList(ListNodePtr head){
2    ListNodePtr p = head;
3    while (p != nullptr) {
4      cout << p->data << endl;
5      p = p->next;
6    }
7  }
```

Searching the list

- To search for a value in the list, we traverse the list and look for a value

```
1  ListNodePtr findTarget( const int& target_data){
2    ListNodePtr p = head;
3    while (p != nullptr && p->data != target_data) {
4      p = p->next;
5    }
6    return p;
7  }
```

List object

- The LinkedList object knows:
  - Its head node
  - Its tail node
  - Its current size
- The LinkedList object can:
  - Initialize itself
  - Return whether or not it is empty
  - Make the list empty
  - Insert data in the front of the list
  - Insert data in the back of the list
  - Remove data
  - Return the pointer to a target list node

List class

```
1  class List {
2  public:
3    List();
4    ~List();
5    int size() const;
6    void makeEmpty();
7    bool isEmpty( ) const;
8    void push_front( const int& data );
```

```
 9     void push_back( const int& data );
10     void remove( const int& data );
11  private:
12     ListNodePtr head, tail;
13     int listSize;
14
15     ListNodePtr findTarget(const int& data);
16  }
```

- Let's look at a full implementation of this class
    - Take some time to appreciate how pointers allow for sophisticated but carefully constructed data structures
    - The key issue is to hide the complexity from the users of this class

Example ListNode.h

```
 1  #ifndef LISTNODE_H
 2  #define LISTNODE_H
 3  #include <iostream>
 4
 5  namespace cs52 {
 6
 7  class ListNode; // forward declaration of ListNode class so we can
       typedef without a compile error
 8  typedef ListNode* ListNodePtr;
 9
10  class ListNode {
11  public:
12     ListNode( const int& data_ = 0, ListNodePtr nextNode = nullptr );
13
14     const int getData() const;
15     void setNext( ListNodePtr nextNode );
16     ListNodePtr getNext() const;
17
18  private:
19     int data;
20     ListNodePtr next;
21  };
22
23  }
24  #endif
```

Example ListNode.cpp

```
1  #include <iostream>
2  #include "ListNode.h"
3
4  namespace cs52 {
5
6  ListNode::ListNode( const int& data_,
7                      ListNodePtr nextNode ) : data( data_ ), next(
                            nextNode ) {
8  }
9
10 const int ListNode::getData() const {
11     return( data );
12 }
13
14 void ListNode::setNext( ListNodePtr nextNode ) {
15     next = nextNode;
16 }
17
18 ListNodePtr ListNode::getNext() const {
19     return( next );
20 }
21
22 }
```

Example List.h

```
1  #ifndef LIST_H
2  #define LIST_H
3  #include <iostream>
4  #include <exception>
5  #include "ListNode.h"
6
7  namespace cs52 {
8
9  class List {
10 public:
11   List();
12   ~List();
13   int size() const;
14   void makeEmpty();
```

```
15    bool isEmpty( ) const;
16    void push_front( const int& data );
17    void push_back( const int& data );
18    void remove( const int& data );
19
20    // use these two lines if running under linux
21    // friend std::ostream& operator <<() ( std::ostream& outs, const
          List& l );
22    // friend std::ostream& operator <<() ( std::ostream& outs, const
          List* l );
23    // use these two lines if running under windows
24    friend std::ostream& operator << ( std::ostream& outs, const List& l
          );
25    friend std::ostream& operator << ( std::ostream& outs, const List* l
          );
26  private:
27    ListNodePtr head, tail;
28    int listSize;
29
30    std::ostream& printList( std::ostream& outs ) const;
31    ListNodePtr findTarget(const int& data);
32    ListNodePtr findTargetPrev(const int& data);
33    void removeNode(ListNodePtr before_del);
34
35  };
36
37  }
38  #endif
```

Example List.cpp

```
1   #include "List.h"
2   #include "ListNode.h"
3
4   namespace cs52 {
5
6   List::List() {
7     head = nullptr;
8     tail = nullptr;
9     listSize = 0;
10  }
11
```

```
12  List::~List() {
13    // when destructing the object, we empty the object!
14    makeEmpty();
15  }
16
17  bool List::isEmpty() const {
18    return( head == nullptr );
19  }
20
21  void List::makeEmpty() {
22    while (head != nullptr) {
23      remove( head->getData() );
24    }
25    head = tail = nullptr;
26  }
27
28  int List::size() const {
29    return( listSize );
30  }
31
32  void List::push_front( const int& data ) {
33    // place data into a ListNode at the front of the list
34    ListNode* newnode = new ListNode( data );
35    // if this is the first insert, tail needs to be updated as well
36    if (head == nullptr && tail == nullptr) {
37      head = tail = newnode;
38    } else {
39      // set the new node's next to point to the current head
40      newnode->setNext( head );
41      // update the head to be the newnode
42      head = newnode;
43    }
44    listSize++;
45  }
46
47  void List::push_back( const int& data ) {
48    // place data into a ListNode at the back of the list
49    ListNode* newnode = new ListNode( data );
50    // if this is the first insert, head needs to be updated as well
51    if (head == nullptr && tail == nullptr) {
52      head = tail = newnode;
53    } else {
54      // set the current tail's next to be the new node
```

```
55        tail->setNext( newnode );
56        // set the tail to be the new node
57        tail = newnode;
58    }
59    listSize++;
60  }
61
62  void List::remove( const int& data ) {
63    // special case when data is at head
64    if(head != nullptr && head->getData() == data){
65      ListNodePtr temp = head->getNext();
66      // only one value in list, both head and help are going to be
              nullptr
67      if (temp == nullptr){
68        tail = nullptr;
69      }
70      delete(head);
71      head = temp;
72    } else {
73      ListNodePtr previous = findTargetPrev(data);
74      if (previous == nullptr){
75        throw std::logic_error("data to remove not found in list");
76      }
77      ListNodePtr current = previous->getNext();
78      // update the link from previous' next to current's next
79      previous->setNext( current->getNext() );
80      // may need to update tail
81      if (current == tail){
82        tail = previous;
83      }
84      delete( current );
85    }
86    listSize--;
87  }
88
89  std::ostream& operator << ( std::ostream& outs, const List& l) {
90    return( l.printList( outs ) );
91  }
92
93  std::ostream& operator << ( std::ostream& outs, const List* l) {
94    return( l->printList( outs ) );
95  }
96
```

```
 97  std::ostream& List::printList( std::ostream& outs ) const {
 98    if (isEmpty())
 99      outs << "Empty List" << std::endl;
100    else {
101      outs << "List has " << size() << " elements: " << std::endl;
102      ListNode* current = head;
103      while (current != NULL) {
104        outs << current->getData() << " -> ";
105        current = current->getNext();
106      }
107      outs << " NULL";
108      outs << std::endl;
109    }
110    return( outs );
111  }
112
113  ListNodePtr List::findTarget(const int& target_data){
114    // special case when data is at head
115    if(head && head->getData() == target_data){
116      return head;
117    }
118    ListNodePtr p = findTargetPrev(target_data);
119    // if p wasn't nullptr
120    if (p != nullptr)
121    {
122      return p->getNext();
123    }
124    return p;
125  }
126
127  ListNodePtr List::findTargetPrev(const int& target_data){
128    // special cases when we cannot have a previous - empty or only one
         value in list
129    if (head == nullptr || head->getNext() == nullptr) {
130      return nullptr;
131    }
132    ListNodePtr p = head;
133    ListNodePtr cur = p->getNext();
134    while (cur != nullptr && cur->getData() != target_data) {
135      std::cout << p->getData() << " " << cur->getData() << std::endl;
136      p = p->getNext();
137      cur = cur->getNext();
138    }
```

```
139     // need a special case for if we didn't find the value - we should
            return nullptr but p is actually tail
140     if (p->getNext() == nullptr){
141       return nullptr;
142     }
143     return p;
144  }
145
146  // removes the node at prev_node->next()
147  void List::removeNode(ListNodePtr prev_node){
148    ListNodePtr node_to_delete = prev_node->getNext();
149    prev_node->setNext( node_to_delete->getNext() );
150    delete( node_to_delete );
151  }
152
153
154  }
```

Example ListDriver.cpp

```
 1  // ListDriver.cpp : Defines the entry point for the console application
       .
 2  //
 3
 4  #include <iostream>
 5  #include <cstdlib>
 6
 7  #include "List.h"
 8  #include "ListNode.h"
 9
10  enum CHOICE { PRINT, QUIT, PUSH_BACK, PUSH_FRONT, REMOVE, ISEMPTY,
       MAKEEMPTY };
11  CHOICE menu();
12
13  int main(int argc, char* argv[])
14  {
15    using namespace cs52;
16    using namespace std;
17
18    List l;
19    CHOICE c;
20    int value;
```

```
21
22    do {
23      c = menu();
24      switch( c ) {
25      case PRINT:
26        cout << l;
27        break;
28      case ISEMPTY:
29        if (l.isEmpty()) {
30          cout << "list is empty" << endl;
31        }
32        else {
33          cout << "list is not empty" << endl;
34        }
35        break;
36      case MAKEEMPTY:
37        l.makeEmpty();
38        break;
39      case PUSH_BACK:
40        cout << "enter an int to insert at the back of the list: ";
41        cin  >> value;
42        l.push_back( value );
43        break;
44      case PUSH_FRONT:
45        cout << "enter an int to insert at the front of the list: ";
46        cin  >> value;
47        l.push_front( value );
48        break;
49      case REMOVE:
50        cout << "enter an int to remove: ";
51        cin  >> value;
52        l.remove( value );
53        break;
54      }
55    } while (c != QUIT);
56
57    return( 0 );
58 }
59
60 CHOICE menu() {
61    using namespace std;
62    char c;
63    CHOICE result;
```

```
64    cout << "i(S)empty (M)akeEmpty Push(F)ront Push(B)ack (R)emove (P)
          rint (Q)uit: ";
65    cin  >> c;
66    switch( c ) {
67    case 'S':
68    case 's':
69      result = ISEMPTY;
70      break;
71    case 'M':
72    case 'm':
73      result = MAKEEMPTY;
74      break;
75    case 'B':
76    case 'b':
77      result = PUSH_BACK;
78      break;
79    case 'F':
80    case 'f':
81      result = PUSH_FRONT;
82      break;
83    case 'R':
84    case 'r':
85      result = REMOVE;
86      break;
87    case 'P':
88    case 'p':
89      result = PRINT;
90      break;
91    case 'Q':
92    case 'q':
93      result = QUIT;
94      break;
95    default:
96      result = menu();
97    }
98    return( result );
99  }
```

Linked list pros and cons

- Pros - Easy
  - insertion

- deletion
- splitting
- joining
- Cons - Hard
    - Traversal is tedious compared to arrays
    - Expensive in terms of space

Linked lists vs. Arrays

- Arrays
    - Static in allocation size
    - Removed items leave wasted space -> O(n)
    - Insertion has more overhead -> O(n)
    - Element access -> O(1)
- Linked lists
    - Expensive to walk/iterate -> O(n)
    - Removing item -> O(1)
    - Inserting item -> O(1)
- Neither is better than the other, they are just different. Use both of them wisely and when they make sense.