Chapter 13: Linked Lists

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

```
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;`

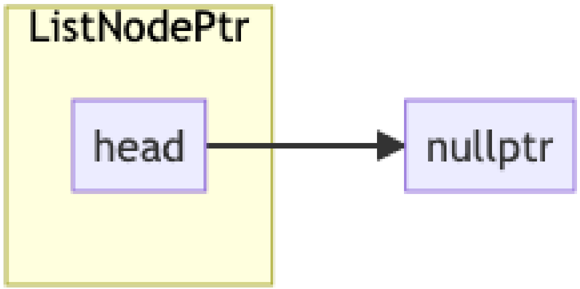

Figure 1: The start of a linked list. Head is empty and points to nullptr

- 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

```
head = new Node(3);
```

- Now we have the following structure:

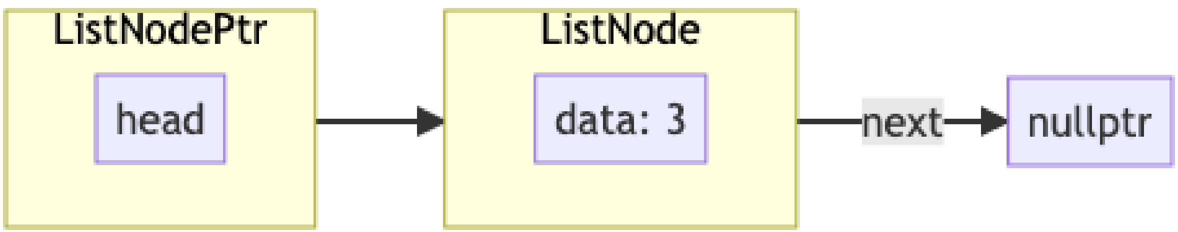

Figure 2: The start of a linked list. Head is empty and points to nullptr

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

```
ListNodePtr newNode = new ListNode(5);
newNode->next = head;
head = newNode;
```

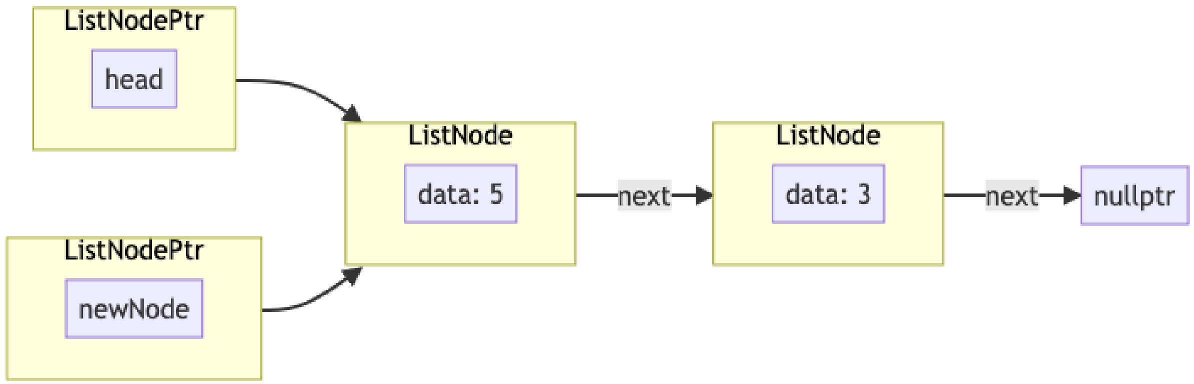

Figure 3: Linked list with two nodes

- 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)

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

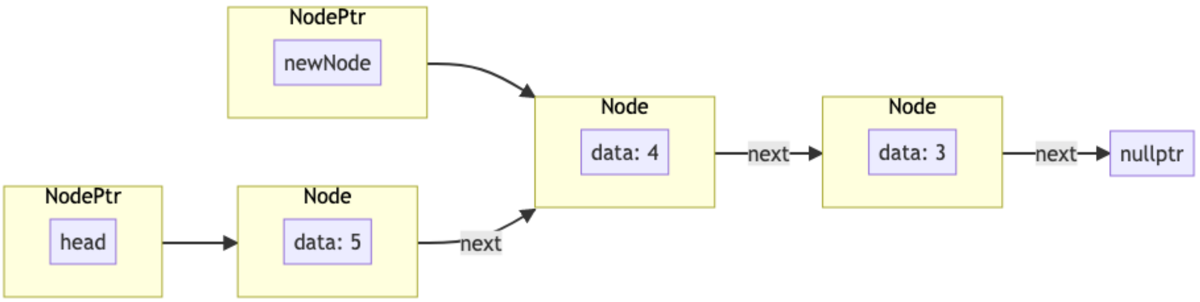

Figure 4: Linked list with three nodes after inserting in middle

Traversing the list

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

```
void printList(ListNodePtr head){
  ListNodePtr p = head;
  while (p != nullptr) {
    cout << p->data << endl;
    p = p->next;
  }
}
```

Searching the list

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

```
ListNodePtr findTarget( const int& target_data){
  ListNodePtr p = head;
  while (p != nullptr && p->data != target_data) {
    p = p->next;
  }
  return p;
}
```

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

```
class List {
public:
  List();
  ~List();
  int size() const;
  void makeEmpty();
  bool isEmpty( ) const;
  void push_front( const int& data );
```

```
  void push_back( const int& data );
  void remove( const int& data );
private:
  ListNodePtr head, tail;
  int listSize;

  ListNodePtr findTarget(const int& data);
}
```

- 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

```
#ifndef LISTNODE_H
#define LISTNODE_H
#include <iostream>

namespace cs52 {

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;
};

}
#endif
```

Example ListNode.cpp

```
#include <iostream>
#include "ListNode.h"

namespace cs52 {

ListNode::ListNode( const int& data_,
                    ListNodePtr nextNode ) : data( data_ ), next(
                       nextNode ) {
}

const int ListNode::getData() const {
    return( data );
}

void ListNode::setNext( ListNodePtr nextNode ) {
    next = nextNode;
}

ListNodePtr ListNode::getNext() const {
    return( next );
}

}
```

Example List.h

```
#ifndef LIST_H
#define LIST_H
#include <iostream>
#include <exception>
#include "ListNode.h"

namespace cs52 {

class List {
public:
  List();
  ~List();
  int size() const;
  void makeEmpty();
```

```
  bool isEmpty( ) const;
  void push_front( const int& data );
  void push_back( const int& data );
  void remove( const int& data );

  // use these two lines if running under linux
  // friend std::ostream& operator <<() ( std::ostream& outs, const
     List& l );
  // friend std::ostream& operator <<() ( std::ostream& outs, const
     List* l );
  // use these two lines if running under windows
  friend std::ostream& operator << ( std::ostream& outs, const List& l
     );
  friend std::ostream& operator << ( std::ostream& outs, const List* l
     );
private:
  ListNodePtr head, tail;
  int listSize;

  std::ostream& printList( std::ostream& outs ) const;
  ListNodePtr findTarget(const int& data);
  ListNodePtr findTargetPrev(const int& data);
  void removeNode(ListNodePtr before_del);

};

}
#endif
```

Example List.cpp

```
#include "List.h"
#include "ListNode.h"

namespace cs52 {

List::List() {
  head = nullptr;
  tail = nullptr;
  listSize = 0;
}

```

```
List::~List() {
  // when destructing the object, we empty the object!
  makeEmpty();
}

bool List::isEmpty() const {
  return( head == nullptr );
}

void List::makeEmpty() {
  while (head != nullptr) {
    remove( head->getData() );
  }
  head = tail = nullptr;
}

int List::size() const {
  return( listSize );
}

void List::push_front( const int& data ) {
  // place data into a ListNode at the front of the list
  ListNode* newnode = new ListNode( data );
  // if this is the first insert, tail needs to be updated as well
  if (head == nullptr && tail == nullptr) {
    head = tail = newnode;
  } else {
    // set the new node's next to point to the current head
    newnode->setNext( head );
    // update the head to be the newnode
    head = newnode;
  }
  listSize++;
}

void List::push_back( const int& data ) {
  // place data into a ListNode at the back of the list
  ListNode* newnode = new ListNode( data );
  // if this is the first insert, head needs to be updated as well
  if (head == nullptr && tail == nullptr) {
    head = tail = newnode;
  } else {
    // set the current tail's next to be the new node
```

```
    tail->setNext( newnode );
    // set the tail to be the new node
    tail = newnode;
  }
  listSize++;
}

void List::remove( const int& data ) {
  // special case when data is at head
  if(head != nullptr && head->getData() == data){
    ListNodePtr temp = head->getNext();
    // only one value in list, both head and help are going to be
        nullptr
    if (temp == nullptr){
      tail = nullptr;
    }
    delete(head);
    head = temp;
  } else {
    ListNodePtr previous = findTargetPrev(data);
    if (previous == nullptr){
      throw std::logic_error("data to remove not found in list");
    }
    ListNodePtr current = previous->getNext();
    // update the link from previous' next to current's next
    previous->setNext( current->getNext() );
    // may need to update tail
    if (current == tail){
      tail = previous;
    }
    delete( current );
  }
  listSize--;
}

std::ostream& operator << ( std::ostream& outs, const List& l) {
  return( l.printList( outs ) );
}

std::ostream& operator << ( std::ostream& outs, const List* l) {
  return( l->printList( outs ) );
}

```

```
std::ostream& List::printList( std::ostream& outs ) const {
  if (isEmpty())
    outs << "Empty List" << std::endl;
  else {
    outs << "List has " << size() << " elements: " << std::endl;
    ListNode* current = head;
    while (current != NULL) {
      outs << current->getData() << " -> ";
      current = current->getNext();
    }
    outs << " NULL";
    outs << std::endl;
  }
  return( outs );
}

ListNodePtr List::findTarget(const int& target_data){
  // special case when data is at head
  if(head && head->getData() == target_data){
    return head;
  }
  ListNodePtr p = findTargetPrev(target_data);
  // if p wasn't nullptr
  if (p != nullptr)
  {
    return p->getNext();
  }
  return p;
}

ListNodePtr List::findTargetPrev(const int& target_data){
  // special cases when we cannot have a previous - empty or only one
     value in list
  if (head == nullptr || head->getNext() == nullptr) {
    return nullptr;
  }
  ListNodePtr p = head;
  ListNodePtr cur = p->getNext();
  while (cur != nullptr && cur->getData() != target_data) {
    std::cout << p->getData() << " " << cur->getData() << std::endl;
    p = p->getNext();
    cur = cur->getNext();
  }
```

```
  // need a special case for if we didn't find the value - we should
     return nullptr but p is actually tail
  if (p->getNext() == nullptr){
    return nullptr;
  }
  return p;
}

// removes the node at prev_node->next()
void List::removeNode(ListNodePtr prev_node){
  ListNodePtr node_to_delete = prev_node->getNext();
  prev_node->setNext( node_to_delete->getNext() );
  delete( node_to_delete );
}


}
```

Example ListDriver.cpp

```
// ListDriver.cpp : Defines the entry point for the console application
   .
//

#include <iostream>
#include <cstdlib>

#include "List.h"
#include "ListNode.h"

enum CHOICE { PRINT, QUIT, PUSH_BACK, PUSH_FRONT, REMOVE, ISEMPTY,
    MAKEEMPTY };
CHOICE menu();

int main(int argc, char* argv[])
{
  using namespace cs52;
  using namespace std;

  List l;
  CHOICE c;
  int value;
```

```

  do {
    c = menu();
    switch( c ) {
    case PRINT:
      cout << l;
      break;
    case ISEMPTY:
      if (l.isEmpty()) {
        cout << "list is empty" << endl;
      }
      else {
        cout << "list is not empty" << endl;
      }
      break;
    case MAKEEMPTY:
      l.makeEmpty();
      break;
    case PUSH_BACK:
      cout << "enter an int to insert at the back of the list: ";
      cin  >> value;
      l.push_back( value );
      break;
    case PUSH_FRONT:
      cout << "enter an int to insert at the front of the list: ";
      cin  >> value;
      l.push_front( value );
      break;
    case REMOVE:
      cout << "enter an int to remove: ";
      cin  >> value;
      l.remove( value );
      break;
    }
  } while (c != QUIT);

  return( 0 );
}

CHOICE menu() {
  using namespace std;
  char c;
  CHOICE result;
```

```
  cout << "i(S)empty (M)akeEmpty Push(F)ront Push(B)ack (R)emove (P)
     rint (Q)uit: ";
  cin  >> c;
  switch( c ) {
  case 'S':
  case 's':
    result = ISEMPTY;
    break;
  case 'M':
  case 'm':
    result = MAKEEMPTY;
    break;
  case 'B':
  case 'b':
    result = PUSH_BACK;
    break;
  case 'F':
  case 'f':
    result = PUSH_FRONT;
    break;
  case 'R':
  case 'r':
    result = REMOVE;
    break;
  case 'P':
  case 'p':
    result = PRINT;
    break;
  case 'Q':
  case 'q':
    result = QUIT;
    break;
  default:
    result = menu();
  }
  return( result );
}
```

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.