Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 6

Date: October 5, 2015

What we’re learning in this Slide Set:

- Nodes and Linked Lists
- Stacks
- Queues

Textbook Chapters:

Relevant to this slide set:

- Chapter 13

Coming soon:

- Chapter 14
- Chapter 15

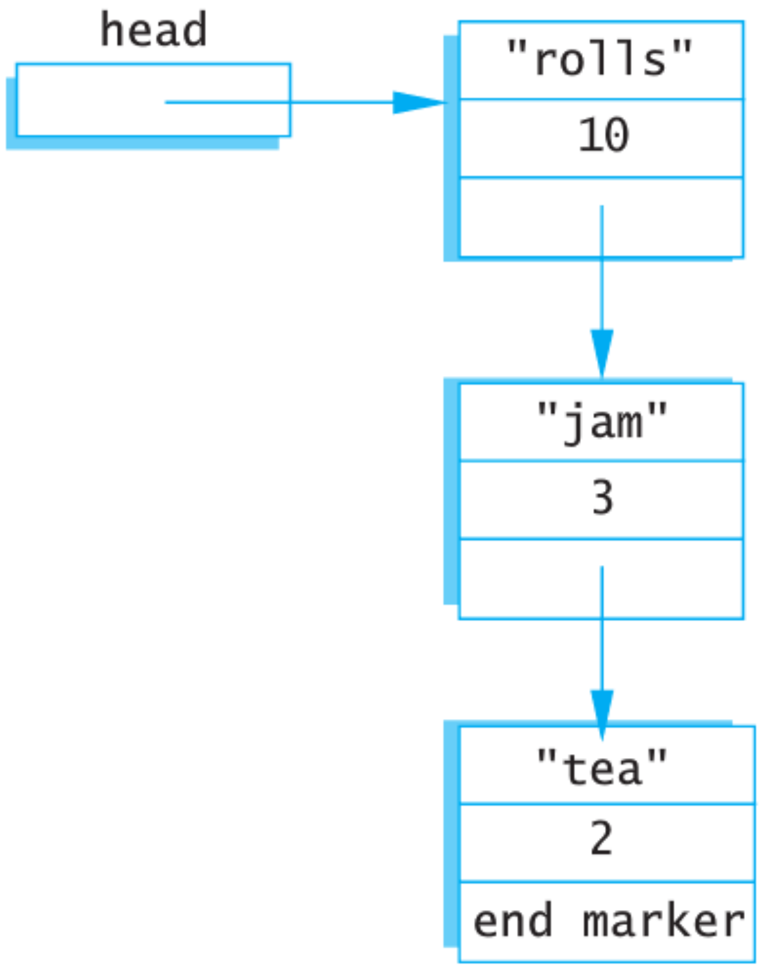
A linked list is constructed using a set of nodes connected by pointers

- It has no fixed size
- It can grow and shrink during the run time of a program
- It is the underlying structure of many key programming abstraction
  - (queues, stacks, and trees)
- Typically nodes are comprised of complex structures (structs or classes)

For example look at this definition of a ListNode:

```
struct ListNode
{
     string item;
     int count;
     ListNode *link;
};
typedef ListNode* ListNodePtr;
```

The corresponding list would look like this:



To use this in a program:

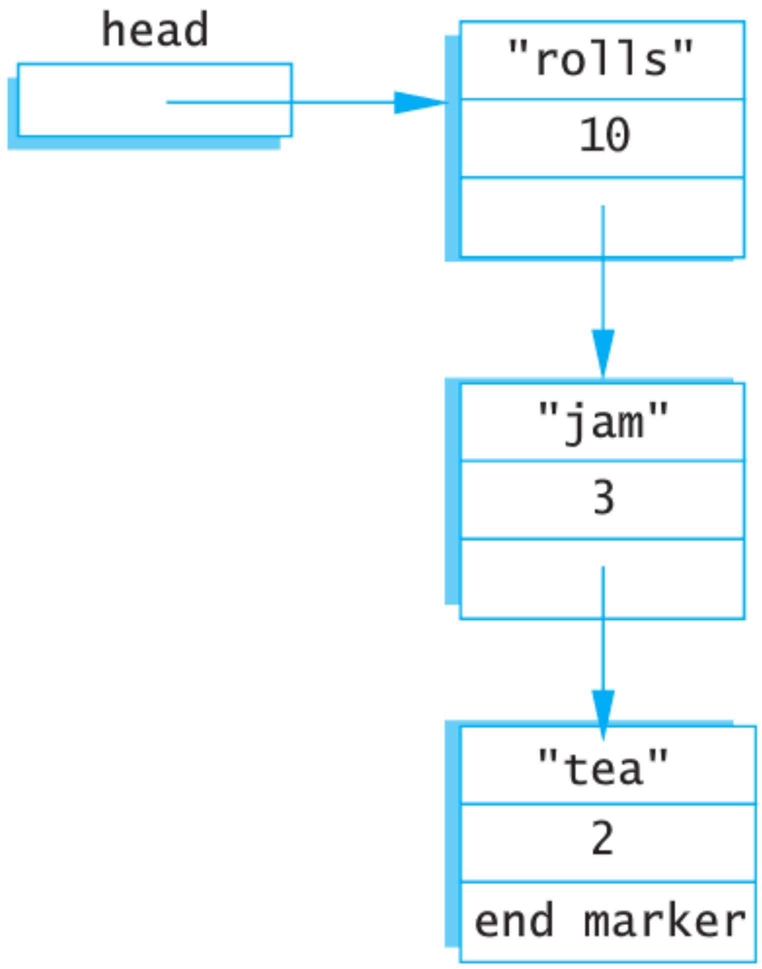
First you define the struct (to create the new type)

Note we have also used typdef to create a pointer to type ListNode

In our list example, we also have a pointer head. Note that it is only a pointer to a node and not a node itself:

```
ListNodePtr head;
```

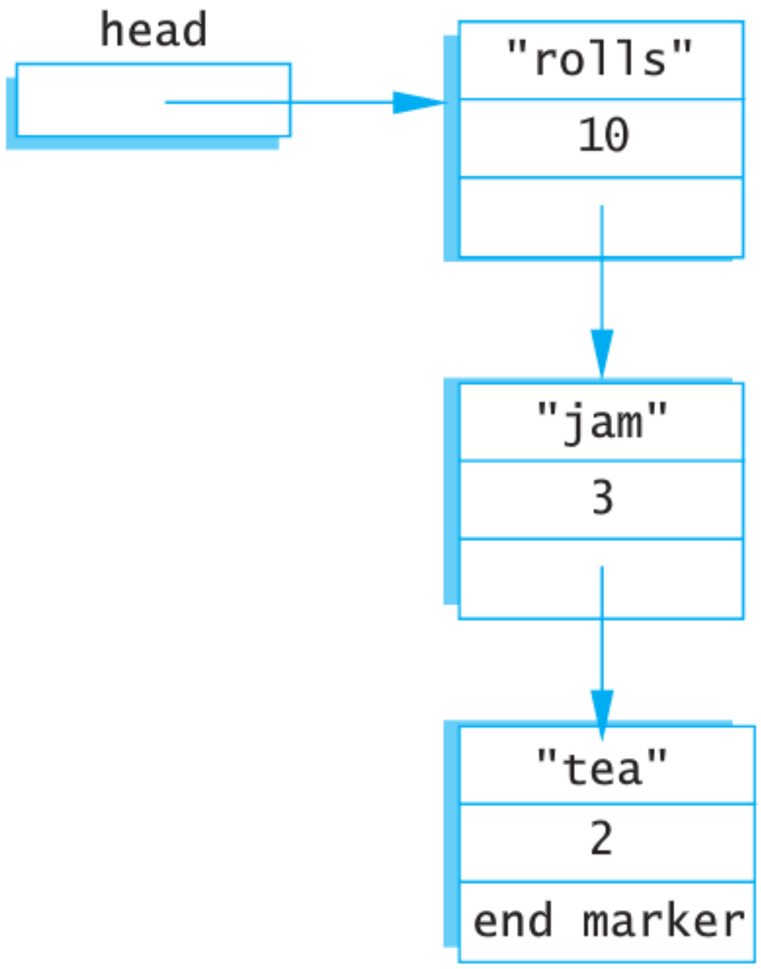
```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
```



Although it is not illegal, the definition of the struct type ListNode is inherently circular (as it contains a pointer to its own type):

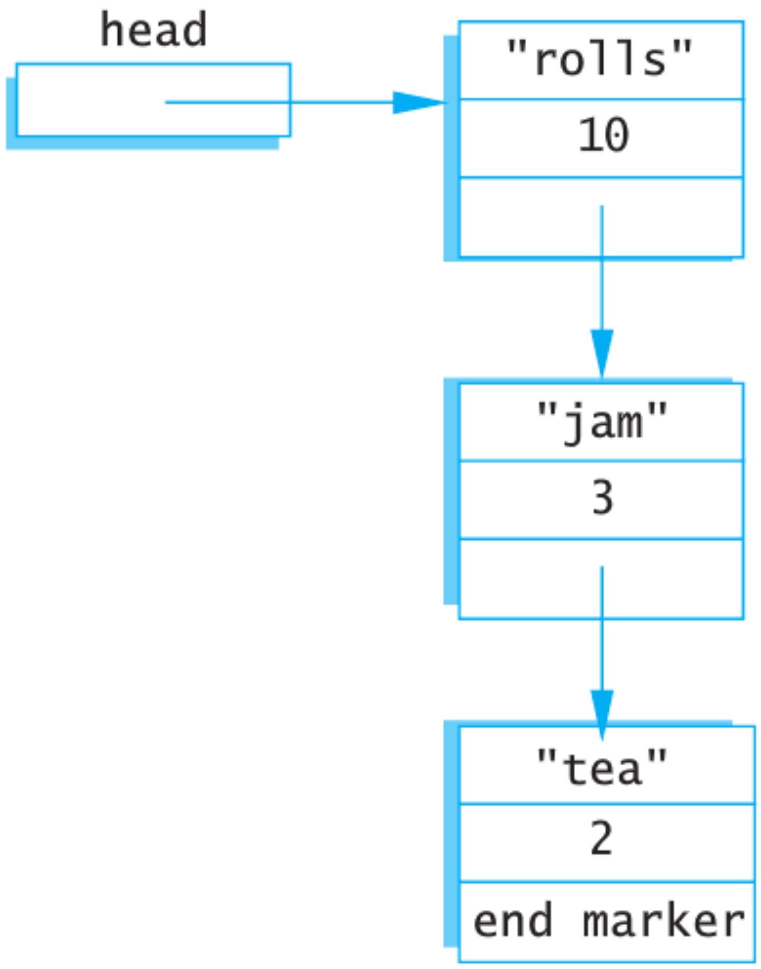
```
struct ListNode
{
     string item;
     int count;
     ListNode *link;
};
typedef ListNode* ListNodePtr;
```

In fact, it is the ability to point to its own type that lets us create a linked list:



If you want to change the number of rolls you are going to buy (according to the list) from 10 to 12, you can use the following statement:

```
(*head).count = 12;
```



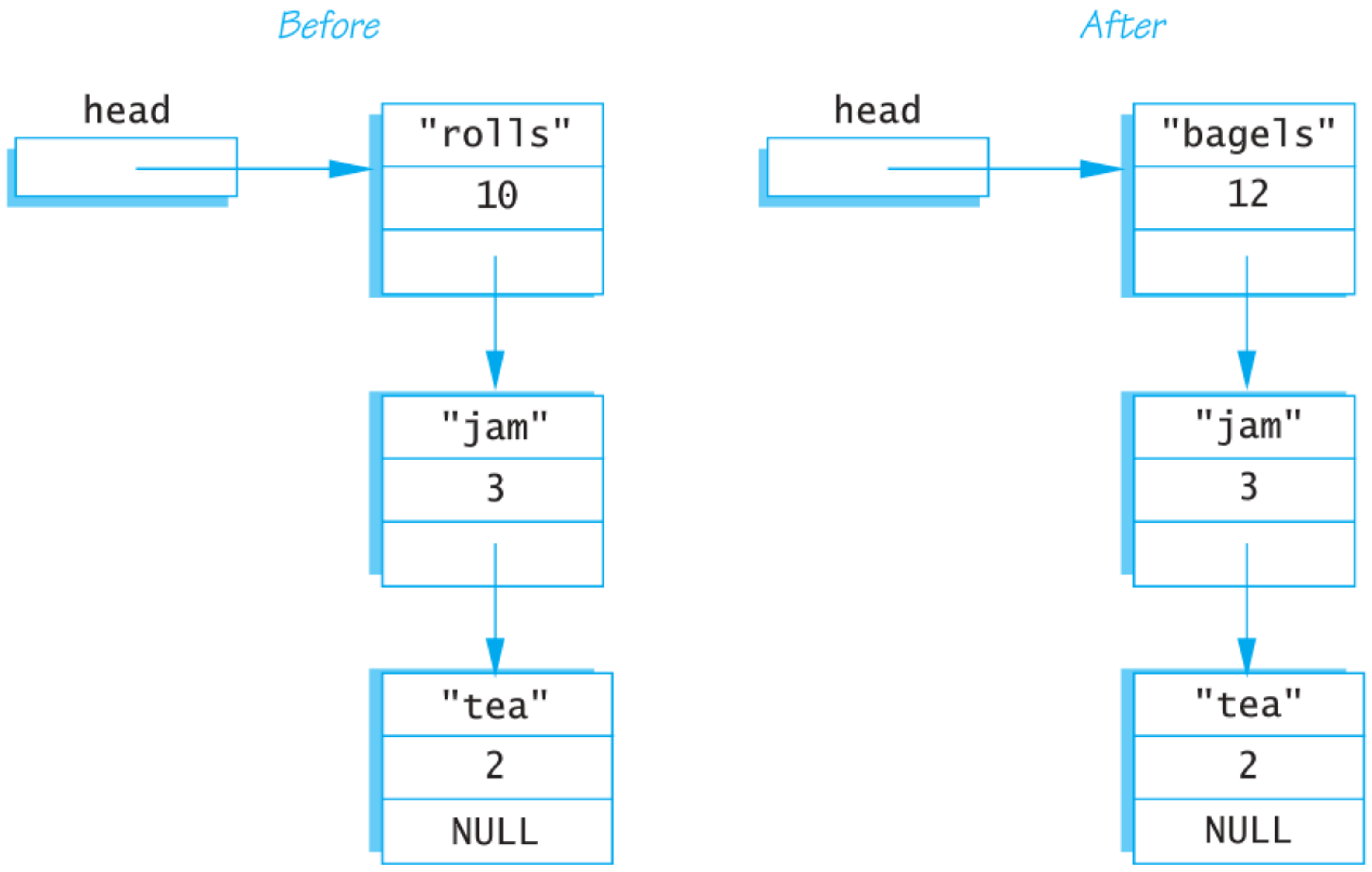
The parentheses are not optional as you want to dereference head (but the dot operator would be performed first without the parentheses).

Generally, people use the arrow operator -> as it is a less confusing notation. It combines the dereferencing operator * with the dot operator.

```
head->count = 12;
```

The arrow operator is commonly used to specify a member of a *dynamic struct* (or object) that is pointed to by a given pointer.

```
head->count = 12;
head->item = "bagels";
```



The arrow operator is commonly used to specify a member of a *dynamic struct* (or object) that is pointed to by a given pointer.

Generically, you use the arrow operator as:

```
Pointer_Variable->Member_Name
```

For another example:

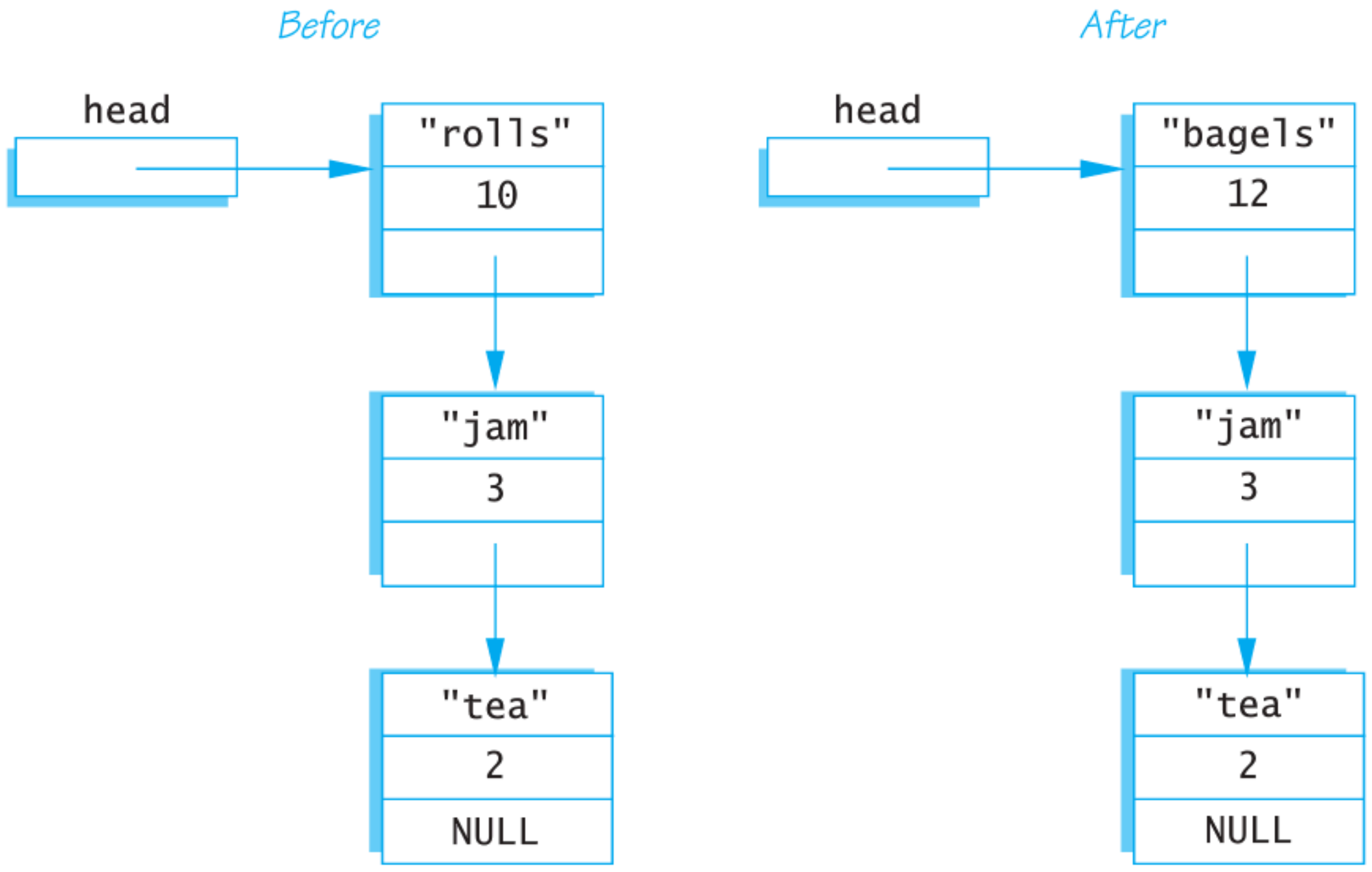
```
struct Record
{
    int number;
    char grade;
};
```

HINT: Remember to allocate memory before you start trying to make assignments

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

The end of the list is specified by the constant NULL

It is a special defined constant that is part of the C++ language in the standard libraries



NULL has two purposes:

- It is used to give a value to a pointer that would otherwise be undefined (not have any value)
  - This prevents potential unintentional references to memory that is not part of your program (which would crash your program)
- It is also used as an “end marker” to indicate the termination/end of a linked list.
- For some compilers, NULL is actually the number 0.
  - But for readability and clarity people use the term NULL (as it is specifically designed for the context of pointers.

It is defined in <iostream> and <cstddef> so you need to include one of these (or a similar library) to use NULL.

You don’t need to use a “using” directive in order to make NULL available to your program

Since it is a special defined constant, the preprocessor will replace it with “0” before the compiler runs so there are no namespace issues.

To set a pointer to NULL, you simply:

```
double *there = NULL;
```

Remember, the constant NULL can be assigned to a pointer of any type because all pointers are addresses.

Since NULL is actually the number zero, there is an ambiguity problem.

Which function would get called if they were overloaded as follows:

```
void func(int *p);
void func(int i);
```

and we made the call `func(NULL);` ?

Since both are equally valid C++11 resolves this by introducing a new constant, nullptr.

- It is not the integer zero, but it is a literal constant used to represent a null pointer.
- You can use it anywhere you would use NULL for a pointer:

```
double *there = nullptr;
```

A linked list is a list of nodes in which each node has a member variable that is a pointer that points to the next/previous node in the list.

- The first node in the list is called the head, the last is called the tail.
- Neither the head nor tail are the actual nodes at the head/tail of the list, they are instead pointers to them
  - Note: the textbook doesn't include reference to the idea of a tail or tail pointer as it is less common.
- The last node in the list points to NULL (as there is nothing after it)
- When you "traverse" the list, you can check if you are at the last node by checking to see if the "next" pointer in the list points to NULL.

Now let's look at some basic functions for manipulating linked lists.

If I define the following simple Node type:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
```

We next need a “head” pointer:

```
NodePtr head;
```

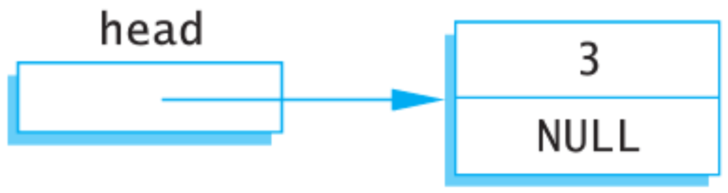
and the ability to store something there:

```
head = new Node;
```

We can then initialize the node to have the following values:

```
head->data = 3;
head->link = NULL;
```

The result of these actions is the following:



WARNING: You always need to maintain a pointer variable to the start (head) of your linked list. Not only is it a way to name the list, but it is a way to ensure that you don't "lose" any of your nodes. Finally, it provides you with a generic way to insert/remove nodes from your list so that you can create generic functions for manipulating linked lists.

Let's look at a repeatable/systematic way to insert and remove items from the head (and/or tail) of a list.

```
void head_insert(NodePtr& head, int the_number);
```

Our function is going to have to use the new operator to create a new node, copy the data into that node, and then insert it into the list.

Let's look at some pseudocode.

```
void head_insert(NodePtr& head, int the_number);
```

Pseudocode for head_insert function;

1. Create a new dynamic variable (new node) pointed to by “temp_ptr”
2. Copy the data into the new node pointed to by “temp_ptr”
3. Make the link member of this new node point to the first node in the original linked list (the head).
4. Make the head pointer variable point to the new node pointed to by temp_ptr.

Here is the code for steps 3 & 4

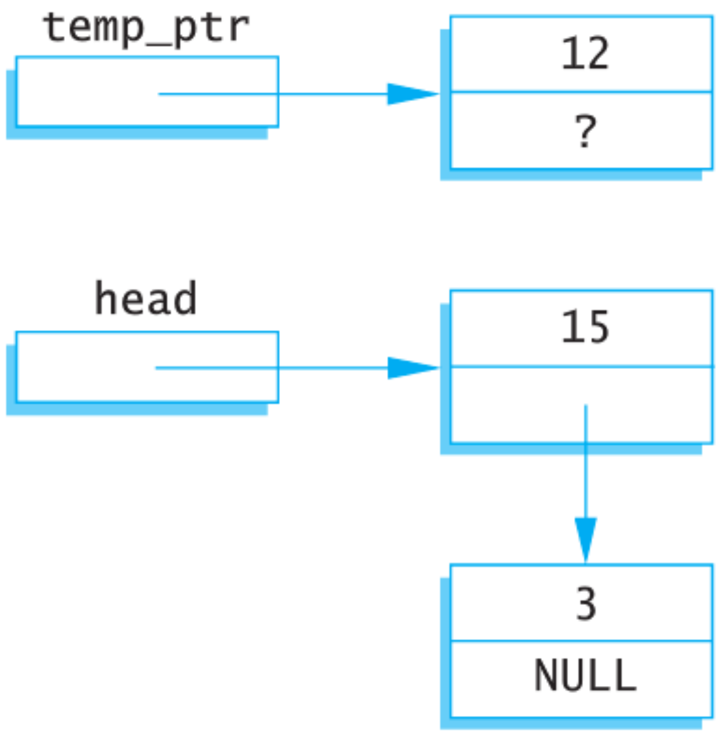
```
temp_ptr->link = head;
head = temp_ptr;
```

```
void head_insert(NodePtr& head, int the_number);
```

Realistically, you should be drawing out these algorithms before you write them until you are ***VERY*** comfortable with pointers and linked lists (i.e. long after you have taken this class).

So let’s do that:

1. Create a new dynamic variable (new node) pointed to by “temp_ptr”
2. Copy the data into the new node pointed to by “temp_ptr”

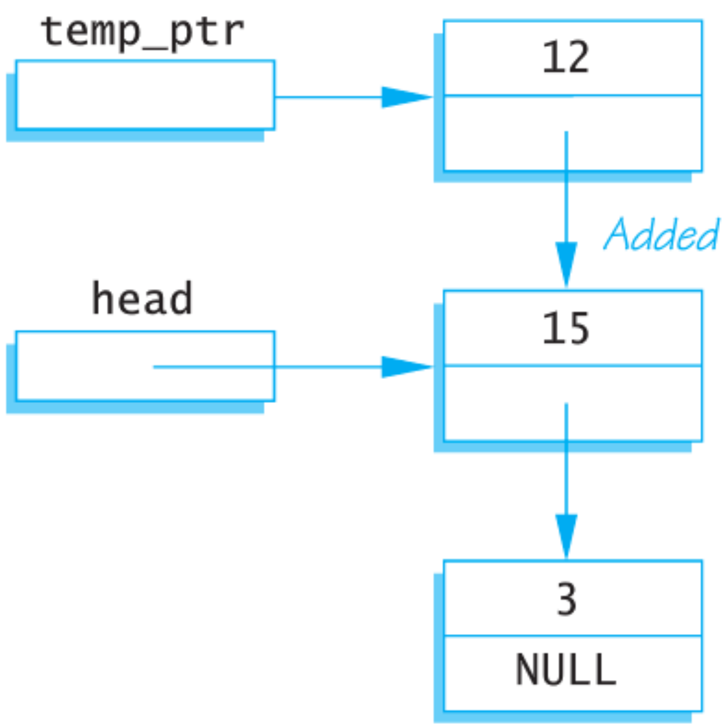


```
void head_insert(NodePtr& head, int the_number);
```

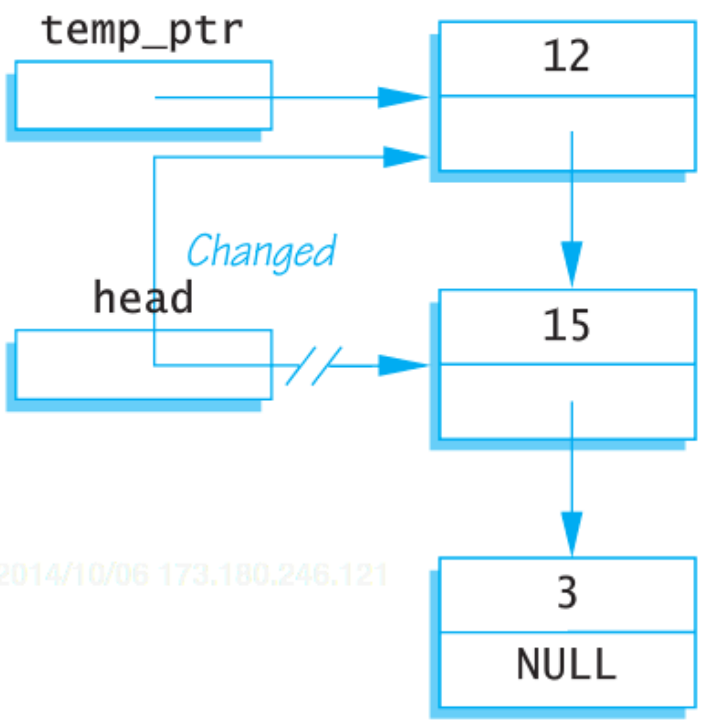
3. Make the link member of this new node point to the first node in the original linked list (the head).

4. Make the head pointer variable point to the new node pointed to by temp_ptr.

3. `temp_ptr->link = head;`

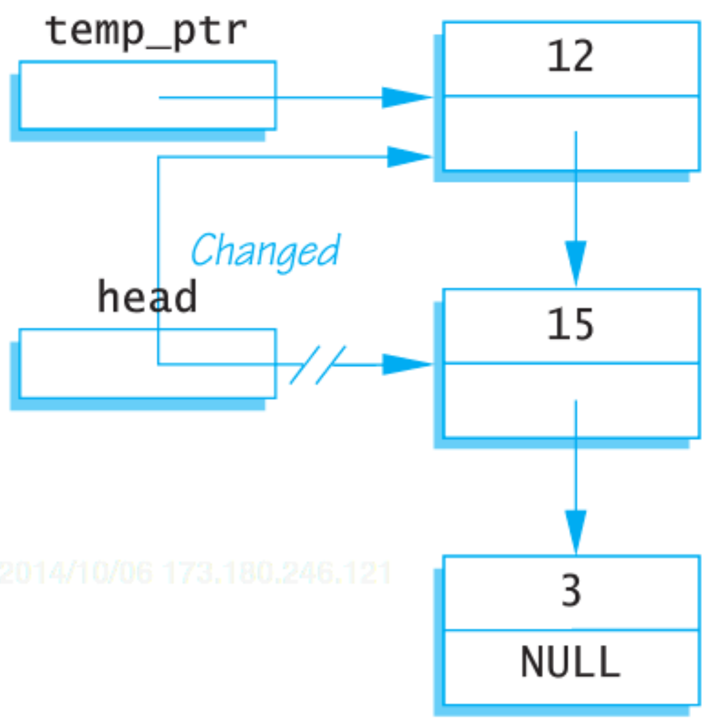


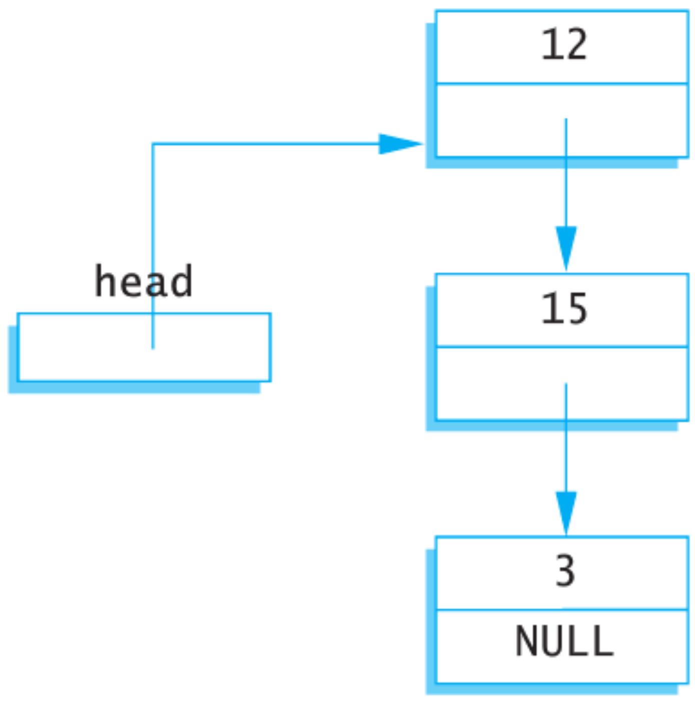
4. `head = temp_ptr;`



```
void head_insert(NodePtr& head, int the_number);
```

After the function call you have:





Special notes:

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;
}
```

Looking at this code, what happens if the original list is an ***empty list*** (i.e. the head pointer currently points to none)? Does it work?

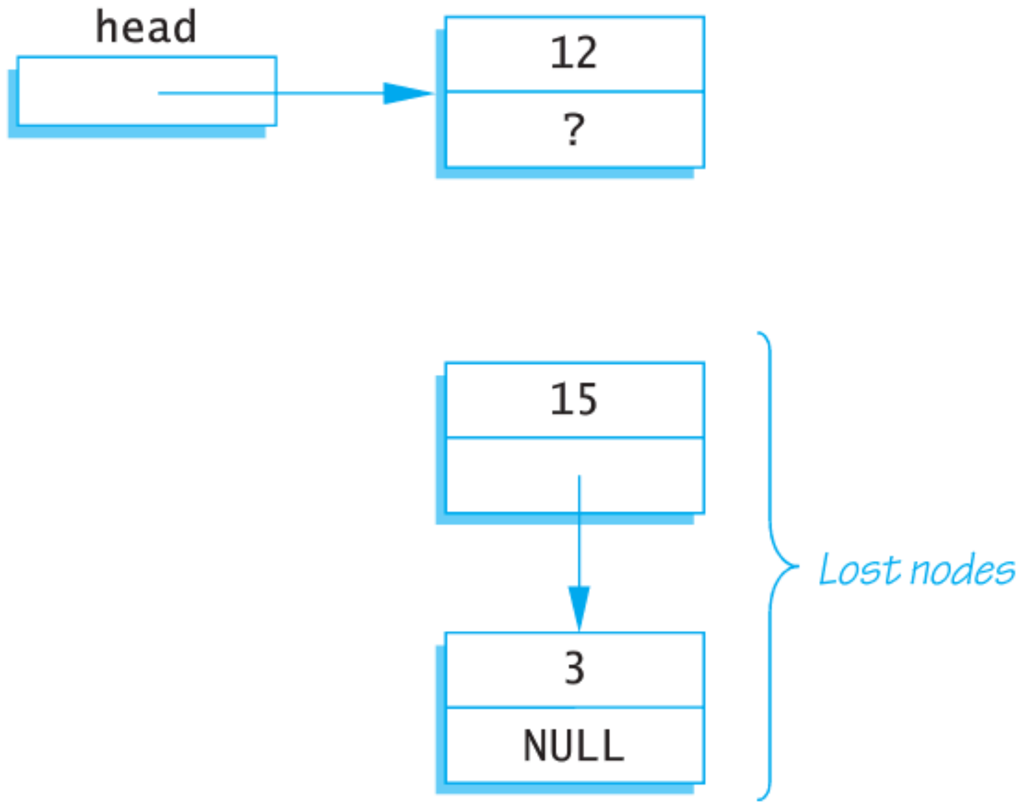
- This is a special corner case you always need to check when designing functions to manipulate lists.

Special notes:

Why did we have a temp_ptr point to the new node instead of having head point to the new node:

```
head = new Node;
head->data = the_number;
```

What happens to the original list when we do this?



We don't have anything pointing to it anymore…

It's lost and now a memory leak in our program that may cause it to crash:

- Worse it could crash the O/S (but that is harder/less likely)

What if we want to search a linked list?

```
NodePtr search(NodePtr head, int target);
//Precondition: The pointer head points to the head of
//a linked list. The pointer variable in the last node
//is NULL. If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that
//contains the target. If no node contains the target,
//the function returns NULL.
```

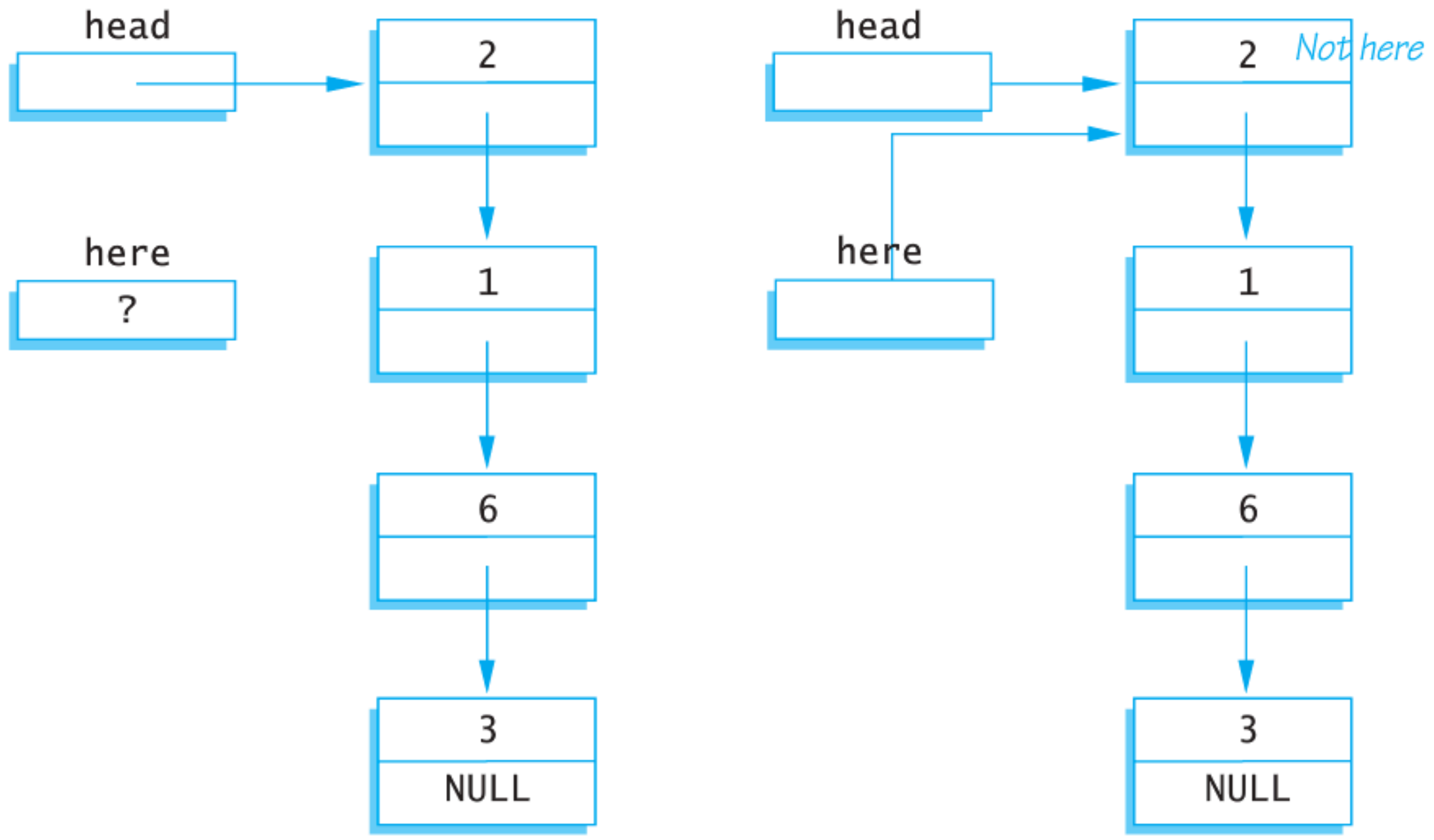
This function returns a pointer to the ***first*** node that contains the target value. If no node in the list contains the target value, it returns NULL.

Since an empty list is a little more difficult, let’s assume only non-empty lists to start with.

```
NodePtr search(NodePtr head, int target);
```

Pseudocode for search function.

1. Make the temp variable "here" point to the first node in the linked list (same as the head pointer).

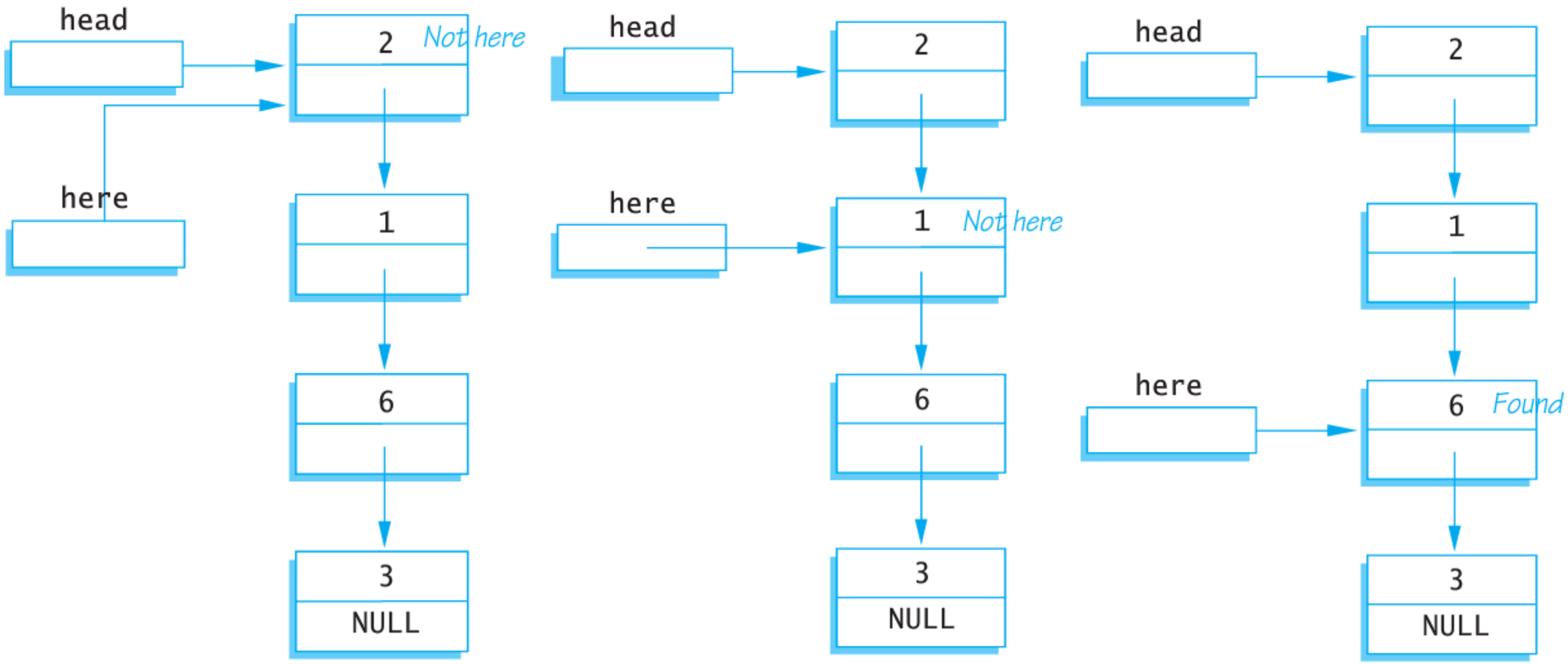


Let's assume the "target" is 6 in this example.

```
NodePtr search(NodePtr head, int target);
```

Pseudocode for search function.

```
2. while ("here" is not pointing to a node containing the target and here
   is not pointing to the last node)

       Make "here" point to the next node in the list.
```



Let’s assume the “target” is 6 in this example.

```
NodePtr search(NodePtr head, int target);
```

Pseudocode for search function.

```
3. if (the node pointed to by here contains "target")
           return here;
   else
           return NULL;
```

Special Notes:

How do we traverse the list?

By having here update it’s value after each comparison to the next item in the list

```
here = here->link;
```

So a preliminary version of the code looks like:

```
here = head;

while (here->data != target &&  here->link != NULL)
    here = here->link;

if (here->data == target)
    return here;
else
    return NULL;
```

Notice we check to see if “here” is currently pointing to the last item in the list by checking to see if the link to the next item is pointing to NULL.

Special Notes:

If the preliminary version of the code looks like:

```
here = head;

while (here->data != target &&  here->link != NULL)
    here = here->link;

if (here->data == target)
    return here;
else
    return NULL;
```

Will it work for an empty list?

Why/Why not?

Our final code looks like:

Function Declaration:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr head, int target);
//Precondition: The pointer head points to the head of
//a linked list. The pointer variable in the last node
//is NULL. If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that
//contains the target. If no node contains the target,
//the function returns NULL.
```

Our final code looks like:

Function Definition:

```
//Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
    {
        return NULL;                    Empty list case
    }
    else
    {
        while (here->data != target &&
                 here->link != NULL)
              here = here->link;

        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Pointers as Iterators (see Ch 18 for more details)

An iterator let you cycle through the data items stored in a data structure so that you can perform the desired action(s) on each data item

It can be an object of some iterator class.

If you think about arrays and linked lists, it can also be an array index or pointer

You can use a pointer/array index to move through a linked list/array one node at a time.

The general outline for an iterator in the context of a linked list is:

```
Node_Type *iter;
for (iter = head; iter != NULL; iter = iter->link)
    Do whatever you want with the node pointed to by iter;
```

What if you want to print out every piece of data in your linked list, how would you change this?

How would you construct iterator code for an array?

Insertion of nodes in an ordered list

```
void insert(NodePtr after_me, int the_number);
//Precondition: after_me points to a node in a linked list.
//Postcondition: A new node containing the_number
//has been added after the node pointed to by after_me.
```

If the list is ordered, you cannot simply insert data at the beginning/end.

This function assumes that a separate function has already identified where the new node is to be inserted into the list.

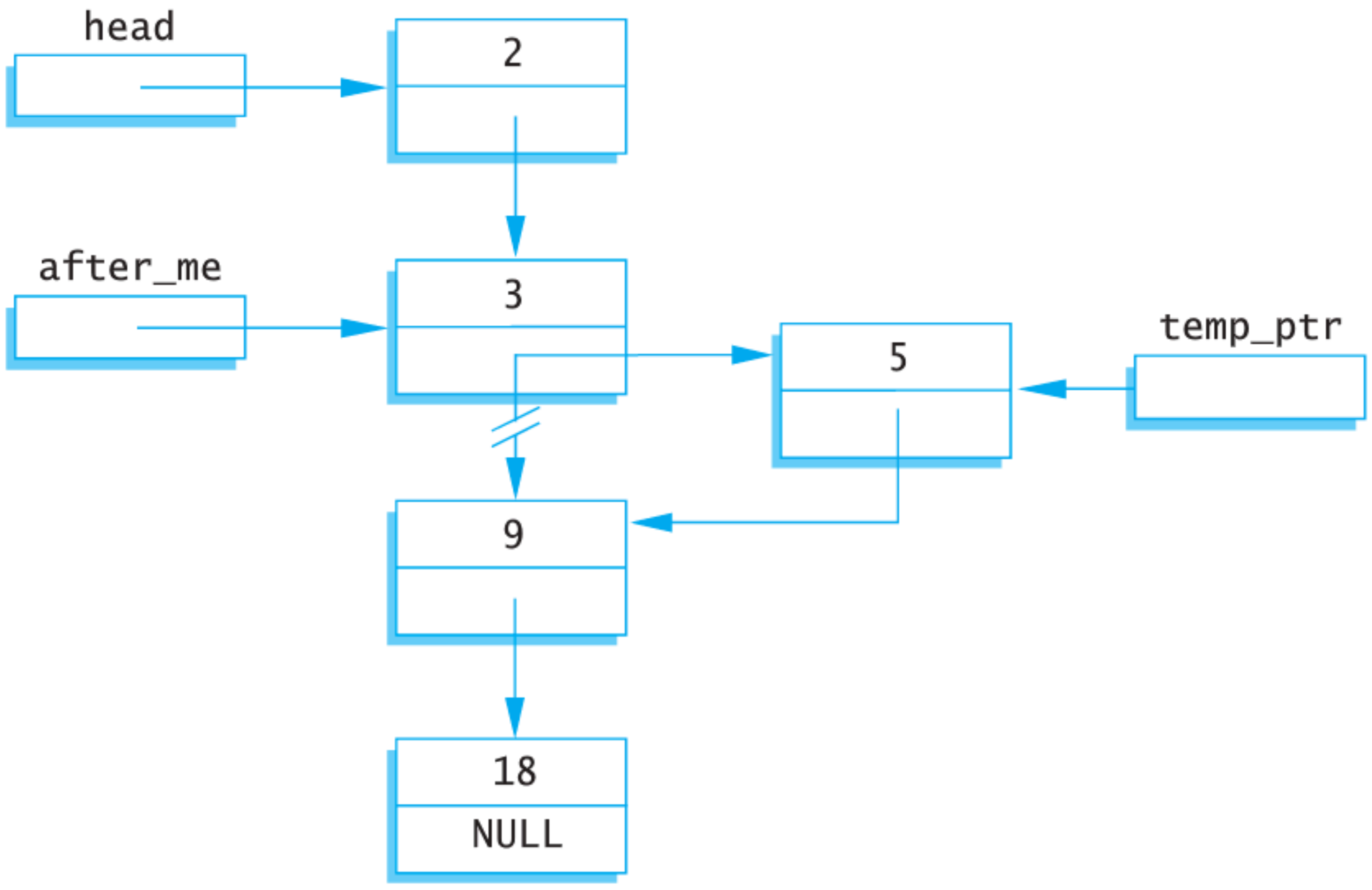
- It returns a pointer called `after_me` pointing to some node in the linked list.

Just like with the insertion at the head_insert function, we have to create a new node.

We then want to insert it after “after_me” with the new data, “the_number”.

```
void insert(NodePtr after_me, int the_number);
```

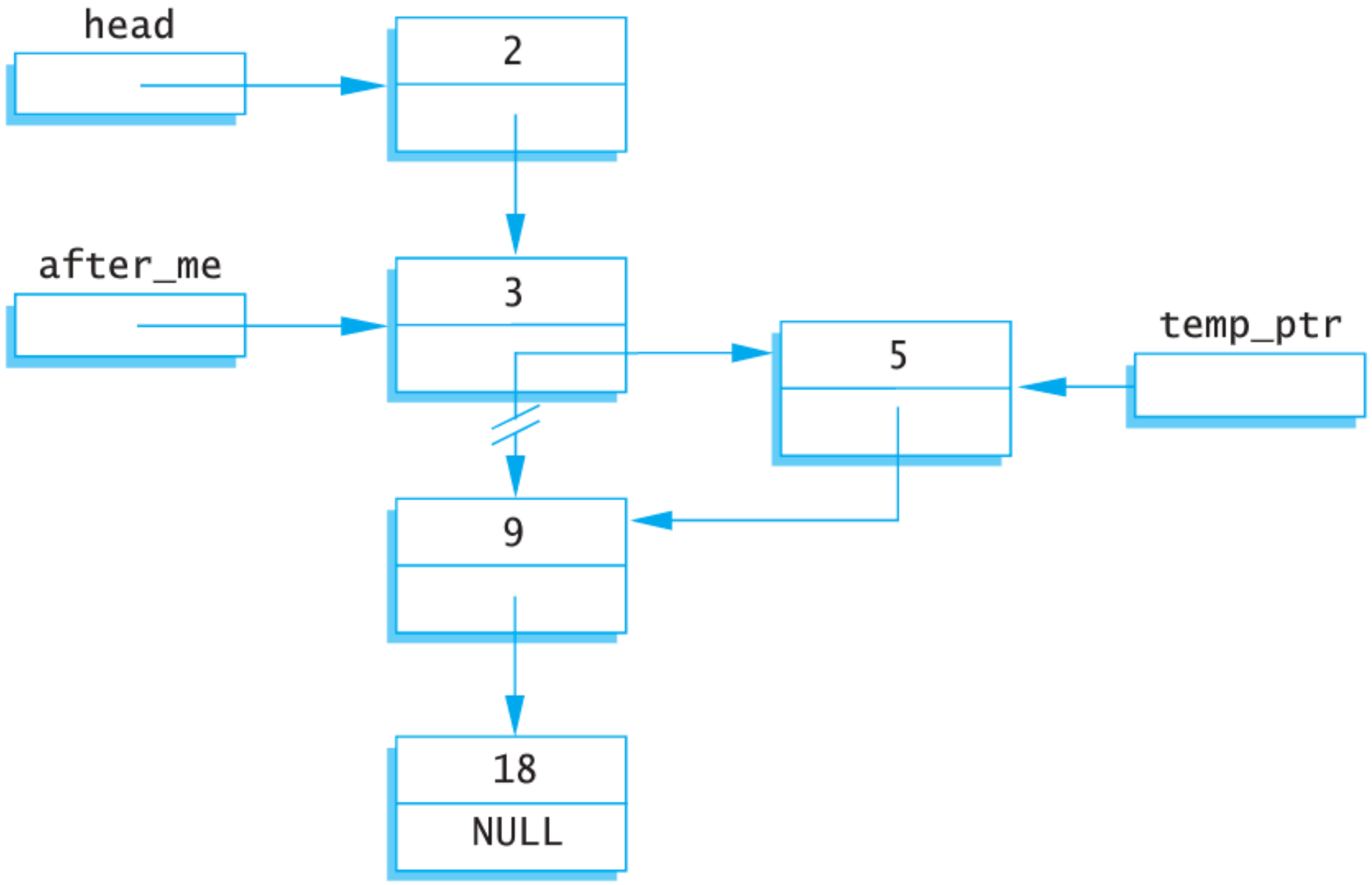
Insertion of nodes in an ordered list.



```
//add a link from the new node to the list:
temp_ptr->link = after_me->link;
//add a link from the list to the new node:
after_me->link = temp_ptr;
```

```
void insert(NodePtr after_me, int the_number);
```

Insertion of nodes in an ordered list.

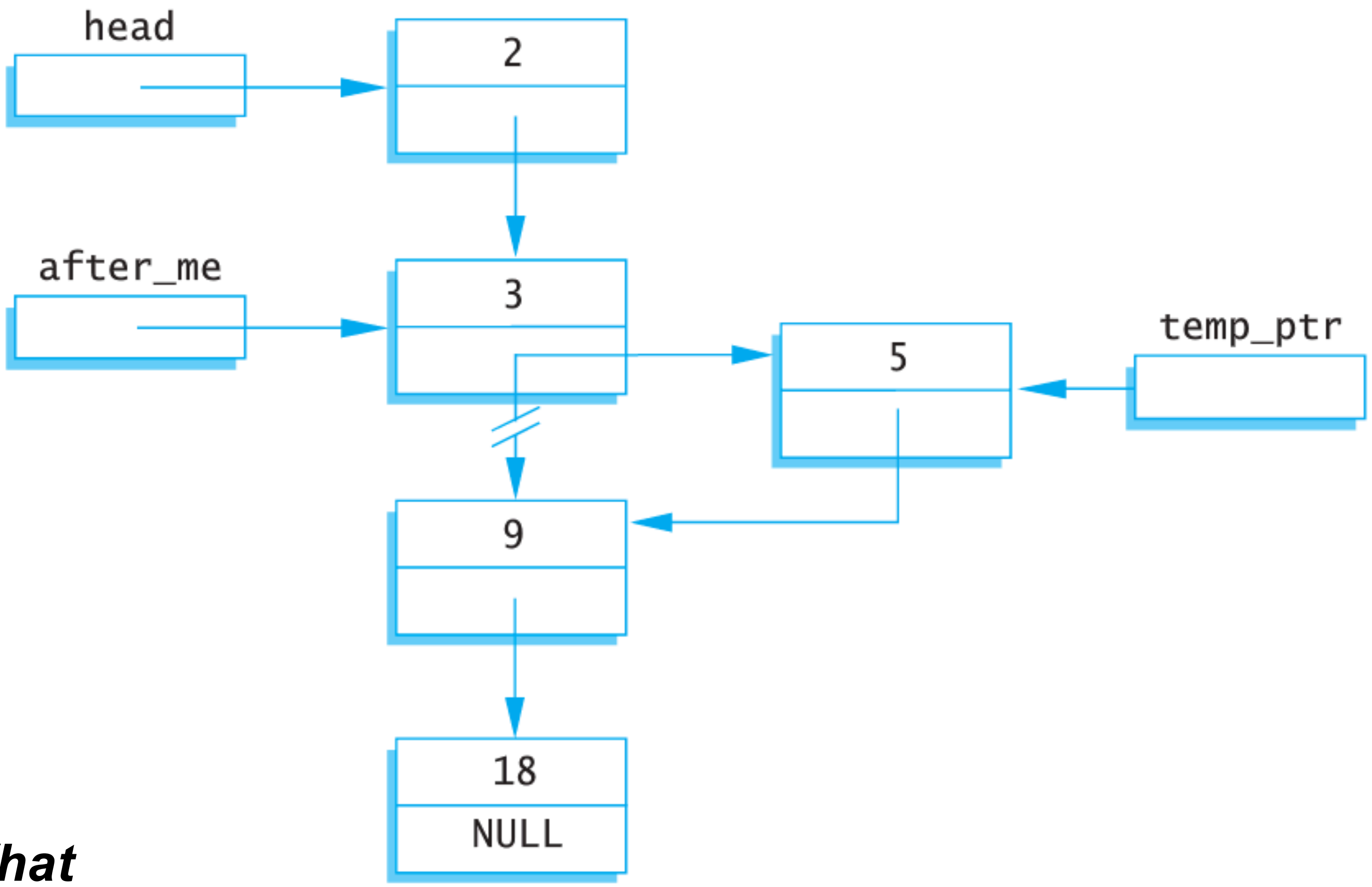


****WATCH THE ORDER*****

```
//add a link from the new node to the list:
temp_ptr->link = after_me->link;
//add a link from the list to the new node:
after_me->link = temp_ptr;
```

```
void insert(NodePtr after_me, int the_number);
```

Insertion of nodes in an ordered list.



****What happens if you reverse them?***

```
//add a link from the new node to the list:
temp_ptr->link = after_me->link;
//add a link from the list to the new node:
after_me->link = temp_ptr;
```

Function Declaration:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void insert(NodePtr after_me, int the_number);
//Precondition: after_me points to a node in a linked
//list.
//Postcondition: A new node containing the_number
//has been added after the node pointed to by after_me.
```

Function Definition:

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

Does this function work if after_me is the last node?

Does this function work if after_me is the fist node?

This function allows you to maintain an ordered list (numerical/alphabetical, etc.).

You are able to readjust the pointers to insert the node into the list, independent of length or position.

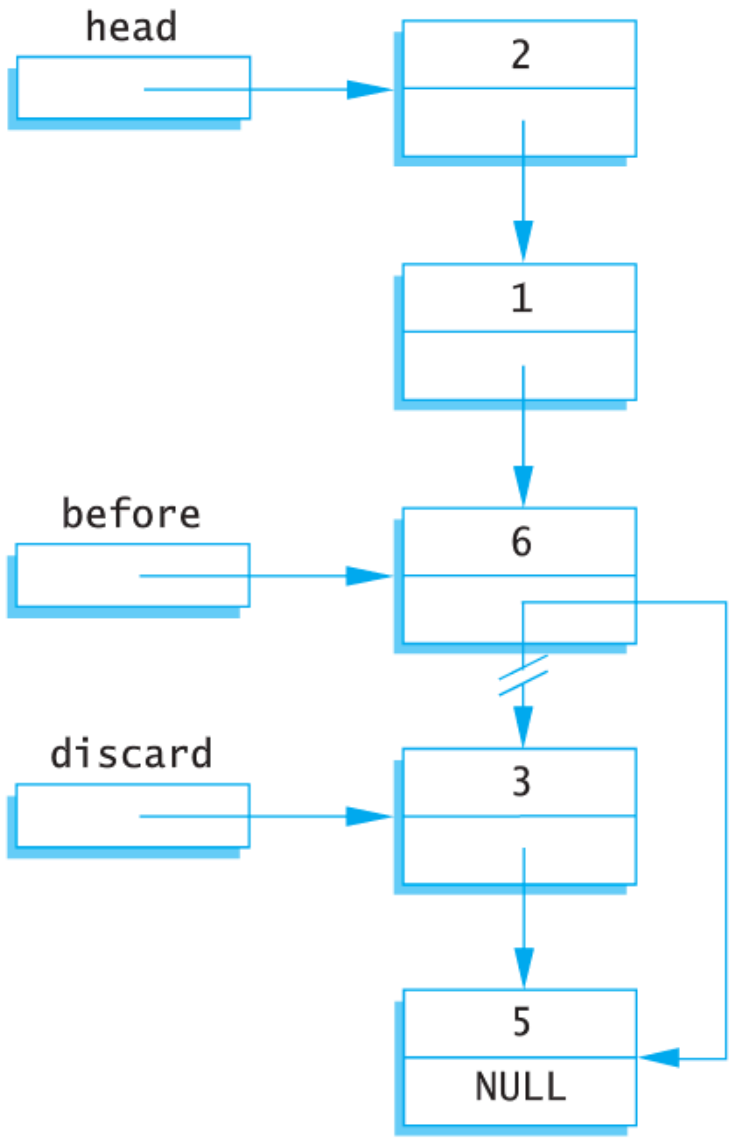
```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

If you tried to do this with an array, you would have to copy values to new locations to shift their ordering.

Despite that accessing data in an array is more efficient than a linked list, the ability to quickly insert and remove data into a linked list should help you to understand why some times they are a better choice of data structure than an array.

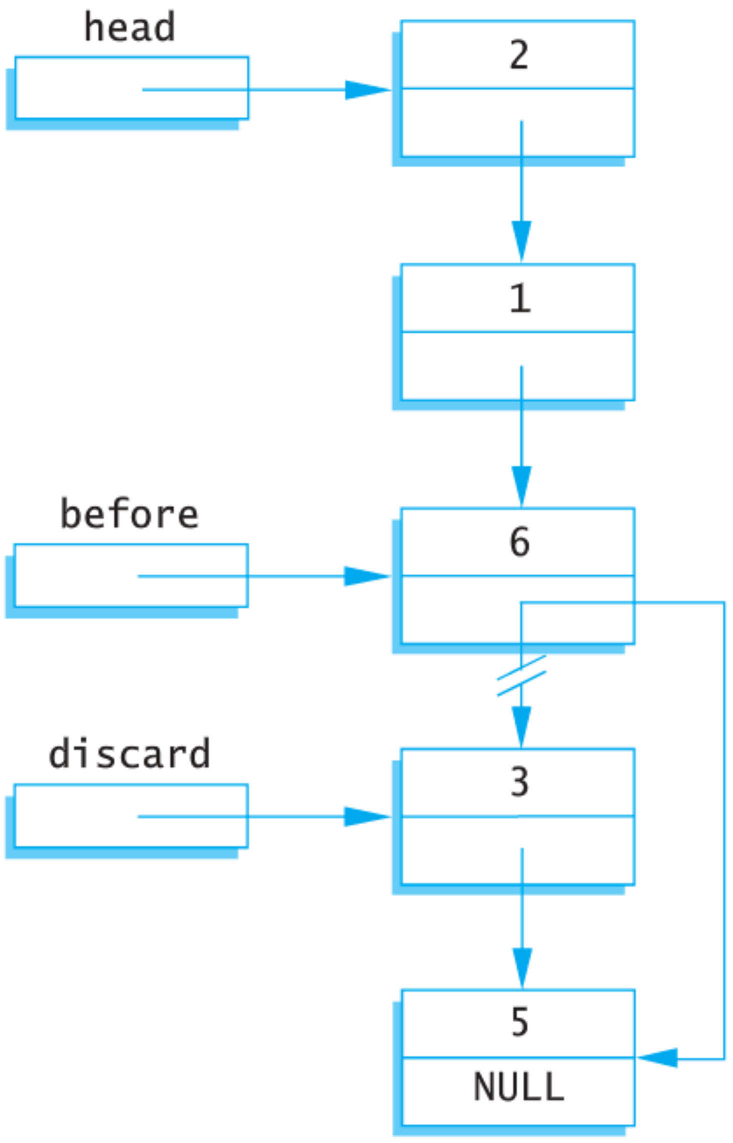
Next we'll look at removing a node from a linked list.

Removing a node from a linked list



1. Position the pointer `discard` so that it points to the node to be deleted and position the pointer `before` so that it points to the node before the one to be deleted.

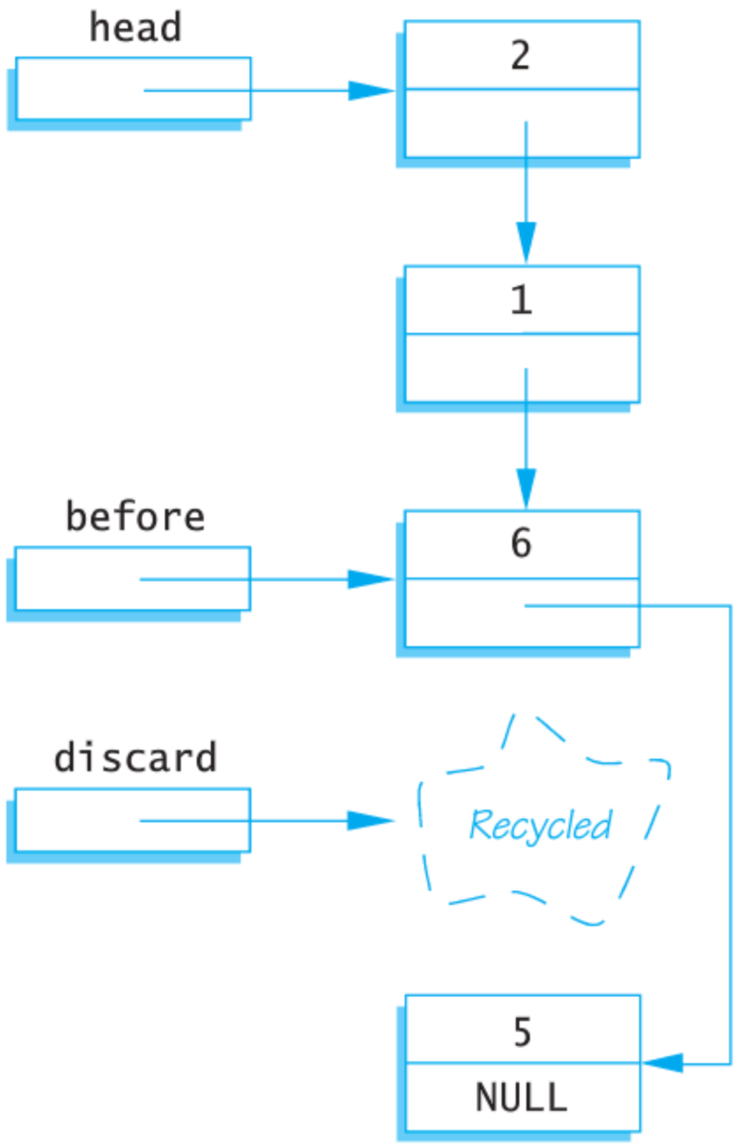
Removing a node from a linked list



2. Reposition the links in the list to remove the discard node and maintain connectivity.

```
before->link = discard->link;
```

Removing a node from a linked list



Then you simply have to destroy the discarded node so it returns to the freestore (unless you plan to use it for something else)

```
delete discard;
```

Removing a node from a linked list

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

So what do we need to do to the above “insert” function to discard a node from a linked list (Hint: the new function’s input parameter should be `before`).

What does the following statement do?

```
head2 = head1;
```

Will this gives you two copies of the same list (why/why not)?

What happens if you change the linked list pointed to by `head1`?

What happens if you delete the node pointed to by `head1`?

What data structures can you create with pointers/linked lists?

Up until now, we've been discussing a normal linked list.

- It only lets you move follow the links in one direction

However, a node in a doubly linked list has two links:

- One pointing to the next node
- One pointing to the previous node

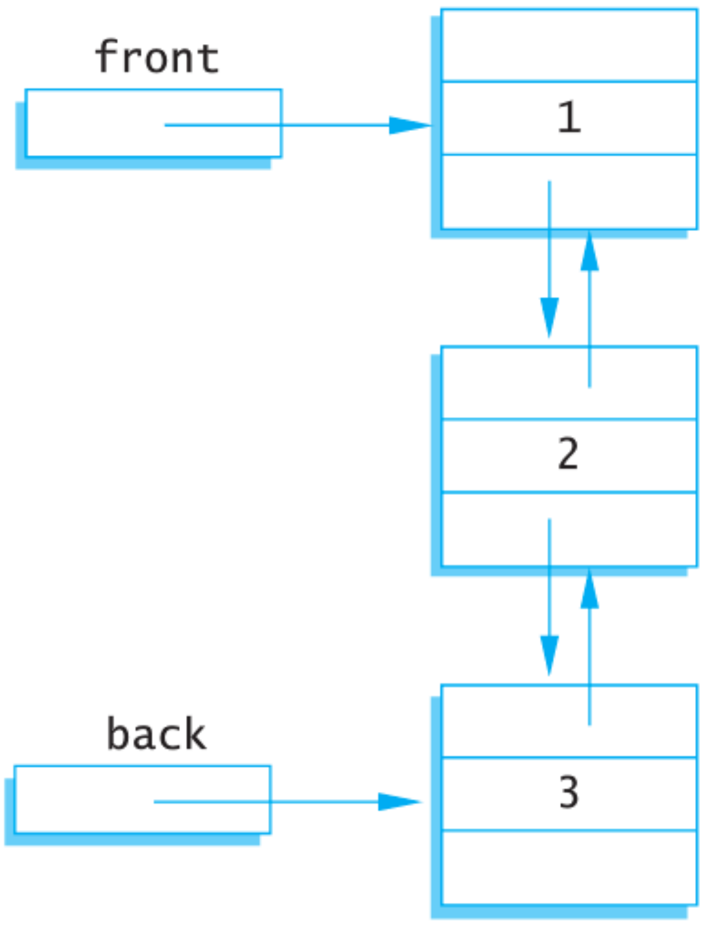
```
struct Node
{
    int data;
    Node *forward_link;
    Node *back_link;
};
```

Doubly linked lists

```
struct Node
{
      int data;
      Node *forward_link;
      Node *back_link;
};
```

Generally, you have a head and tail pointer

Also some of your functions for manipulating the list (constructors, etc), will have to change to account for the additional link (compared to a generic singly linked list).

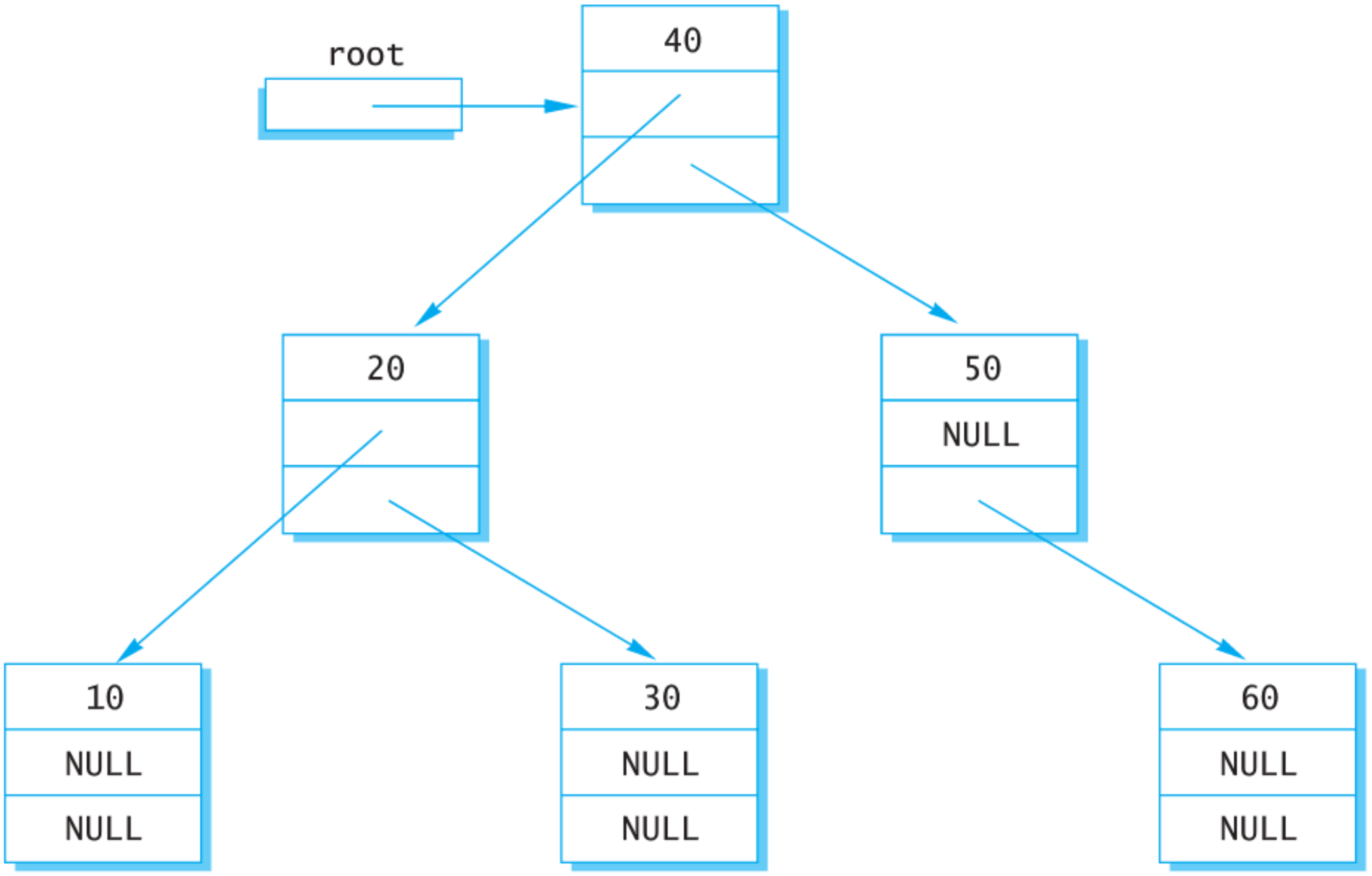


A binary tree

Each node has 2 children

There are no cycles in a tree.

```
struct TreeNode
{
      int data;
      TreeNode *left_link;
      TreeNode *right_link;
};
```



A binary tree

```
struct TreeNode
{
    int data;
    TreeNode *left_link;
    TreeNode *right_link;
};
```

You can reach any node from the top (root) node) by following the path of links.

There are no cycles in a tree.

If you follow these links, you eventually get to the end (the leaves)

It is called a binary tree because each node has two links that point to two children nodes (or the value NULL).

There are other kinds of trees with different numbers of links to different numbers of child nodes, but the binary tree is the common case.

A tree is not a form of a linked list, but its use of pointers is similar to how they are used in a linked list.

Because they are traversed along branches from root to leaf, the links to the child nodes generally have left and right in their name.

Linked Lists of Classes

In C, you would obviously create linked lists using structs.

However, in C++, you can also create them using classes.

- The logic is identical, you just have to use the syntax of a class instead of a struct.

The following class is for the same Node structure we were using before and includes a set of public accessor and mutator functions.

Node Class

Definition:

```
//This is the header file for Node.h. This is the interface for
//a node class that behaves similarly to the struct defined
//in Display 13.4
namespace linkedlistofclasses
{
    class Node
    {
    public:
        Node( );
        Node(int value, Node *next);
        //Constructors to initialize a node

        int getData( ) const;
        //Retrieve value for this node

        Node *getLink( ) const;
        //Retrieve next Node in the list

        void setData(int value);
        //Use to modify the value stored in the list

        void setLink(Node *next);
        //Use to change the reference to the next node

    private:
        int data;
        Node *link;
    };
    typedef Node* NodePtr;
} //linkedlistofclasses
//Node.h
```

Node Class

Implementation:

```
//This is the implementation file Node.cpp.
//It implements logic for the Node class. The interface
//file is in the header file Node.h
#include <iostream>
#include "Node.h"

namespace linkedlistofclasses
{
    Node::Node( ) : data(0), link(NULL)
    {
        //deliberately empty
    }

    Node::Node(int value, Node *next) : data(value), link(next)
    {
        //deliberately empty
    }

    //Accessor and Mutator methods follow

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

    Node* Node::getLink( ) const
    {
        return link;
    }
```

Node Class

Implementation

Continued:

```
                         …
    void Node::setData(int value)
    {
        data = value;
    }

    void Node::setLink(Node *next)
    {
        link = next;
    }
} //linkedlistofclasses
//Node.cpp
```

Program using Node Class:

```
//This program demonstrates the creation of a linked list
//using the Node class. Five nodes are created, output, then
//destroyed.

#include <iostream>
#include "Node.h"

using namespace std;
using namespace linkedlistofclasses;

//This function inserts a new node onto the head of the list
//and is a class-based version of the same function defined
//in Display 13.4.
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    //The constructor sets temp_ptr->link to head and
    //sets the data value to the_number
    temp_ptr = new Node(the_number, head);
    head = temp_ptr;
}
```

This function is logically identical to our other head_insert function with structures, except the constructor is used to initialize the data and next fields.

Program

using

Node Class

Continued:

```
int main()
{
    NodePtr head, tmp;

    //Create a list of nodes 4 -> 3 -> 2 -> 1 -> 0
    head = new Node(0, NULL);
    for (int i = 1; i < 5; i++)
    {
        head_insert(head, i);
    }
    //Iterate through the list and display each value
    tmp = head;
    while (tmp != NULL)
    {
        cout << tmp->getData() << endl;
        tmp = tmp->getLink();
    }
    //Delete all nodes in the list before exiting
    //the program.
    tmp = head;
    while (tmp != NULL)
    {
        NodePtr nodeToDelete = tmp;
        tmp = tmp->getLink();
        delete nodeToDelete;
    }
    return 0;
}
```

Two common data structures built using linked lists are:

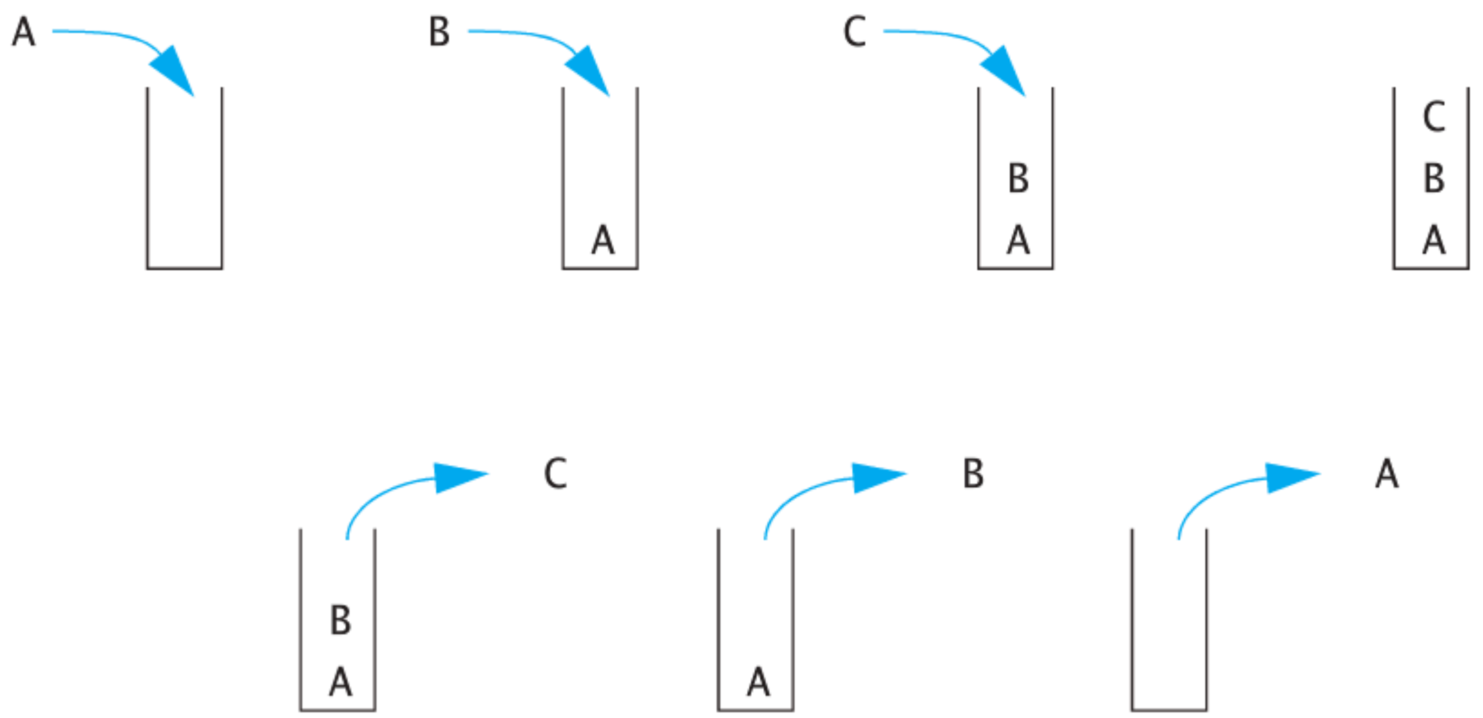
- stacks and
- queues.

Although you could implement them with a doubly linked list, their design only calls for a “normal” (singly) linked list.

Stacks

A Stack is a data structure that lets you retrieve data in the reverse order from which it is stored.

Hint: Think of a stack of books



If I store ‘A’ then ‘B’ then ‘C’ (as shown here), I have to retrieve them in reverse order: ‘C’ then ‘B’ then ‘A’.

Stacks

Stack are a LIFO data structure:

Last in/first out.

They are commonly used in many language programming tasks:

- Data is passed to and from a function call on the stack
- A key aspect of how recursive function calls work

Let’s see a simple example

Stack.h

```
//This is the header file stack.h. This is the interface for the class Stack,
//which is a class for a stack of symbols.
#ifndef STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

    typedef StackFrame* StackFramePtr;

    class Stack
    {
    public:
        Stack( );
        //Initializes the object to an empty stack.
        Stack(const Stack& a_stack);
        //Copy constructor.

        ~Stack( );
        //Destroys the stack and returns all the memory to the freestore.
```

(Constructors and Destructor)

Stack.h Continued:

```
            void push(char the_symbol);
            //Postcondition: the_symbol has been added to the stack.

            char pop( );
            //Precondition: The stack is not empty.
            //Returns the top symbol on the stack and removes that
            //top symbol from the stack.

            bool empty( ) const;
            //Returns true if the stack is empty. Returns false otherwise.
        private:
            StackFramePtr top;
        };
    }//stacksavitch

    #endif //STACK_H
```

Stacks have two fundamental operations:

- Push- adding an item to the stack
- Pop- removing an item from the stack
  - (Hint: Think of a Pez dispenser)

Example

Program:

```
//Program to demonstrate use of the Stack class.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main( )
{
   stack s;
   char next, ans;

   do
   {
       cout << "Enter a word: ";
       cin.get(next);
       while (next != '\n')
       {
           s.push(next);
           cin.get(next);
       }

       cout << "Written backward that is: ";
       while ( ! s.empty( ) )
       cout << s.pop( );
       cout << endl;

       cout << "Again?(y/n): ";
       cin >> ans;
       cin.ignore(10000, '\n');
   } while (ans != 'n' && ans != 'N');

   return 0;
}
```

(`cin.ignore` is discussed in Chapter 8; it discards input remaining on the current input line up to 10,0000 characters or until a return is entered. It also discards the return.)

Stacks

Remembering that Stacks are a LIFO data structure:

- All data is inserted at the top (head) of the linked list, and
- All data is removed from the top (head) of the linked list.

An empty stack is therefore an empty linked list (with top pointing to NULL)

So what does the implementation of the different functions in the Stack class look like:

(Warning: Remember you need a copy Constructor. Do you recall why?)

Stack Class Implementation:

```
//This is the implementation file stack.cpp.
//This is the implementation of the class Stack.
//The interface for the class Stack is in the header file stack.h.
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace stacksavitch
{
    //Uses cstddef:
    Stack::Stack( ) : top(NULL)
    {
        //Body intentionally empty.
    }

    Stack::Stack(const Stack& a_stack)

     <The definition of the copy constructor is Self-Test Exercise 11.>

    Stack::~Stack( )
    {
        char next;
        while (! empty( ))
            next = pop( ); //pop calls delete.
    }


    //Uses cstddef:
    bool Stack::empty( ) const
    {
        return (top == NULL);
    }
```

Stack Class

Implementation:

```
void Stack::push(char the_symbol)

    <The rest of the definition is Self-Test Exercise 10.>

//Uses iostream:
char Stack::pop( )
{
    if (empty( ))
    {
        cout << "Error: popping an empty stack.\n";
        exit(1);
    }
    char result = top->data;

    StackFramePtr temp_ptr;
    temp_ptr = top;
    top = top->link;

    delete temp_ptr;

    return result;
}
}//stacksavitch
```

Recommended practice: Based on what we've talked about, write the push function for this Stack class

Queues

Queues are a FIFO data structure; they handle data in a ***first in, first out*** manner:

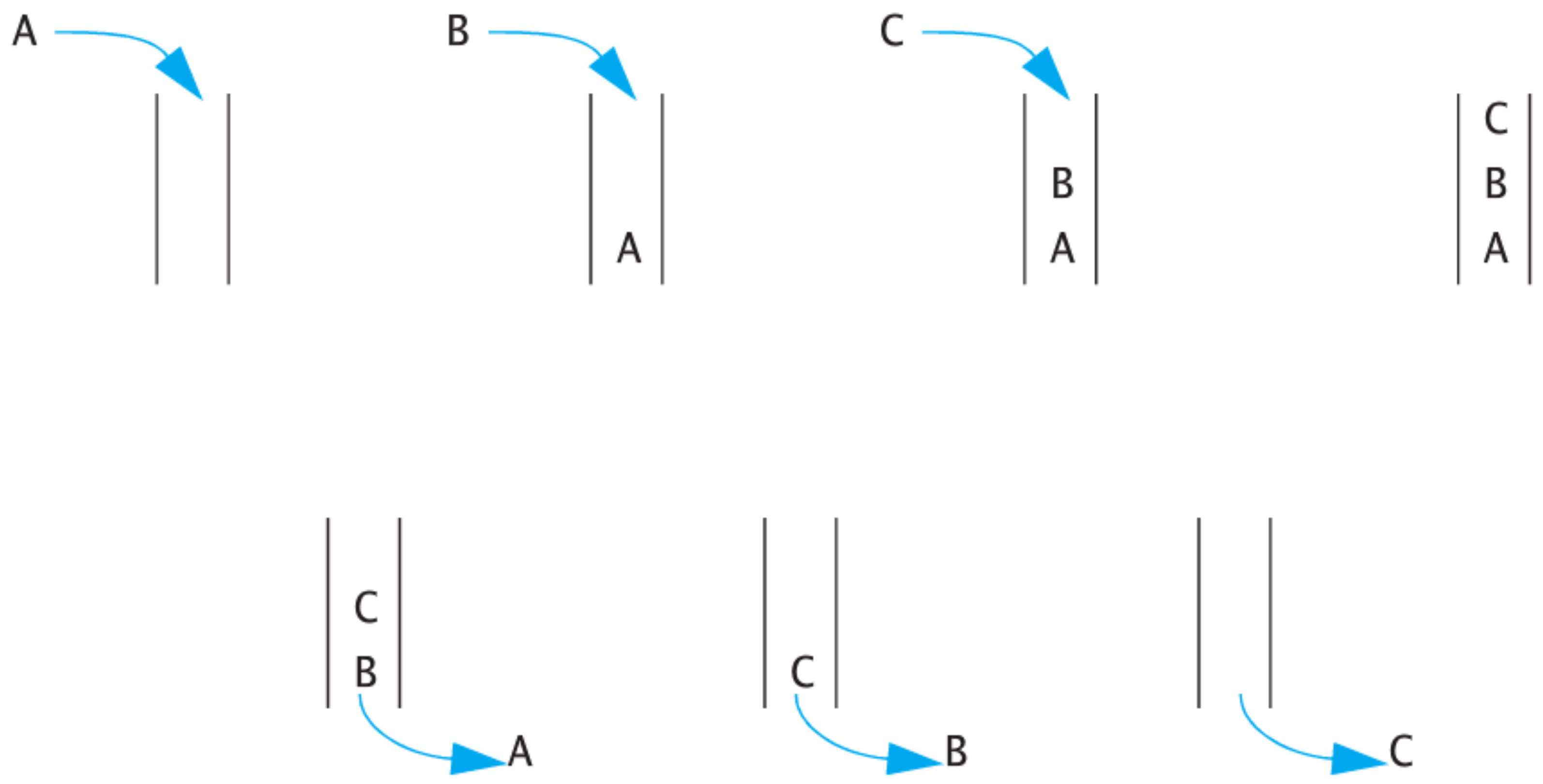
- It’s exactly the same as when you wait in a “line” (queue) for a cashier
- People are served in the order in which they arrive.

While a queue can be implemented using a singly linked list, it needs a head and a tail pointer:

- You can think of the head as the front of the line (queue) and the tail as the end of the line (queue)
- Without a tail pointer, you would have to traverse the entire list to add a new item to the list.
  - Significant run time overhead compared to storing one additional address variable

Queues

Queues are a FIFO data structure; they handle data in a ***first in, first out*** manner:



Let's look at how you implement this…

queue.h

```
//This is the header file queue.h. This is the interface for the class Queue,
//which is a class for a queue of symbols.
#ifndef QUEUE_H
#define QUEUE_H
namespace queuesavitch
{
    struct QueueNode
    {
        char data;
        QueueNode *link;
    };
    typedef QueueNode* QueueNodePtr;

    class Queue
    {
    public:
        Queue();
        //Initializes the object to an empty queue.
        Queue(const Queue& aQueue);
        ~Queue();
        void add(char item);
        //Postcondition: item has been added to the back of the queue.
        char remove();
        //Precondition: The queue is not empty.
        //Returns the item at the front of the queue and
        //removes that item from the queue.
        bool empty() const;
        //Returns true if the queue is empty. Returns false otherwise.
```

queue.h CONTINUED…

```
        private:
            QueueNodePtr front; //Points to the head of a linked list.
                                //Items are removed at the head
            QueueNodePtr back;  //Points to the node at the other end of the
                                //linked list. Items are added at this end.
        };
    }//queuesavitch
    #endif //QUEUE_H
```

This particular queue stores data of type char and has a “front” and “back” pointer instead of a “head” and “tail” pointer.

The two basic operations you can perform on a queue are:

- Add an item, and
- Remove an item.

Let’s see how you would use a queue in an application.

Remember:

<The `ignore` *member of* `cin` *is discussed in Chapter 8. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return (* `'\n'`*) at the end of the line.>*

Program using queue class:

```
//Program to demonstrate use of the Queue class.
#include <iostream>
#include "queue.h"
using namespace std;
using namespace queuesavitch;
```

(don’t forget your namespace)

main()

```
int main()
{
    Queue q;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:: ";
        while ( ! q.empty() )
            cout << q.remove();
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    } while (ans !='n' && ans != 'N');

    return 0;
}
```

If an empty queue is an empty linked list, what does this mean for your “empty” member function?

In other words, what should the values of front and back be if the queue is empty?

Let’s look…

Implementation of queue class:

```
//This is the implementation file queue.cpp.
//This is the implementation of the class Queue.
//The interface for the class Queue is in the header file queue.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using namespace std;

namespace queuesavitch
{
    //Uses cstddef:
    Queue::Queue() : front(NULL), back(NULL)
    {
        //Intentionally empty.
    }

    Queue::Queue(const Queue& aQueue)
            <The definition of the copy constructor is Self-Test Exercise 12.>
    Queue::~Queue()
            <The definition of the destructor is Self-Test Exercise 13.>

    //Uses cstddef:
    bool Queue::empty() const
    {
        return (back == NULL); //front == NULL would also work
    }
```

Implementation of queue class continued: (add function)

```
//Uses cstddef:
void Queue::add(char item)
{
    if (empty())
    {
        front = new QueueNode;
        front->data = item;
        front->link = NULL;
        back = front;
    }

    else
    {
        QueueNodePtr temp_ptr;
        temp_ptr = new QueueNode;
        temp_ptr->data = item;
        temp_ptr->link = NULL;
        back->link = temp_ptr;
        back = temp_ptr;
    }
}
```

Implementation of queue class continued: (remove function)

```
    //Uses cstdlib and iostream:
    char Queue::remove()
    {
        if (empty())
        {
            cout << "Error: Removing an item from an empty queue.\n";
            exit(1);
        }

        char result = front->data;

        QueueNodePtr discard;
        discard = front;
        front = front->link;
        if (front == NULL) //if you removed the last node
            back = NULL;

        delete discard;

        return result;
    }
}//queuesavitch
```

What would the implementation code for the copy constructor be?

What would the implementation code for the destructor be?

- Hint: recall the code for the stack’s destructor.

```
Queue::~Queue()
```

Summary:

A node is a struct/class object that has one or more member variables that are pointers.

- Nodes are connected to “adjacent” nodes by pointer their member variable “links” to their “neighbours”

A linked list is a list of nodes where each node points to the next node in the list.

- The end of the list is indicated by setting the pointer member variable to NULL/nullptr

Data structures built using linked lists include:

- Stacks (LIFO data structures)
- Queues (FIFO data structures)

Review Questions for Slide Set 6

- What are characteristics of linked lists?
- What are benefits of linked lists?
- What is the structure of a Linked List node? Why is it inherently circular? Why is this okay?
- What is the function of the keyword typedef?
- How do you dereference a pointer to a struct or an object (what are the two ways to dereference a dynamic struct or object)?
- What is the purpose of NULL/nullptr? What is the difference between the two?
- Do you need a using directive to reference NULL? Why or Why not?

Review Questions for Slide Set 6

- Can you use NULL with all pointer types? Why or Why not?
- Why is there an ambiguity problem (potentially) with NULL?
- Why do you need a head ptr for a linked list/stack/queue?
- Why do you need a tail pointer for a queue?
- What does LIFO mean? Give an example. When is it used?
- What does FIFO mean? Give an example. When is it used?
- Be able to explain the operation of and write the code for the basic functions of insert_head, insert_tail, remove_head, remove_tail, push, pop, search, remove.
- What test cases do your linked list functions need to be able to pass?

Review Questions for Slide Set 6

- What is an iterator?
- Why does order matter when altering the order of nodes in a linked list? For example, if you have

  temp_ptr-> link = current_node->link //followed by

  current_node->link = temp_ptr

  If you reverse the order of these instructions, will your inked list be the same?
- What is a required feature of a doubly linked list?
- What are the features of a binary tree?
- What are the differences if you want to create a Stack class if the node structure is defined in a class versus a struct?

Review Questions for Slide Set 6

- How would you define an empty queue, stack and linked list?
- Do these types of data structures (e.g. stack/queue/linked list) require copy constructors? If yes why?
- Do these types of data structures (e.g. stack/queue/linked list) require destructors? If yes why?
- Do these types of data structures (e.g. stack/queue/linked list) require custom assignment overloaded functions (as opposed to the default)? If yes why?
- What is the definition of a node?