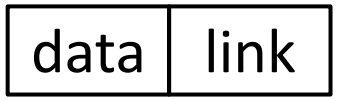
MAKING A SINGLY LINKED LIST

CSC 223

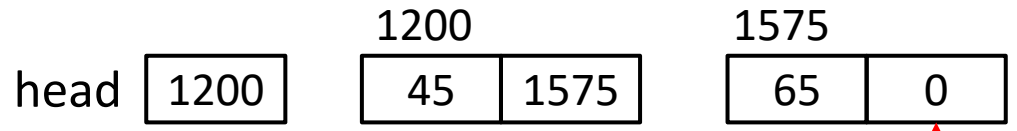
Linked Lists

- So far, we have been using an array to store the components of a list of items
- An array is a contiguous structure because the storage locations in the array must be in the one block of memory
- There are some problems using an array:
  - The array size is fixed
  - In an unsorted array, searching for an item is slow
  - In a sorted array, insertion and deletion is slow because it requires data movement
- An alternate data structure that can be used to hold list components is called a **linked list** which does not have the requirement that items be stored in contiguous memory locations
- A linked list is a collection of **nodes** containing two components:
  - The data items in the list
  - The address of the next node in the list (the **link**)

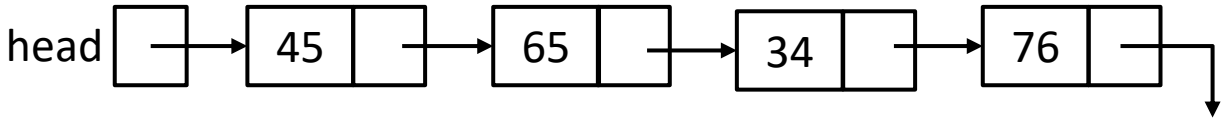


Node Connections

- An example of a linked list
  - link field for the first node in the list
  - The link field of the last node in the list is **`nullptr`**, which is a constant containing the value **`0`**



- Linked list nodes are typically created dynamically so the actual addresses are unknown
- A more common depiction of a linked list uses arrows instead of addresses
- A node is declared as a `class` or `struct` where the data type of the node depends on the specific application and the link component is a pointer variable

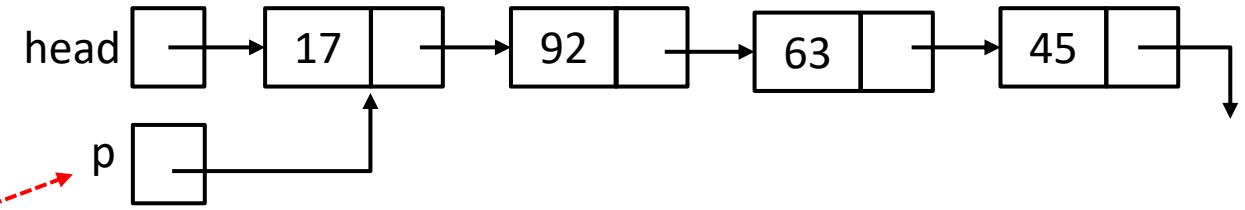



```
struct nodeType
{
      int info;
      nodeType *link;
};

int main()
{
      nodeType *head = new nodeType;
}
```

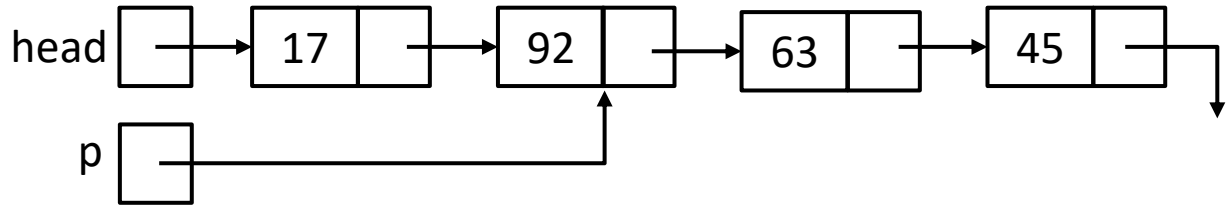
Declaration of a pointer to a node variable

Accessing Nodes

A separate pointer variable called a **cursor**, is often used to access items in a linked list (this keeps the head pointer fixed to ensure that the beginning of the list is not lost)



`current = head`
`current` = address of node with info 17
`current->info = 17`
`current->link` = address of node with info 92
`current->link->info = 92`



`current = head->current`
`current` = address of node with info 92
`current->info = 92`
`current->link` = address of node with info 63
`current->link->info = 63`

Singly Linked List ADT

- We want to build a class that will add, remove, and display items in a linked list
- We will make this a template class so the list can contain objects of any type
- The only instance variable required is a pointer to a node structure
    - The list is empty if the value of this variable is nullptr

The operations for this class are:

- Insert a node before the first node (this will build the list in a backward direction with the nodes being stored in the reverse order as they were entered)
- Insert a node after the last node (this will build the list in a forward direction with the nodes being stored in the same order as they were entered)
- Add a node after a node with the given info field
- Remove the node with a given info field
- Print the list in order
- Print the list in reverse order

Header file for `SinglyLinkedList` class

Start by defining a struct for the nodes:

```
template <class Type>
struct nodeType<Type>
{
   Type info;
   nodeType<Type> *link;
};
```

Here is the private section of the header file:

```
private:
    nodeType<Type> *head = nullptr;
```

Exception Classes

We also want to define a couple of exception classes which are encapsulated into the public part of the Singly Linked List class

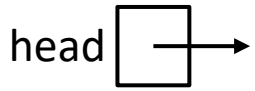
```
// Exception classes
  class EmptyListException
  {
    public:
      EmptyListException(){ message = "List
is empty."; }
      string what(){ return message; }
    private:
      string message;
};
```

```
class NodeNotFoundException
  {
    public:
      NodeNotFoundException(){ message =
"Node not found in list."; }
      string what(){ return message; }
    private:
      string message;
  };
```

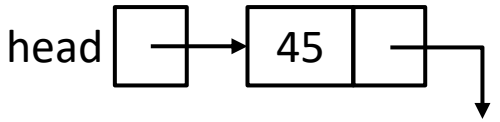
Building the List

```
SinglyLinkedList<int> myList;
```

- When a list object is instantiated in a client program, the list is initially empty

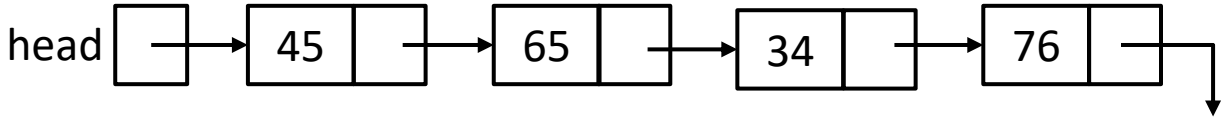


- After a node is added the list looks like this



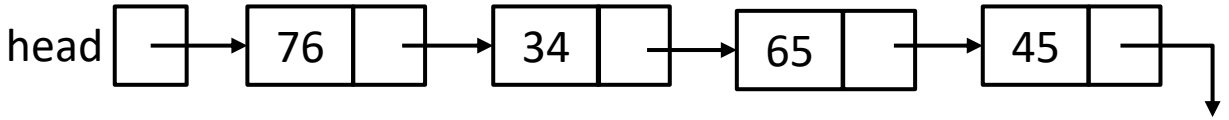
- From here, we can build the list by adding nodes to the end of the list or by adding nodes to the beginning of the list
- If we add nodes 45, 65, 34, 76 to the end of the list (build a forward list), we get this

Nodes are stored in the order they were added :



- If we add nodes 45, 65, 34, 76 to the beginning of the list (build a backward list), we get this

Nodes are stored in the reverse order from how they were added :



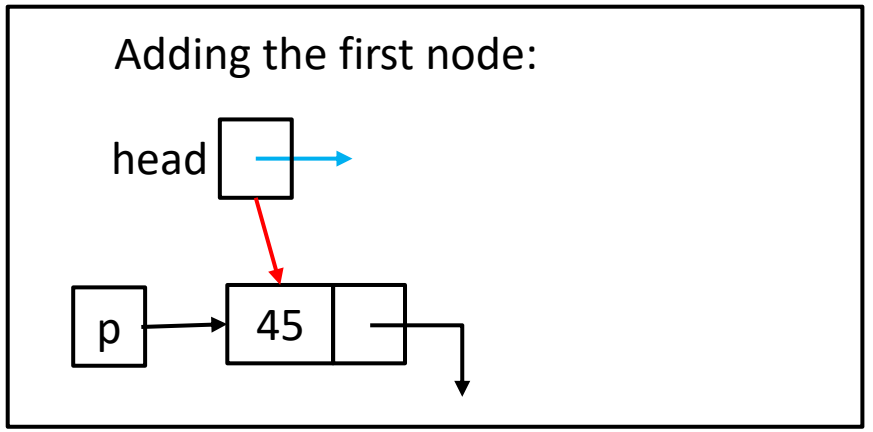
Build a Backward List

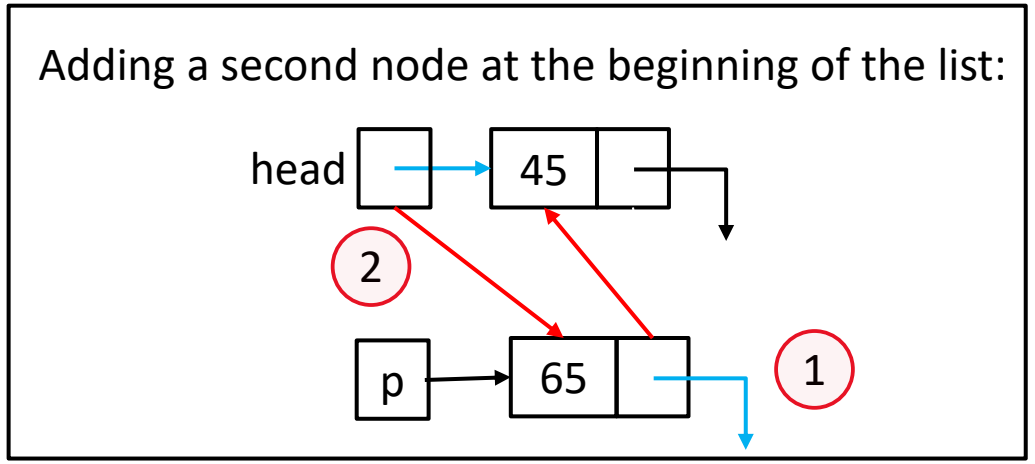
- It’s easier to build a backward list than a forward list
- Here is function prototype from the header
- Note the special case that occurs if the list is empty
- In either case, a node must be dynamically created

```
void buildBackward(Type value);
    // Precondition - The value of the info field is supplied
    // Postcondition - If list is empty, a new node is created
    // and head points to new node, otherwise, a new node is
    // created and is added to the beginning of the list
```

Code for `buildBackward`

```
void SinglyLinkedList<Type>::buildBackward(Type
value)
{
  nodeType *p = new nodeType;
  p->info = value;
  p->link = nullptr;

  if (head == nullptr)
  {
    head = p;
  }
  else
  {
    p->link = head;
    head = p;
  }
}
```





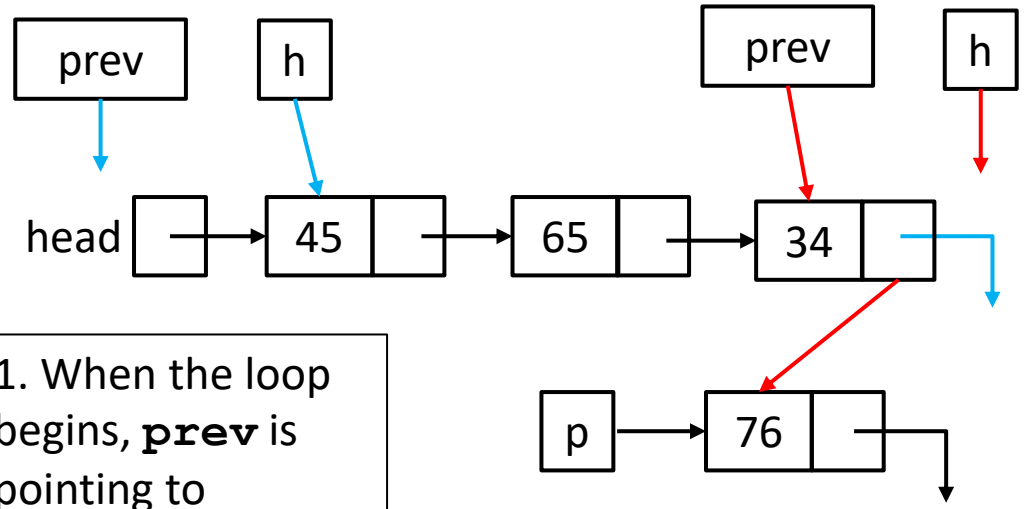
affected link
new link

Build A Forward List

- In a singly linked list, the link pointers go in one direction so a pointer to the last node in the list is not available
- The function will need extra code to find the pointer to the last node in the list
- The dynamic node creation and the check for first node in the list remain the same

```
void buildForward(Type value);
    // Precondition - the value of the info field is supplied
    // Postcondition - If list is empty, a new node is created
    // and head points to new node, otherwise, a new node is
    // created and is added to the end of the list
```

Code for **buildForward**



1. When the loop begins, **prev** is pointing to **nullptr** and **h** is pointing to **head**.

affected link
new link

2. When the loop ends, **prev->link** is pointing to the new node and **h** is **nullptr.**

```
void SinglyLinkedList<Type>::buildForward(Type
value)
{
  nodeType<Type> *p = new nodeType<Type>;
  p->info = value;
  p->link = nullptr;

  if (head == nullptr)
  {
    head = p;
  }
  else
  {
    nodeType<Type> *prev = nullptr;
    nodeType<<Type> *h = head;
    while (h != nullptr)
    {
      prev = h;
      h = h->link;
    }
    prev->link = p;
    }
}
```

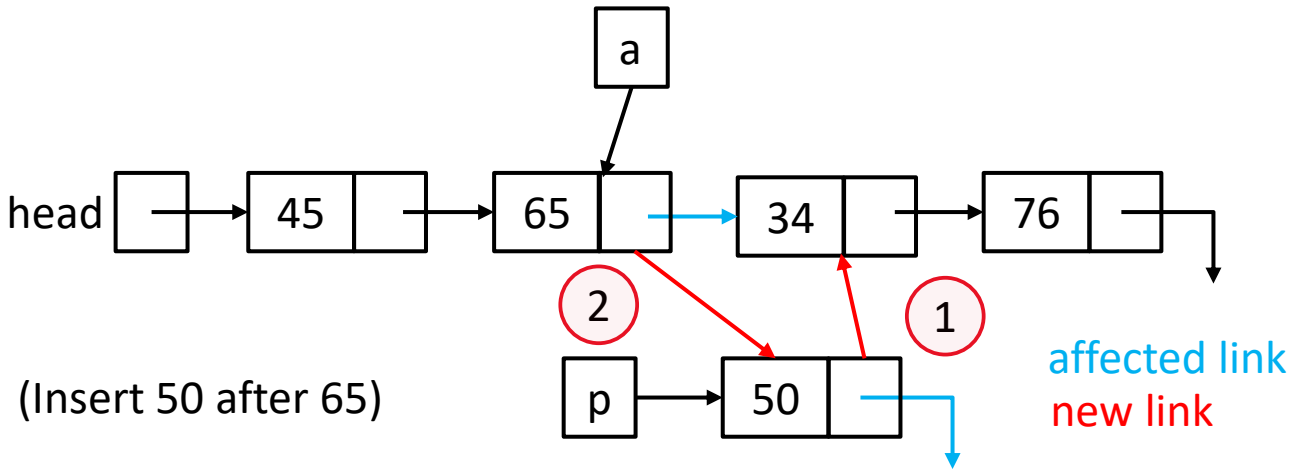
Inserting an Interior Node

- We can add a node to the beginning and end of the list, but what happens if we want to insert a node inside the list?
- The **`insertAfter`** function will find a node with the info field provided, create a new node with the value to be inserted, and then rearrange the pointers to link the node into the list after the specified node
- This function will throw an exception if the list is empty or if the specified node is not found in the list

```
void insertAfter(Type afterValue, Type newValue)
          throw (EmptyListException, NodeNotFoundException);
    // Precondition - The values of the node to insert after
    // and the new node are supplied
    // Postcondition - A new node is created and is added
    // to the list after the node with the value supplied
    // If the list is empty, or the supplied node is not found,
    // an exception is thrown
```

Code for `insertAfter`

- Can't insert into an empty list
- If the **a** pointer is null after the loop ends, the node with the specified info field was not found in the list
- Otherwise, **a** will point to the node to insert after



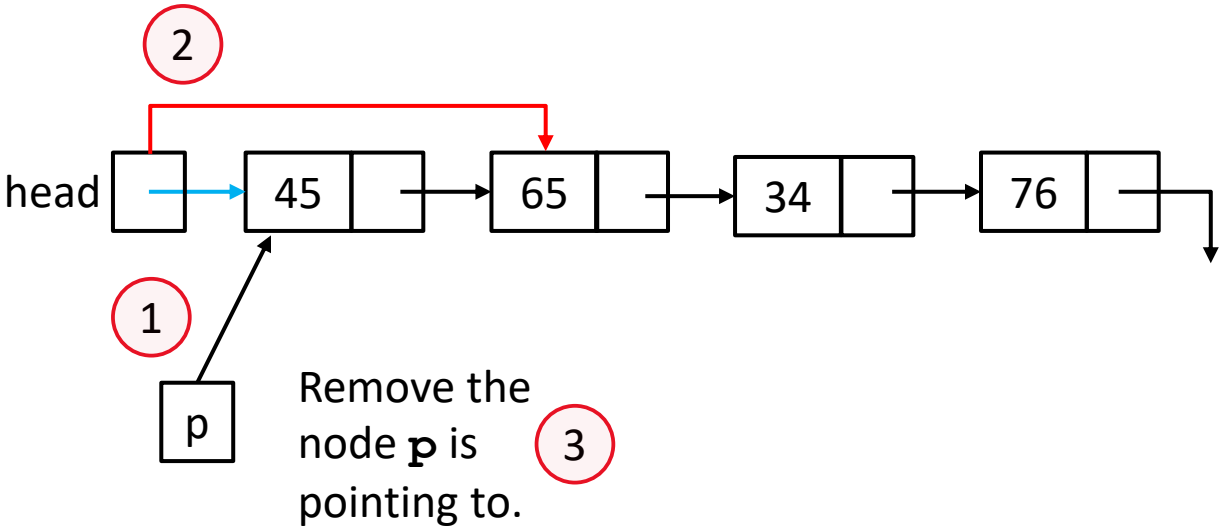

```
void
SinglyLinkedList<Type>::insertAfter(Type
afterValue, TypenewValue) throw
(EmptyListException,
NodeNotFoundException)
{
  nodeType<Type> *a;
  if (head == nullptr)
    throw EmptyListException();
  a = head;
  while (a != nullptr && a->info !=
afterValue)
    a = a->link;
  if (a == nullptr)
    throw NodeNotFoundException();
  else
  {
    nodeType *p;
    p = new nodeType<Type>;
    p->info = newValue;

    p->link = a->link;
    a->link = p;
  }
}
```

Removing a Node

- Removing a node from the list requires
  - That pointers to the node to be removed and the node before that node be found
  - The link pointers are rearranged to bypass the node
  - The delete operation is performed on the removed node to prevent memory leaks.
- The special cases of removing the first or last node must be considered
- Finally, the empty list and node not found exceptions are thrown if needed

```
void removeNode(Type value)
          throw (EmptyListException, NodeNotFoundException);
    // Precondition - The value of the node to be removed is supplied
    // Postcondition - The node is removed from the list
    // and the memory for that node is released
    // If the list is empty, or the supplied node is not found
    // an exception is thrown
```

Code for **removeNode** (part 1)



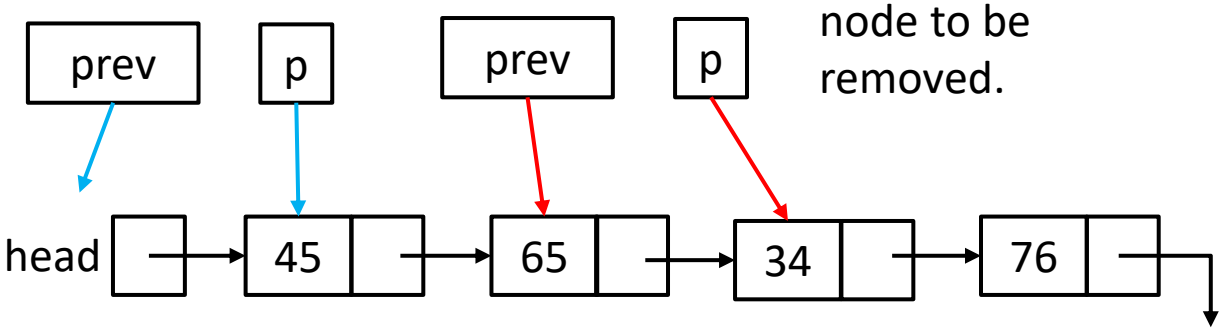
affected link
new link

```
void
SinglyLinkedList<Tpe>::removeNode(Type
value) throw (EmptyListException,
NodeNotFoundException)
{
  nodeType *prev, *p, *q;
  if (head == nullptr)
    throw EmptyListException();

  if (head->info == value)    //
Removing head node
  {
    p = head;
    head = head->link;
    delete p;
  }
  else      // Find node to remove
```

Code for **removeNode** (part 2)

```
else      // Find node to remove
  {
    prev = nullptr;
    p = head;
    while (p != nullptr && p->info != value)
    {
      if (p->link != nullptr)
        prev = p;
      p = p->link;
    }
    if (p == nullptr)      // Node not found,
throw exception
      throw NodeNotFoundException();
```

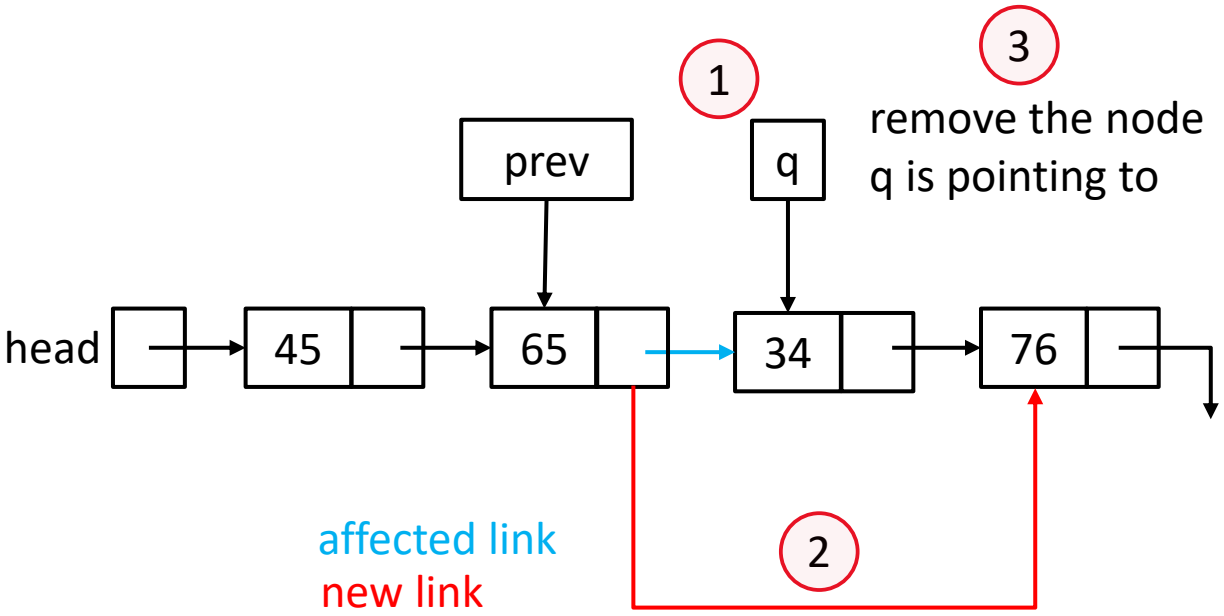
when the loop ends, **p** is pointing to the node to be removed.



affected link
new link

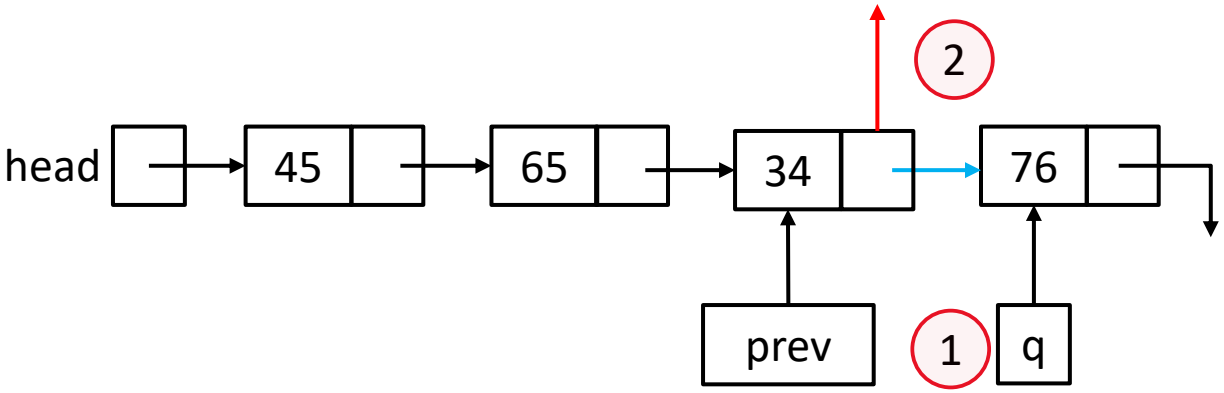
value = 34
(the node to be removed).

Code for **removeNode** (part 3)




```
else if (prev->link->link == nullptr)
// removing the last node
     {
       q = prev->link;
       prev->link = nullptr;
       delete q;
     }
     else  // Removing an interior node
     {
       q = prev->link;
       prev->link = prev->link->link;
       delete q;
     }
   }
}
```

Code for **`removeNode`** (part 4)



remove the node
q is pointing to 3

```
else if (prev->link->link == nullptr)
// removing the last node
    {
      q = prev->link;
      prev->link = nullptr;
      delete q;
    }
    else  // Removing an interior node
    {
      q = prev->link;
      prev->link = prev->link->link;
      delete q;
    }
  }
}
```

Printing the List

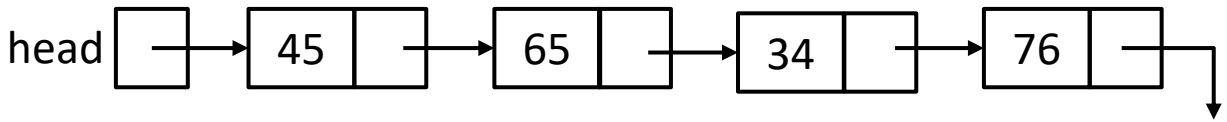
- To print the list, start at the beginning (head), visit each node, and print the info field, then get the link to the next node until the link is the **`nullptr`** signifying the end of the list
- The process of visiting each node in a list is sometimes called traversal as in "traversing a linked list"
- The function will throw an exception if you try to print an empty list

```
void print() throw (EmptyListException);
    // Postcondition - Displays the value of each
    // info field starting with the first node and
    // ending with the last node
    // If the list is empty, an exception is thrown
```

Code for **`print`**

```
void SinglyLinkedList<Type>::print()
throw (EmptyListException)
{
  if (head == nullptr)
    throw EmptyListException();

  nodeType<Type> *p = head;
  while (p != nullptr)
  {
    cout << p->info << " ";
    p = p->link;
  }
  cout << endl;
}
```

For this list, the output of print will be:
45 65 34 76



Traverse a Linked List

- The print function is a specific example of traversing a linked list.
- To **traverse** a list start with the pointer to the first node and step through the nodes of the list
- Use a cursor to preserve the pointer to the first node of the list:

You can replace this line with any processing required

```
current = first;
while (current != nullptr)
{
  cout << current->info << " ";
  current = current->link;
}
cout << endl;
}
```

Print the list Backwards

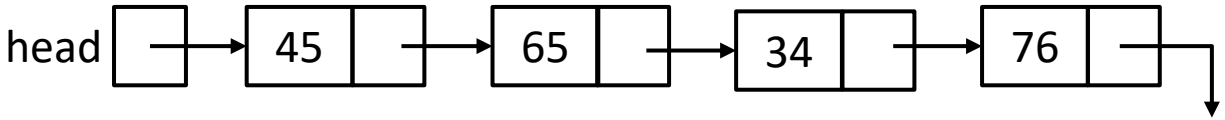
- A singly linked list has only forward pointers, so extra code is needed to print the list in reverse and recursion can be used to accomplish this task
- The recursion requires that the pointer to the next node in the list be provided and since the head pointer is private, two overloaded functions will be needed
- The first function is public, takes no parameters and is called by the client
- It calls the private function and starts the recursion by passing it the head pointer
- The public function throws an exception if it finds that the list is empty

Code for **`reversePrint`**

Public non-recursive function

Private recursive function

For this list, the output of the reverse print will be:
76 34 65 45




```
void SinglyLinkedList<Type>::reversePrint() throw
(EmptyListException)
{

  if (head == nullptr)
    throw EmptyListException();
  reversePrint(head);
}

void SinglyLinkedList<Type>::reversePrint(nodeType
*p)
{
  if (p != nullptr)
    {
     reversePrint(p->link);
     cout << p->info << " ";
    }
}
```