Lecture 4: Pointers, Dynamic Arrays, & Linked Lists

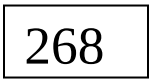
Read: Chpt.4, Carrano.

Recall that a variable must be stored in some location in memory. Hence, every variable in a program has two attributes:

Location: Address of memory (left-value)
Content of the location: Value (right-value)

Example:

```
int   v = 268;
```

v: 

To access a variable, we can use:

- Symbolic name
- Address of memory location (pointer)

Pointer & Pointer variables:

Pointer: Memory address of a variable.
Pointer Variable: Variable that can be used to store the actual memory address of a memory cell.

Q: How do we declare a pointer variable?

Statically Allocating Pointer Variable:
Syntax:

```
type_name    *pointer_name1, *pointer_name2;
```

Example:

```
int   *p;
```

Or,

```
int          v, *p1, *p2;
double   *q1, *q2, w;
```

Warning:

```
int*     p, q, r;
// p is of pointer type but q, r are of int type!
```

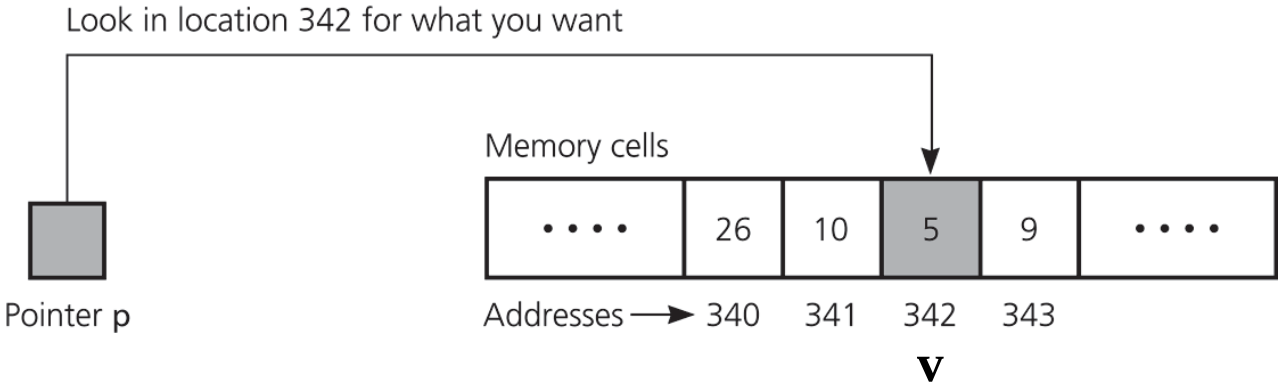
The “&” and “*” Operators:

The *address-of* operator, &:

When & is used in front of a variable v, &v corresponds to the address of the variable v.

Example:

```
int   *p, v;
v = 5;
p = &v;
// &v places the address of the variable v into the pointer
// variable p
```



The dereferencing (indirection) operator, *:

When * is used in front of a pointer variable p, *p corresponds to the memory cell referencing (pointing) by p.

Example:

```
int     v, *p1, *p2;
v = 168;
p1 = &v;
*p1 = 268;
cout << “v = ” << v << endl;
p2 = p1;
*p2 = 368;
cout << “v = ” << v << endl;
```

Output:

```
v = 268
v = 368
```

Three Roles of * and &:

	Binary	**Unary Operator**	**Unary Declaration Modifier**
*****	Multiply	Dereference	Declaring pointer
&	Bit-wise AND	Take address of	“what follows is a reference to”

Q: How do we initialize a pointer and/or make pointers point to things?

Pointer Initialization:

- Set pointer to NULL:

```
int     *p = NULL;
```

  Remark: Null is a special constant pointer value that can be assigned to a pointer variable of any type.

- Set pointer to the address of a variable of the correct type using the reference operator &:

```
int     v = 268;
int     *q = &v;
```

- Set pointer to contents of another pointer variable:

```
int     *p, *q;
int     v = 268;
q = &v;
p = q;
```

- Set pointer to point to dynamically allocated instance:

```
double   *p = new double;
int      *q = new int(268);
```

Dynamic Allocation:

In many languages, new ***unnamed*** instance of dynamic variable can be created during run-time.

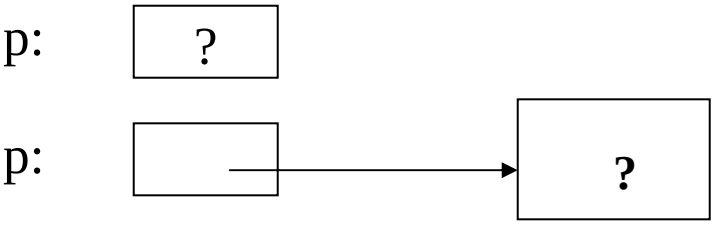
Using *new* Operator in C++:

Syntax:

```
type_name    * p;          // Define pointer variable p
p = new type_name;         // Create a dynamic variable of
                           // type_name pointed at by p
```

Or,

```
type_name    * p = new type_name;
```



Example:

```
int     *p,
p = new int;
*p = 560;
```

Or,

```
int   *p = new int;
*p = 560;
```

Or,

```
int     *p = new int(268);
```

Allocating & De-allocating Dynamic variables:

- Memory for dynamic variable must be allocated from ***heap***.
- Size of heap varies according to different languages & systems.
- In C++, dynamic variable is created using ***new***. If *new* is called and there is no more memory available in the heap, the program will end!

  Remark: Older compiler may return a pointer value NULL when no more new dynamic variable can be created.
- When no longer used, memory for dynamic variables should be freed and returned to the heap.
- Whenever you use dynamic allocation, you must have a plan to de-allocate the dynamic variables. In C++, this is accomplished by using the ***delete*** operator.

  Syntax: delete ptr;
- Once deleted, the dynamic variable will be destroyed. However, the pointer pointing to the dynamic variable remains intact and all pointers pointing to a delete dynamic variable are now undefined (dangling pointer)!

Example of Dangling Pointers:

```
int   *p, *q;
p = new int(268);
q = p;
delete p;              // Delete dynamic variable
cout << *q;            // What happens?
```

After delete p, the pointers p and q are pointers pointing to a de-allocated memory cell. They become ***dangling pointers***.

Memory Leak:

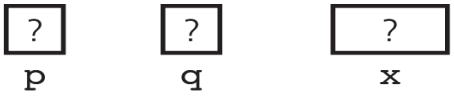
Declaring a pointer variable allows the variable to point to an instance of an object of the base type, but it does not actually create such an instance, nor does it set the pointer to point to one! However, if ***new*** is used, memory for the dynamic variable of the base type will be created.

Warning: Never write a piece of code like the following:

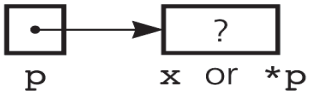
```
int     x = 27;
int*   p = new int;
p = &x;                        // Memory leak!
```

Memory leak describes the situation when a programmer allocates one or more variables ***dynamically*** but they become inaccessible because the programmer never explicitly deletes them.

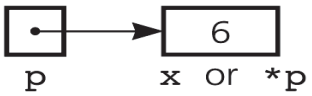
Example:

(a)
```
int *p, *q;
int      x;
```


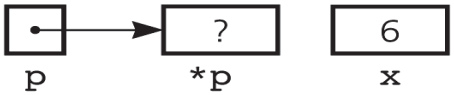
(b) `p = &x;`



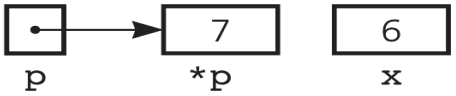
(c) `*p = 6;`



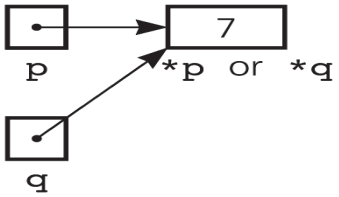
(d) `p = new int;`

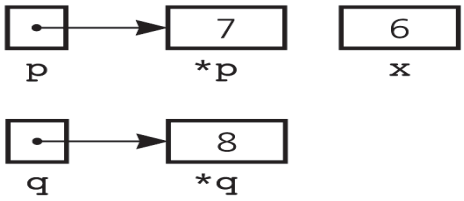


(e) `*p = 7;`

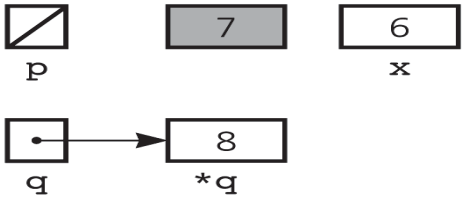


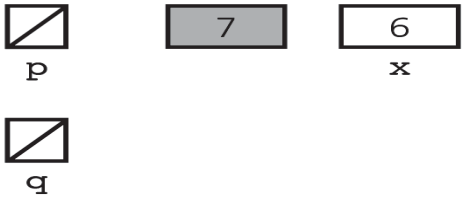
(f) `q = p;`



(g)
```
q = new int;
*q = 8;
```


(h) `p = NULL;`



(i)
```
delete q;
q = NULL;
```


Programming Strategy in Using Dynamic Variables:

- Whenever you use “new” to allocate a dynamic variable, have a plan for exactly where and when you will use “delete” on that variable.
- Do not delete a variable more than once!
- Do not apply “delete” to a pointer variable that is:
  - Currently pointing to a variable which was not allocated via “new”.
  - Currently set to NULL.
  - Whose value has not been initialized.
- Do not try to access a variable once it has been deleted.

Scope considerations:

- Recall *scope* describes the region of a program over which a particular variable is “alive” and visible.
- For local variables, once it is going out of scope, all memory used for local variables will be de-allocated.
- For dynamically allocated variables, they will remain allocated until they are explicitly de-allocated (deleted) via the *delete* operator.
- For pointers used in referencing dynamically allocated variables, they will be de-allocated and deleted once they go out of scope. However, the dynamic variables to which they point at will *not* be de-allocated, resulting in memory leaks.

Pointer can also be passed as parameter or returned as function value.

Passing Pointer as Parameter:

```
void  example1(int* p);
int   example2(int*& q, double r);
```

The * operator is associated with the type_name instead of the parameter in the function prototype.

Returning Pointer as Function Value:

```
double*   calculate(double a, double b)
{
   //  Return a pointer to a dynamically allocated
   //  variable holding the product a*b.
   //  Caller responsible for deleting the
   //  dynamically allocated memory!

   double*   p = new double(a*b);
   return p;
}
void aCaller()
{
   double*   p = calculate(3.7, -2.6);
   cout << “The answer is: ” << *p << endl;
   delete p;
}
```

Array and Pointer Variables:

In C++, an ***array variable*** is a ***(const) pointer variable*** pointing to the first indexed array variable.

Example: Using array and pointer variables.

```
#include <iostream>
using namespace std;
typedef  int*  IntPtr;
int main( )
{
   IntPtr  p;
   int       a[10];
   int       index;

   for (index = 0; index < 10; index++)
      a[index] = index;
   p = a;                    // Legal (but never assign a pointer to a)
   for (index = 0; index < 10; index++)
      cout << p[index] << " ";
   cout << endl;

   for (index = 0; index < 10; index++)
      p[index] = p[index] + 1;
   for (index = 0; index < 10; index++)
      cout << a[index] << " ";
   cout << endl;
   return 0;
}
```

Output:

```
0  1  2  3  4  5  6  7  8  9
1  2  3  4  5  6  7  8  9  10
```

Another Example:

```
   void func(double* p, int length);
      …
   void useArray()
   {
        double*    p;
        double     x[5];
        for (int i = 0 ; i < 5 ; i++)
             x[i] = 2.0 * i;
        p = x;
        p[3] = 1.3;
        func(x,5);
        …
   }
      …
   void func(double* p, int length)
   {
        for int i = 0 ; i < length ; i++)
             cout << p[i] << endl;
   }
```

Remark: *void func(double* p, int length)* is equivalent to *void func(double p[], int length)*.

Manipulating Addresses using Pointers:
Example:

```
int    A[5];
int    B[10];
int    *ptr1   =  A;      // ptr1 points at A[0]
int    *ptr2   =  B;      // ptr2 points at B[0]
int    *ptr3   =  A[0];   // ptr3 points at A[0]
int    *ptr4   =  B[4];   // ptr4 points at B[4]

if (ptr1 == ptr3)         // this is true
       …
if (ptr2 != ptr4)         // this is also true
       …
```

Q: How about <, <=, >, and >=?

They are defined only for pointers that point to the same array or the object that is in the memory immediately after the array.

Using Increment/Decrement Operators:

```
ptr1++;                   // ptr1 points at A[1]
ptr4––;                   // ptr4 points at B[3]
```

Using Addition/Subtraction Operators:

```
ptr2 = ptr2 + 3;          // ptr2 points at B[3]
ptr2 = ptr2 – 2;          // ptr2 now points at B[1]
```

Remark: No multiplication and division of pointers!

Dynamic Array:

An array allocated using new operator and whose array size can be determined during run-time.

Creating Dynamic Arrays using Pointer Type:

Syntax:

```
typedef  typeName*   yourTypeName;
yourTypeName         arrayExample;
arrayExample = new type_name[array_size];
```

Example:

```
int     array_size = 268;
typedef  double*   avePtr;
avePtr    aveArray;
aveArray = new double[array_size];
```

Or,

```
type    *p = new type[array_size];
```

Example:

```
int          array_size = 268;
double*   aveArray = new double[array_size];
```

Remark: This will create a dynamic array aveArray of size array_size with base type double. Array_size can be a constant or an expression (including a simple variable) whose value may not be known until runtime.

Destroying Dynamic Arrays using delete Operator:

Syntax:

```
delete [] arrayPtr;    // Do not use delete arrayPtr
```

Destructor:

Recall that a dynamic variable is accessible only through a pointer variable and the memory allocated for the dynamic variable is NOT released at the end of its block even though the memory allocated for a local pointer goes away. In C++, there are member functions, called ***destructor***, that are implicitly called when a class object passes out of scope.

- Destructor is specially designed to allow us to delete dynamically allocated simple variables, arrays, and class instances. It is sort of the “inverse” of a constructor.
- Unlike constructors, there is exactly one destructor in a class having prototype: ~className();
- If defined correctly, the destructor will do whatever clean-up the programmer intends, part of which is deleting dynamic memory allocated in by the object's constructors.
- As with constructors, C++ will automatically create a destructor (which does nothing) if you do not declare one yourself.
- A destructor is called exactly once when an instance is going out of scope (or otherwise being deleted).
- If in a function, you have a local variable that is an object with a destructor, when the function ends, the destructor will be called automatically.

Potential Problems in Using Array for ADT List:

- Array has fixed size
- Data must be shifted during insertions and deletions

Possible Solution:

Linked data structures

- Linked list is able to grow in size as needed
- Does not require the shifting of items during insertions and deletions

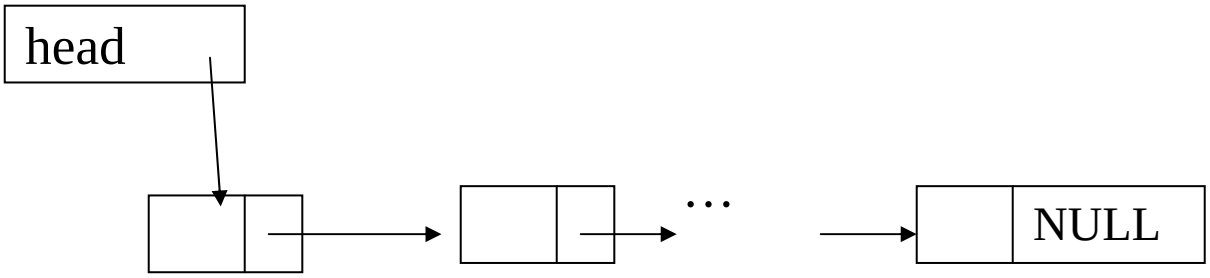
A ***linked list*** is a collection of objects being "linked" together with pointers such that

- Each element (node) of the linked list has some data and a pointer to the location *in memory* where the next element of the list can be found.
- The last element of the list will have a pointer value NULL.

Data Structure:

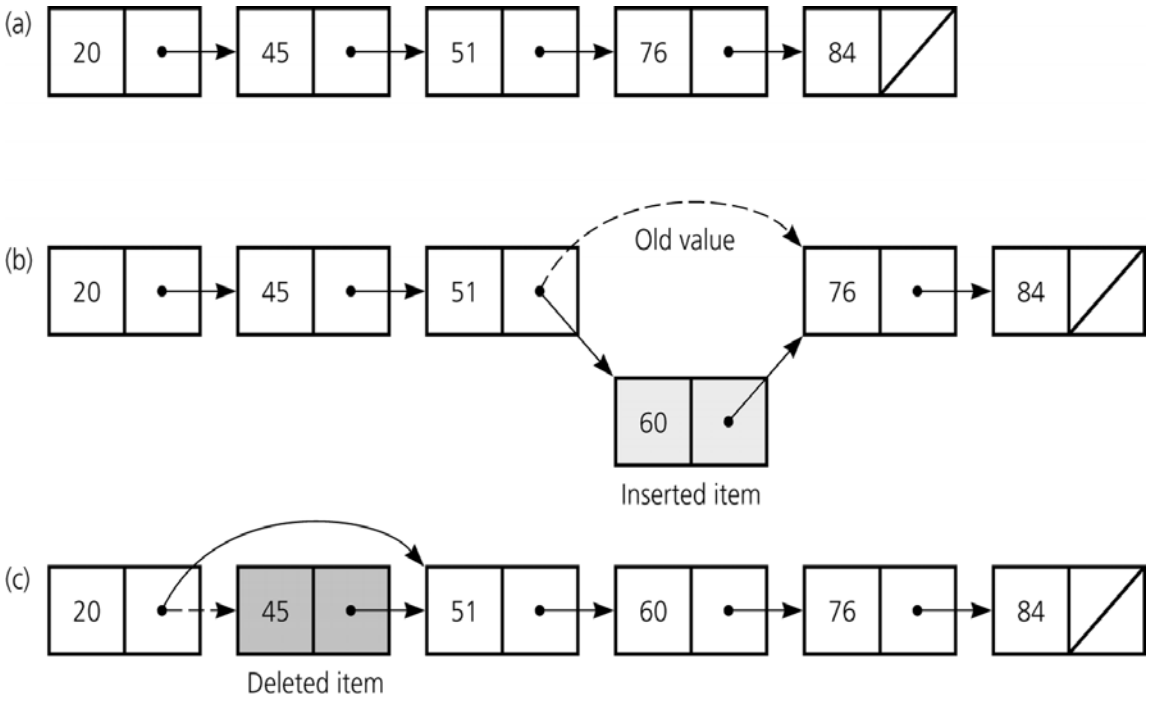
Node structure:

Linked list:



Empty List: head = NULL.

Simple Insertion and Deletion in Linked List:



C++ Implementation Strategy:

Recall that struct and class are essentially the same in C++ except their default accessibility.

- When we design **ADTs**, we will use ***class***.
- When we design **data structures**, we will usually use ***struct***.

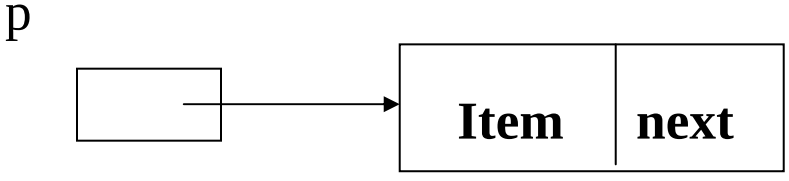
Hence, a node in a linked list is usually implemented using ***struct***.

Example: A linked list of integers.

```
struct Node
{
      int         item;
      Node*    next;
      Node(int val) : item(val), next(NULL)
      {
      }
};
```

Allocating a Node Dynamically:

```
      Node*   p;
      p = new Node;
```



Accessing Node Member:

(*p).item = 268;

Or using –> operator,

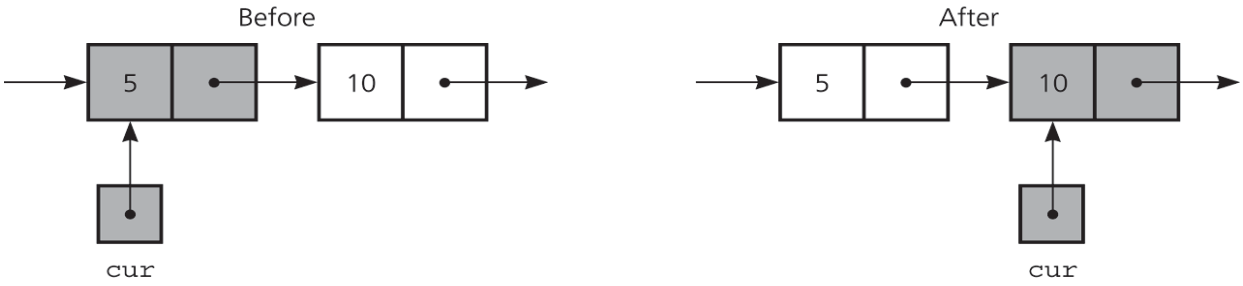
P –> item = 268;

Traversing a Linked List:

Use a pointer variable cur to keep track of the current node.

```
for (Node *cur = head; cur != NULL; cur = cur –> next)
    cout << cur –> Item << endl;
```

Example: Executing cur = cur –> next.



Example: Searching an Item in a List.

```
Node* search(Node* head, int target)
//Precondition: The pointer head points to the first object
//of a linked list. The pointer variable in the last node is
//NULL. If the list is empty, then head is NULL.
//Postcondition: Returns a pointer that points to the first
//node that contains the target. If no node contains
//the target, the function returns NULL.
{
   Node*   temptr = head;          // use temptr for traversal
   if (temptr == NULL)             // check for empty list
   {
      return NULL;
   }
   else
   {
     while (temptr –> item != target && temptr –> next !=
                  NULL)
           temptr = temptr –> next;    //     check next node

     if (temptr –> item == target) // target found
         return temptr;
      else
         return NULL;
   }
}  // end search
```

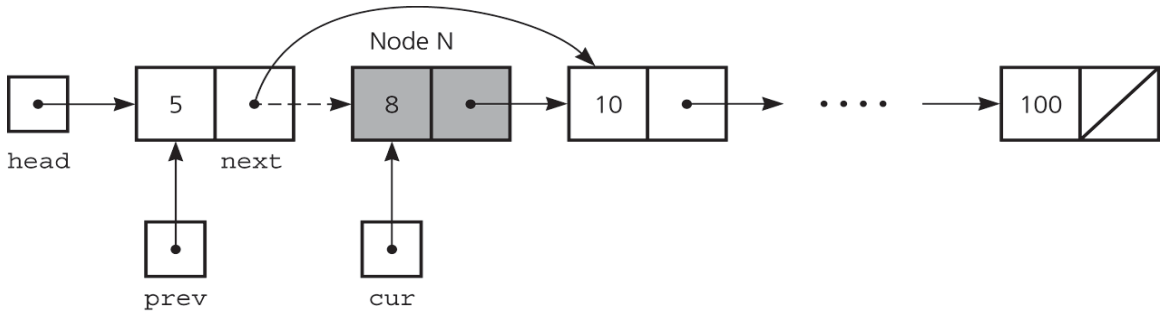
List Modification Algorithms:
Deleting a Node from a Linked (Ordinal) List:

- *Required function prototype*:
  bool remove(Node*& head, int pos);
- If a node at the indicated position exists, delete it & return TRUE; otherwise; return FALSE.

Remarks:

1. Generally NEVER want routines like this to print error messages and the like. Simply return some sort of status code, here a Boolean success code, suffices.
2. One must return the deleted node to system.

Approach: Use "prev" and "cur" pointers for deletion, where cur points at the node to be deleted.



Initially, prev = NULL, cur = head.

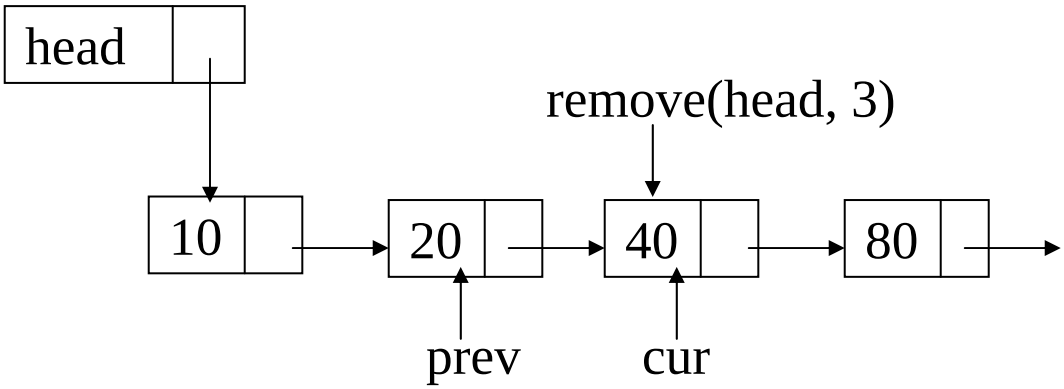
Special Cases:

1. Position to be deleted does not exist or list is empty.
2. Delete an interior node of list.
3. Delete first node of list.

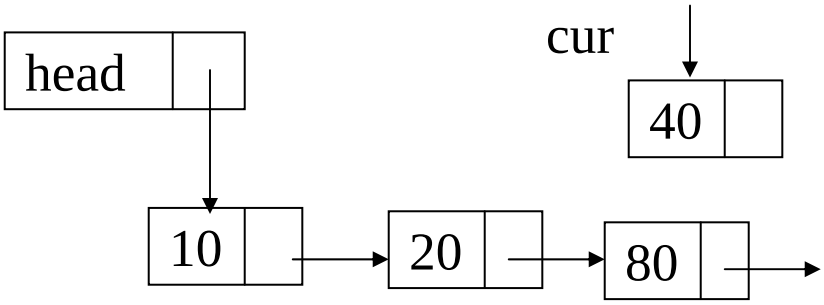
Algorithm:

Consider a “typical” list with some elements.

Before:



After:



prev –> next=cur –> next;

```
bool   remove(Node*&  head, int  pos)
{
    Node*   prev = NULL; // initialize prev and cur pointers
    Node*   cur = head;

    int   curPos = 1;           // start searching at pos = 1
    while ( (cur != NULL) && (curPos != pos) )
    {                           // searching for pos to delete
        prev = cur;
        cur = cur –> next;
        curPos++;
    }

    if (cur == NULL)        // no deletion
        return false;

    // cur points to node to be deleted.
    if (prev == NULL)       // delete first node
        head = cur –> next;
    else                        // delete an interior node
        prev –> next = cur –> next;

    delete   cur;               // return node to memory
    return   true;
}
```

Complication:

A general design principle is that functions should do only one thing! The function ***remove*** does *two* things:

(i) Find an element.

(ii) Delete an element.

Remedy:

Suppose we define a function, say ***Node* find(int pos)***, which would return a pointer pointing at the item (based on position) to be deleted. Function "remove" would then call "find", and delete the item found.

Slight Problem:

We want prev, not cur!

Solution:

In the case of ordinal (unsorted) lists, do

```
prev = find(pos–1);
cur = prev –> next;
```

We now need to check for special cases when

1. pos == 1.
2. pos == lengthOfList + 1.

Implementing *find* Operation:

```
Node*   find(Node*  head, int  pos)
{
   if (pos < 1)
      return   NULL;

   Node*   cur  = head;

   int   curPos = 1;

   while ( (cur != NULL) && (curPos != pos) )
   {
      cur = cur –> next;
      curPos++;
   }

   return   cur;
}
```

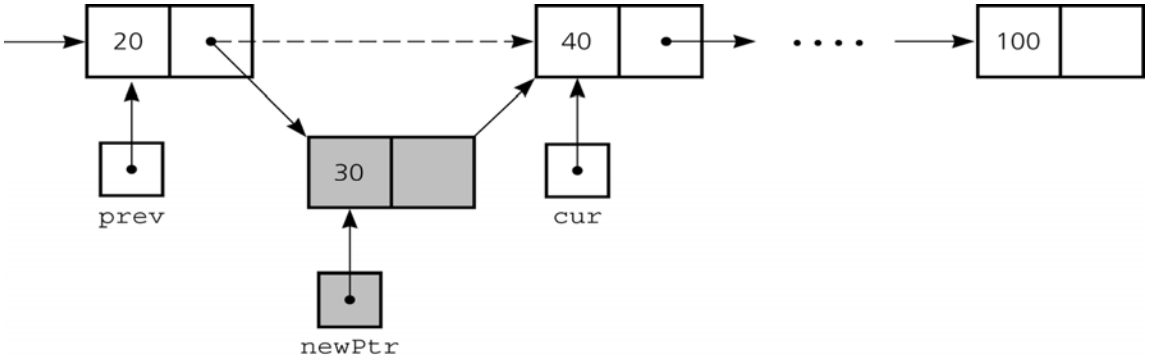
An Improved remove Operation:

```
bool   remove(Node*&  head, int  pos)
{
   // check for special cases, then use "find" if necessary
   if (head == NULL)
      return   false;

   if (pos == 1)                    // delete first item
   {
      Node*   p = head;
      head = head –> next;
      delete   p;
      return   true;
   }
   // find "prev" and "cur" as identified above
   // Use "find" to locate "prev"; then "cur" is the next one
   Node*   prev = find(pos–1);
   if (prev == NULL)           // position not found
      return   false;
   Node*   cur = prev –> next;
   if (cur == NULL)            // pos = lengthOfList + 1
      return   false;

   prev –> next = cur –> next;   //  delete node
   delete   cur;
   cur = NULL;
   return   true;
}
```

Inserting a Node into a Linked (Ordinal) List:

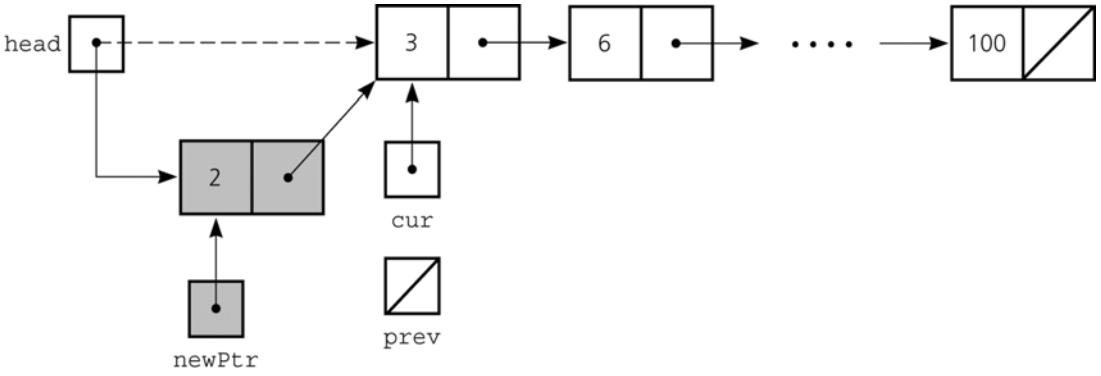
We use a ***prev*** pointer to point at the node preceding the new node after insertion.

General insert:




```
Node*   newPtr = new Node(item);
newPtr –> next = prev –> next;
prev –> next = newPtr;
```

Insert at the beginning of List:




```
Node*   newPtr = new Node(item);
newPtr –> next = head;
head = p;
```

```
bool   insert(Node*&  head, int  pos, int  item)
//preconditions:
//1. head is either NULL, or it points to the first node in a
//linked list
//2. pos identifies the ordinal position where a new node
//containing "item" is to be placed.
//postconditions:
//If it is possible to do so, a new node containing the "item"
//is created and linked into the appropriate spot in the list.
{
    if (pos == 1)                // insert at the beginning of list
    {
        Node*   newPtr = new Node(item);
        newPtr –> next = head;
        head = newPtr;
        return   true;
    }

    Node*   prev = find(pos-1);        // find prev for insert
    if (prev == NULL)                  // position not found
        return   false;

    Node*   newPtr = new  Node(item);  // create new node
    newPtr –> next = prev –> next;
    prev –> next = newPtr;
    return   true;
}
```

Packaging Issues:

"Node" as we defined it above is unsuitable for a variety of reasons

- The application data (namely the "int item" in struct Node) is intermixed with list implementation data (namely "next")
- Methods such as remove, insert, and the like should be methods of the ***list class***, not an individual ***list element***.

Consider:

```
Node* p = ...;
```

Q: Is p a pointer to a list of nodes, or a pointer to an individual node, or a pointer to application data?

The problem is that so far we have combined three separate abstractions together!

- The application data
- List elements
- The List itself

Three Explicit Classes:
Data Class:

```
class Data
{
	public:
		Data();
		Data(const Data& d);
		Data(…);
		...
	private:
	… (in our case, this would be the single integer, but…)
};
```

The other two required classes are the List class and the ListNode class with the latter class being defined as a private struct within the former.

Pointer-Based Implementation of ADT List:

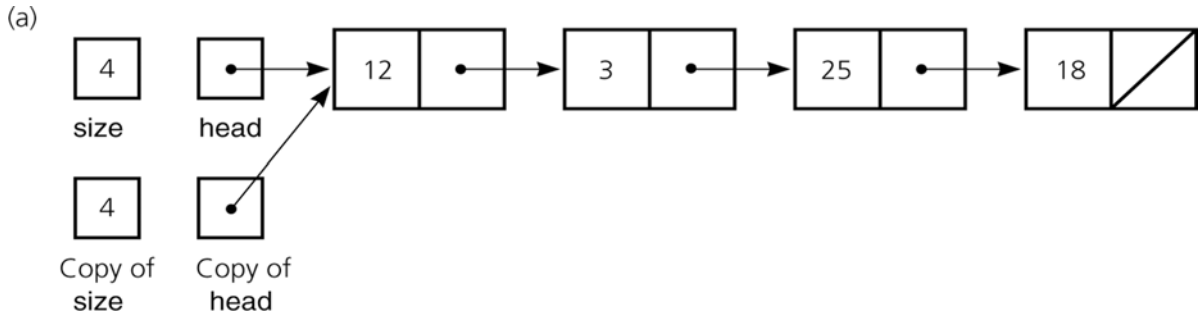
```
// Header file ListP.h for the ADT list using pointer

#include "ListException.h"
#include "ListIndexOutOfRangeException.h"

typedef   Data  ListItemType;

class  List
{
public:
// List();                              // default constructor
   List(const List& aList);           // copy constructor
   ~List();                           // destructor

// standard list operations:
   bool   isEmpty() const;
   int      getLength() const;
   void   insert(int index, ListItemType newItem)
          throw(ListIndexOutOfRangeException, ListException);
   void   remove(int index)
          throw(ListIndexOutOfRangeException);
   void   retrieve(int index, ListItemType& dataItem) const
          throw(ListIndexOutOfRangeException);
```

```
private:
  // Note private data structure for the nodes
  struct   ListNode                  // a node on the list
  {
    ListItemType   item;             // a data item on the list
    ListNode   *next;                // pointer to next node
  }; // end struct

  int   size;                        // number of items in list

  ListNode   *head;                  // pointer to list of items


  ListNode   *find(int  index) const;
  // Returns a pointer to the index-th node in the linked list

}; // end class

// End of header file.
```

Default Constructor:

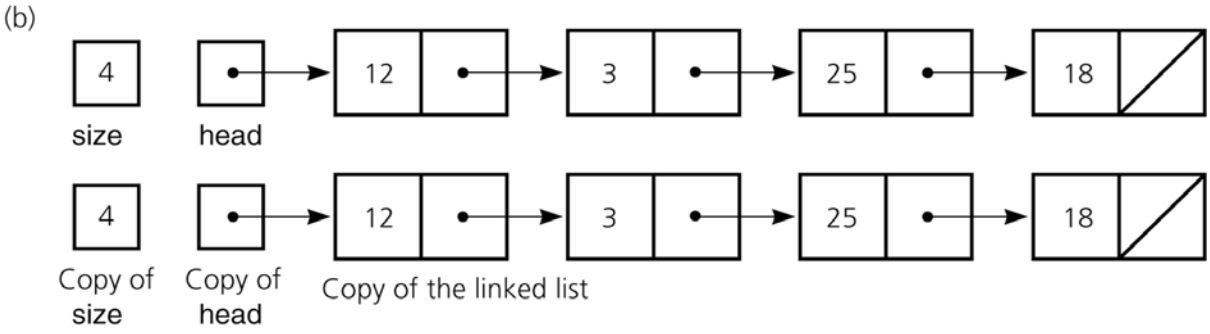
```
List::List(): size(0), head(NULL)
{
}
```

Copy Constructor:
Shallow Copy:



Only the data members (head, size) will be copied.

Deep Copy:



The whole list is copied; must implement copy constructor.

```
List::List(const List& aList): size(aList.size)
{
   if (aList.head ==NULL)        // empty list
      head = NULL;
   else
   {
      head = new ListNode;       // get first node
      assert(head != NULL);      // check memory allocation
      head –> item = aList.head –> item;
                                 // copy first item of old list
      // copy rest of old list
      ListNode *newPtr = head;
      for (ListNode *origPtr = aList.head –> next;
            origPtr != NULL; origPtr = origPtr –> next)
      {
         newPtr –> next = new ListNode;
         assert(newPtr –> next != NULL);
         newPtr = newPtr –> next;
         newPtr –> item = origPtr –> item;
      }

      newPtr –> next = NULL;
   } // endif
} // end copy constructor
```

assert Statement:

Must use

#include <cassert>

Syntax:

assert(boolean_expression);

// terminates program if false

Destructor:

```
List::~List()
{
   while (!isEmpty())
      remove(1);
}  // end destructor
```

isEmpty():

```
bool List::isEmpty() const
{
      return bool(size == 0);
}  // end isEmpty
```

getLength():

```
int List::getLength() const
{
   return size;
}  // end getLength
```

find(index):

```
List::ListNode *List::find(int index) const
{
   if ( (index < 1) || (index > getLength()) )
      return NULL;                    // index out of range
   else
   {  // marching down the list
      ListNode *cur = head;
      for (int skip = 1; skip < index; ++skip)
         cur = cur –> next;
      return cur;
   }
}  // end find
```

retrieve(index, dataItem):

```
void List:: retrieve(int index, ListItemType& dataItem) const
{
   if ( (index < 1) || (index > getLength()) )
      throw ListIndexOutOfRangeException(
         “ListOutOfRangeException: index out of range”);
   else
   {  // marching down the list
      ListNode *cur = find(index);
      dataItem = cur –> item;
   }
}  // end retrieve
```

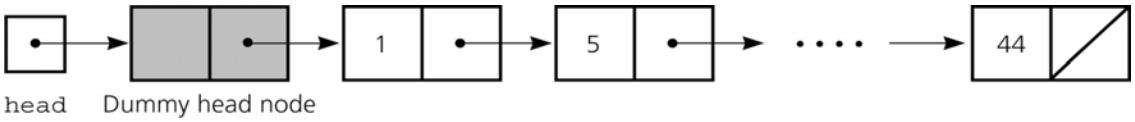
Comparing Array-Based & Pointer-Based Implementations:

- **Size**
  - Increasing the size of a resizable array can waste storage and time
- **Storage requirements**
  - Array-based implementations require less memory than a pointer-based ones
- **Access time**
  - Array-based: Requires constant access time
  - Pointer-based: The time to access the i^{th} node depends on i
- **Insertion and deletions**
  - Array-based: Require shifting of data
  - Pointer-based: Require traversing the list

Variations of Linked List:

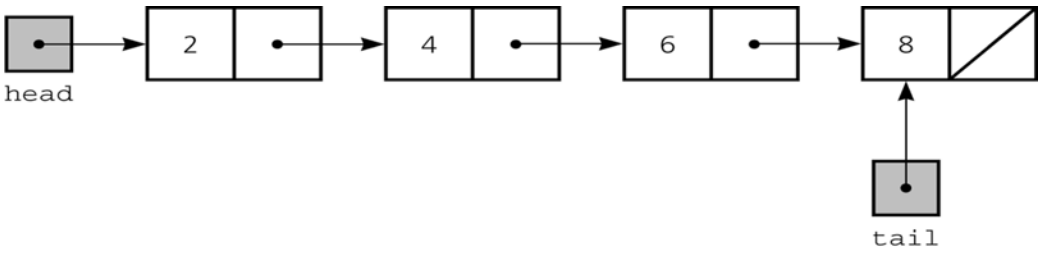
0. Singly linked list:

1. Linked list with dummy header node:



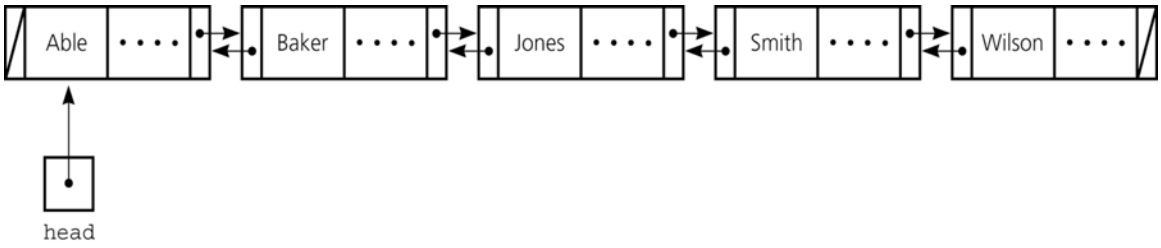
Empty list: head –> next = NULL

2. Linked list with head and tail pointers:



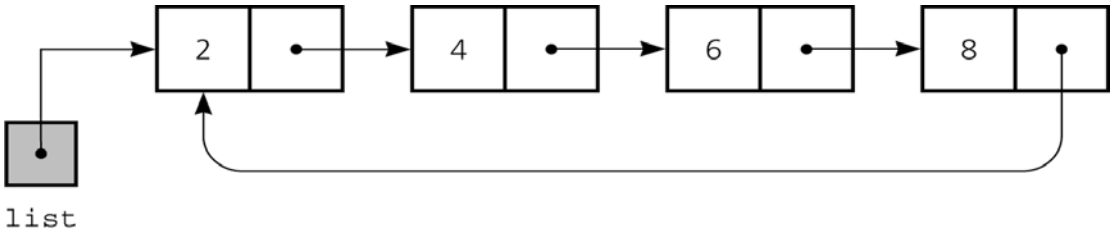
Empty list: head = tail = NULL

3. Doubly linked list:



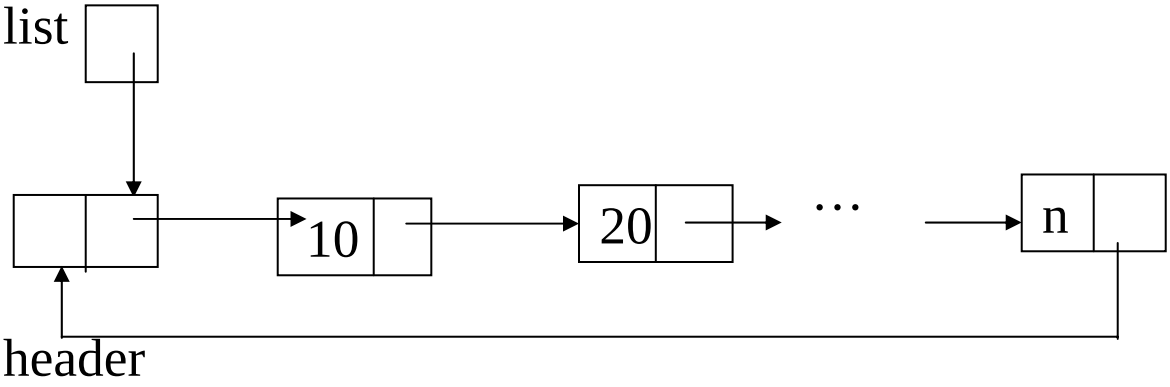
Empty list: head = NULL

4. Circular linked list:



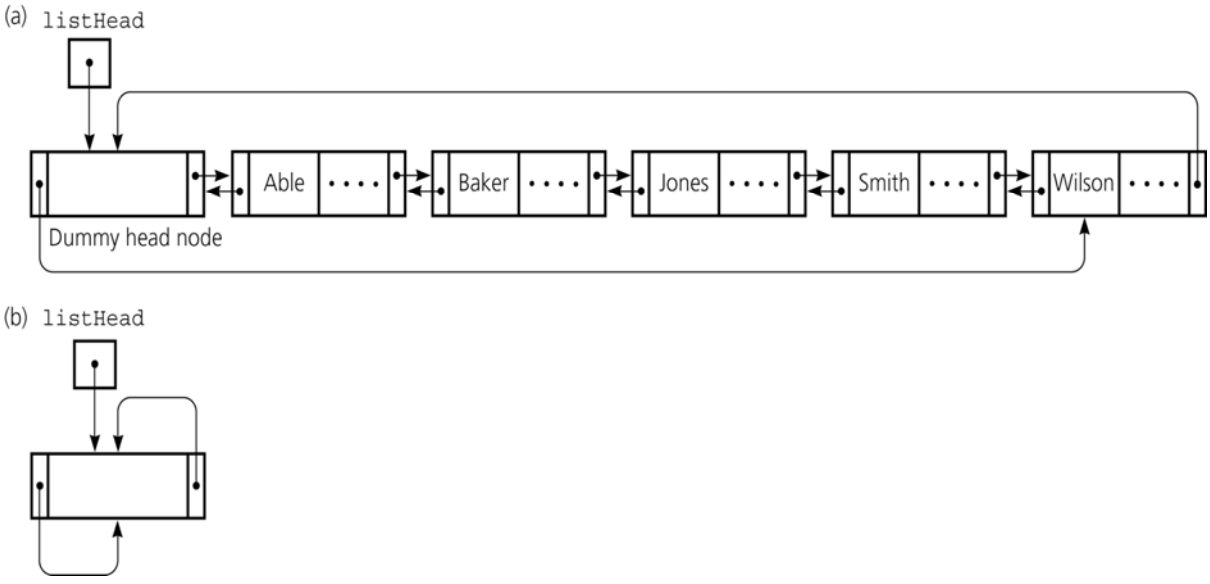
Empty list: list = NULL

5. Circular linked list with header node:



Empty list: list = list –> next

6. Circular doubly linked list with header node:



Empty list: list = list –> next

Example: Doubly linked list class.

```
class dList     // Class definition for doubly linked list
{
public:
// dList();
   dList(const dList& aList);
   ~dList();

   bool isEmpty() const;
   int getLength() const;
   void insert(int index, ListItemType newItem)
        throw(ListIndexOutOfRangeException, ListException);
   void remove(int index)
        throw(ListIndexOutOfRangeException);
   void retrieve(int index, ListItemType& dataItem) const
        throw(ListIndexOutOfRangeException);

private:
   struct doubleNode
   {
     ListItemType item;
     doubleNode *prev, *next;  //pointers to previous/next node
   }; // end struct

   int size;
   doubleNode *head;
   doubleNode *find(int index) const;
}; // end dList class
```

Copy Constructor:

```
dList::dList(const dList& aList): size(aList.size)
{
   if (aList.head ==NULL)       // empty list
      head = NULL;
   else
   {
      head = new doubleNode;    // get first node
      assert(head != NULL);     // check memory allocation
      head –> item = aList.head –> item;
                                // copy first item of old list
      head –> prev = NULL;      // set prev pointer of first node
      doubleNode *newPtr, *tempPtr;  //copying other node(s)
      *newPtr = head;
      for (doubleNode *origPtr = aList.head –> next;
            origPtr != NULL; origPtr = origPtr –> next)
      {
         newPtr –> next = new doubleNode;
         assert(newPtr –> next != NULL);
         tempPtr = newPtr;
         newPtr = newPtr –> next;
         newPtr –> prev = tempPtr;
         newPtr –> item = origPtr –> item;
      }
      newPtr –> next = NULL;
   }  // endif
}  // end copy constructor
```

```
void dList::insert(int index, ListItemType newItem)
{
   int newlength = getLength() + 1;

   if ( (index < 1) || (index > newLength) )     // no insert
      throw ListIndexOutOfRangeException(
      "ListOutOfRangeException: index out of range");
   else
   {
      doubleNode *newPtr = new doubleNode;  //allocate node
      if (newPtr == NULL)
         throw ListException(
         "ListException: Out of memory");
      else
      {
         size = newLength;
         newPtr –> item =newItem;
         if (index == 1)              // insert at beginning of list
         {
            newPtr –> next = head;
            newPtr –> prev = NULL;

            if (head != NULL)      // insert to non-empty list
               newPtr –> next –> prev = newPtr;

            head = newPtr;
         }
         else                         // general insert
```

```
        {
          doubleNode *backPtr = find(index – 1);
          newPtr –> prev = backPtr;
          newPtr –> next = backPtr –> next;
          backPtr –> next –> prev = newPtr;
          backPtr –> next = newPtr;
        } // endif
      } // endif
    } // endif
} // end insert
```

HW: Implement other list operations.