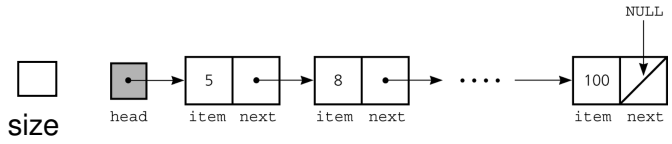
```
//*********************************************************
// ***Example taken from Carrano 3rd Edition ListP Linked List - Ch. 4
// ****************************************************************
// Header file LabListP.h for the ADT list.
// Pointer-based implementation.
// The first "position" in a list is position = 1,
//   as implemented in the insert(), remove(), retrieve(),
//   and private ptrTo() methods.
//*********************************************************

//*********************************************************
// The "typedef" below must be configured for the type
//   of data stored in each node in the list
//*********************************************************
typedef  desired-type-of-list-item   ListItemType;

class ListClass
{
public:
// constructors and destructor:
   ListClass();                          // default constructor
   ~ListClass();                         // destructor
   // Copy Constructor, Assignment operator=
   ListClass(const ListClass& existingList);
   ListClass& operator=(const ListClass& rhs);

// list operations:
   bool isEmpty() const;
   int  getLength() const;     // Access method

   bool insert(int position, ListItemType& newItem);
   bool remove(int position);
   bool retrieve(int position, ListItemType& dataItem) const;

private:
   struct ListNode              // a node on the list
   {
      ListItemType  item;       // a data item on the list
      ListNode     *next;       // pointer to next node
   };

   int size;           // number of items in list
   ListNode *head;     // pointer to linked list of items

   ListNode *ptrTo(int position) const;
   // Returns pointer to the node at position (1 .. k) in list
};
```



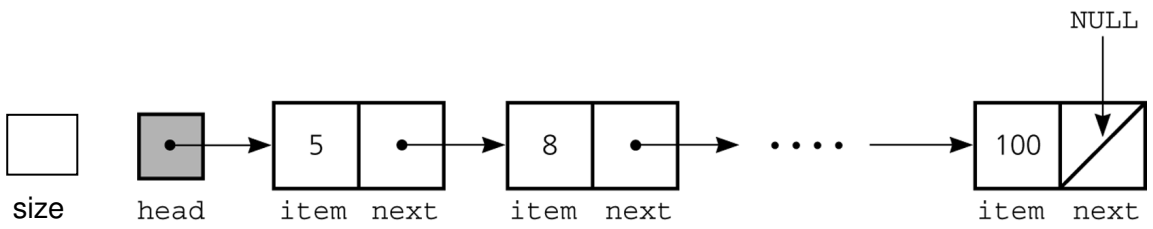
```
//*********************************************************
// Implementation file LabListP.cpp for the ADT list.
// Pointer-based implementation.
//*********************************************************
#include "LabListP.h"     // header file
#include <cstddef>        // for NULL
#include <cassert>        // for assert()
```

```
ListClass::ListClass()         // Default Constructor
    : size(0), head(NULL)
{
}
```

```
ListClass::~ListClass()        // Destructor
{
   while (!isEmpty())
      remove(1);
}
```

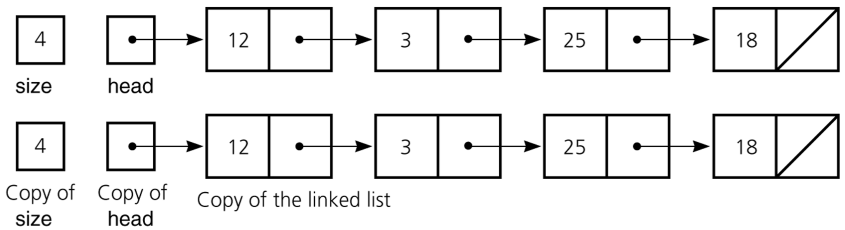
```
bool ListClass::isEmpty() const
{
   return bool(size == 0);
}
```

```
int ListClass::getLength() const   // Access method
{
   return size;
}
```



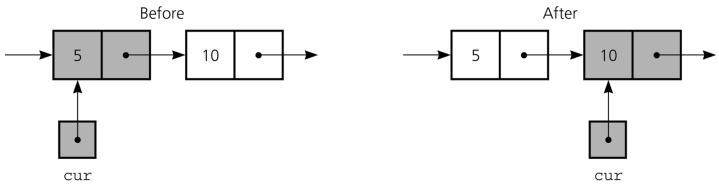
```
// Copy Constructor: Make DEEP Copy
ListClass::ListClass(const ListClass& existingList)
  : size(existingList.size)
{
   if (existingList.head == NULL)
   {
      head = NULL; // original list is empty
   }
   else
   {
      // copy first node
      head = new ListNode;
      assert(head != NULL); // check allocation

      head->item = existingList.head->item;
      head->next = NULL;

      // copy rest of list
      ListNode *newPtr = head; // new list pointer

      // newPtr points to last node in new list
      // origPtr points to nodes in original list

      for (ListNode *origPtr = existingList.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;
      }
   }
}
```

(b)



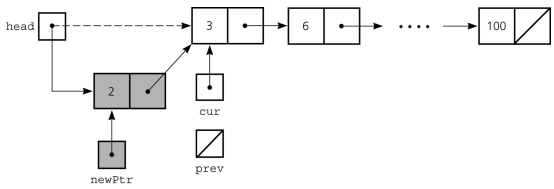
```
// Assignment operator=()  -  Make DEEP Copy
ListClass&  ListClass::operator=(const ListClass& rhs)
{
   // TODO
   // Similar to Copy Constructor, except
   // - Avoid self-assignments such as  "X = X;"
   // - Delete existing this-instance content before
   //   making this-instance a copy of the rhs instance

   return(*this);
}
```

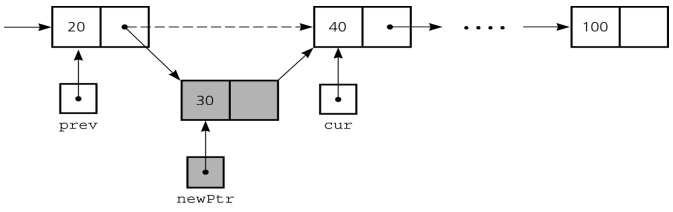
```
ListClass::ListNode *ListClass::ptrTo(int position) const
// ------------------------------------------------
// Locates a specified node in a linked list.
// Precondition: position is the number of the
//    desired node.  Valid range is (1 .. size).
// Postcondition: Returns a pointer to the desired node.
// If position < 1 or
//    position > the number of nodes in the list,
//       returns NULL.
// ------------------------------------------------
{
   if ( (position < 1) || (position > size) )
      return NULL;

   else // count from the beginning of the list
   {
      ListNode *cur = head;

      for (int skip = 1; skip < position; ++skip)
         cur = cur->next;

      return cur;
   }
}
```



```
bool ListClass::retrieve(int position,
                         ListItemType& dataItem) const
{
   bool success = bool((position >= 1) ||(position <= size));

   if (success)
   {
      // get pointer to node, then data in node
      ListNode *cur = ptrTo(position);

      dataItem = cur->item;
   }

   return(success);
}
```

```
bool ListClass::insert(int position, ListItemType newItem)
{
   int newLength = size + 1;

   bool success = bool((position >= 1) ||
                       (position <= newLength));

   if (success)
   {
      // create new node and place newItem in it
      ListNode *newPtr = new ListNode;

      if (newPtr == NULL)
         return(false); // cannot insert - allocation failed

      size = newLength;

      newPtr->item = newItem;

      // attach new node to list
      if (position == 1)
      {
         // insert new node at beginning of list
         newPtr->next = head;
         head = newPtr;
      }

      else
      {
         // insert new node to right of previous node
         ListNode *prev = ptrTo(position - 1);

         newPtr->next = prev->next;
         prev->next = newPtr;
      }
   }

   return(success);
}
```





```
bool ListClass::remove(int position)
{
   ListNode *cur;

   bool success = bool((position >= 1) ||
                       (position <= size));

   if (success)
   {
      --size;

      if (position == 1)
      {
         // delete the first node from the list

         // Locate node to be deleted: cur
         cur = head;       // save pointer to node

        // Remove cur node from the list
         head = head->next;
      }


      else
      {
         // Locate previous node
         ListNode *prev = ptrTo(position - 1);

         // Locate node to be deleted: cur
         cur = prev->next; // save pointer to node

        // Remove cur node from the list
         prev->next = cur->next;
      }

      // return node to system
      cur->next = NULL;        // safety - remove from list
      delete cur;
      cur = NULL;              // safety
   }

   return(success);
}
```