CSCI 2170 Linked List (1)

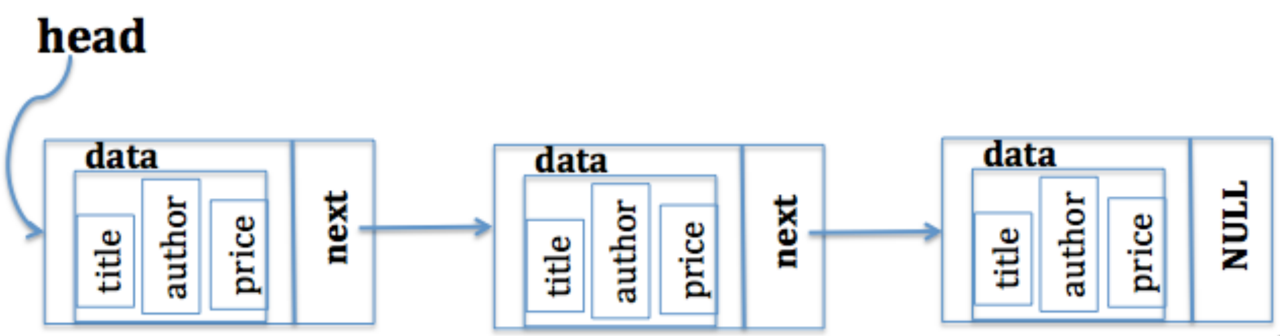
1. **Advantages** of using linked list, instead of array, to store data:
   a. Memory efficiency → exact amount of memory is allocated for the data
   b. Time efficiency → insertion into and deletion from a list are more efficient

2. **Define the basic structure to build a linked list:**

```
struct    BookStruct
{
    string    title;
    string    author;
    float     price;
};
typedef BookStruct ListItemType;

struct   Node
{
        ListItemType   data;
        Node*          next;
};
typedef Node* NodePtr;
```

3. **Examine a linked list of 3 nodes:**



- The 1st element in the list is special. Its name is "head". It is of type NodePtr, not Node
  - NodePtr head;
  - It is the only name by which the list nodes may be accessed
  - When the list is empty, i.e., when the list is first created and no node has been inserted into the list, designate head to be NULL
    - head=NULL;
  - (head==NULL) is a condition we can use to test whether the list is empty
  - (head != NULL) is a condition we can use to test that the end of the list has not been reached

- The ***next*** field of a node contains the memory address of the next node in the list
  - Important!! – it is how the nodes are linked together
  - The next field of the last node in the list has value NULL
    - It provides a way of detecting the end of the list

4. How to create a linked list of data items?

<u>For simplicity, the data will simply be an integer number in the following discussion:</u>

```
typedef int ListItemType;
struct   Node
{
         ListItemType  data;
        Node* next;
};
typedef Node* NodePtr;
```

a. create a linked list with 3 nodes to store contact information of three person

```
NodePtr    cur = new  Node;      // create the first node
if (cur != NULL)
{
        cur→data=5;
        cur→next = NULL;
}
head = cur;   // linked list with a single node. Head pointer is pointing to the node

// create the second node for insertion
NodePtr    cur = new Node;
if ( cur != NULL)
{
        cur→data = 9;
        cur→next = NULL;
}

cur→next = head;  // linked the two nodes together by putting the new node
head = cur;                // at the beginning of the list, head is updated to point
cur = NULL;               // to the new "head" of the list
```

practice: create the 3rd node (with a value 100) and put it at the beginning of the list (how about at the end of the list? or in the middle of the list?

5. Traversing the list (starting from the head of the list, visit the nodes in the list one by one)

a. print out the information in the list

```
NodePtr   curr=head;
while (curr!=NULL)              // stops when the next field of the last
{                               // node in the list is reached.

        cout << curr→data << endl;

        curr= curr→next;   // important! This is how to get from one
}                              // node to the next node
```

b. **Given a list of N nodes, print out the information of the node at position "*position*"**

```
NodePtr curr=head;
int  i=0;
while (curr !=NULL && i<position)  // detecting end of list should
{                                   // always be the first condition ( "short circuit evaluation" )
        curr = curr→next;
        i++;
};
if (curr!=NULL)
     cout << curr→data;
```

c. **Given a list of N nodes, search for a specific number (linear search)**

```
void Search(NodePtr & head, ListItemType toFind)  {

    bool found=false;
    NodePtr  curr=head;
    while (curr !=NULL)
    {       if (curr→data == toFind)
            {       cout << toFind << " is found"<< endl;'
                    found=true;
            }
            curr=curr→next;
    }
    if (!found)
         cout << "the value is not in the list" << endl;
}
```

d. **Given a list of N nodes, return a Nodeptr that points to the item at a specified position This is referred to as the PtrTo function**

```
NodePtr PtrTo(NodePtr  & head,  int  position) {
    NodePtr cur = head;
    for (int skip = 0; skip < position; ++skip)
      cur = cur->next;
    return cur;
}
```

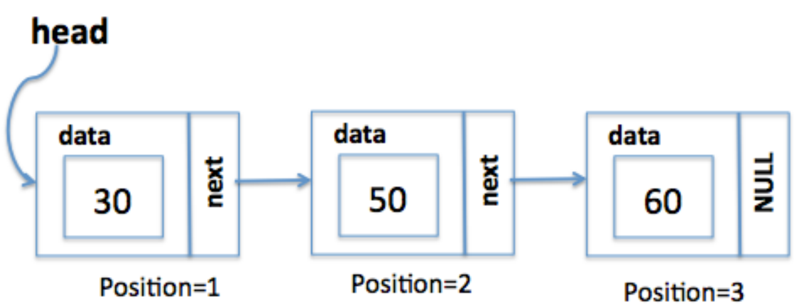
practice :
(1) how to print the position of the item in the list if the item is found?
(2) how to print out the content of the last node in the list?

6. Unsorted Linked list class

```
class listClass
{
  public:
        // constructors and destructor:
        listClass();                // default constructor
        listClass(const listClass& L); // copy constructor
        ~listClass();               // destructor

        // list operations:
        bool ListIsEmpty() const;
        int ListLength() const;
        void ListInsert(int NewPosition, listItemType NewItem,  bool& Success);
        void ListDelete(int Position, bool& Success);
        void ListRetrieve(int Position, listItemType& DataItem, bool& Success) const;
    private:
        int     Size;  // number of items in list
        nodePtr Head;  // pointer to linked list of items

        nodePtr PtrTo(int Position) const;
        // Returns a pointer to the Position-th node in the linked list.
}; // end class
```



a. insert a node at position "position" in an "unsorted list"
(This function should be to handle insertion at ALL proper locations)

Two cases: Case 1: position == 1 ➔ insert at the beginning of list
Case 2: position != 1 ➔ insert in the middle or end of list

Step1: create a new node, assign proper values to the new node
newNode = new Node
newNode→data = newData
newNode→next = NULL

Step2: if the new node is to be added at the beginning:
newNode→next = head

head = newNode;

Question: Does it take care of empty list situation?

Step 3: if the new node is to be added in the middle or at the end:

```
Nodeptr prev=PtrTo(newPosition-1);
// insert new node after node to which Prev points
NewPtr->next = Prev->next;
Prev->next = NewPtr;
```

Step 4: update the size of the list.

Question: Does this work for end of list insertion?

b. delete a node at position "position" in the list

two cases: (1) delete from the beginning → change the value of "head"

(2) delete from the middle or from the end of list → list traversal

Step 1: case 1 – position is 1

Detach first node from the list, update "head" value

cur = head;
head = head→next;

Step 2: case 2 – position is not 1:

Step 2a : traverse down the list and find the deletion point: **prev** points to the node right before the deletion location, and cur points to the node to be deleted:

nodePtr prev = PtrTo(**position-1**);
nodePtr cur = prev→next;

Step 2b: delete the node by: detach and relink

prev→next = cur→next;

Step 3: release the node cur is pointing at (for both case 1 and case 2)

cur→next = NULL;
delete cur;
cur= NULL;

Step 4: Update the size of the list

Linked list (unsorted)
Implementation file

```
#include "ListP.h"     // header file
#include <cstddef>    // for NULL
#include <cassert>    // for assert()
using namespace std;

listClass::listClass(): Size(0), Head(NULL)
{
}  // end default constructor

listClass::listClass(const listClass& L): Size(L.Size)
{
  if (L.Head == NULL)
    Head = NULL;  // original list is empty

  else
  {
    // copy first node
    Head = new Node;
    assert(Head != NULL);  // check allocation
    Head->item = L.Head->item;

    // copy rest of list
    nodePtr NewPtr = Head;  // new list pointer

    // NewPtr points to last node in new list
    // OrigPtr points to nodes in original list
    for (nodePtr OrigPtr = L.Head->next;   OrigPtr != NULL;  OrigPtr = OrigPtr->next)
    {
      NewPtr->next = new Node;
      assert(NewPtr->next != NULL);
      NewPtr = NewPtr->next;
      NewPtr->item = OrigPtr->item;
    }  // end for

    NewPtr->next = NULL;
  }  // end if
}  // end copy constructor

listClass::~listClass()
{
  bool Success;

  while (!ListIsEmpty())
    ListDelete(1, Success);
```

```
} // end destructor

bool listClass::ListIsEmpty() const
{
  return bool(Size == 0);
}  // end ListIsEmpty

int listClass::ListLength() const
{
  return Size;
}  // end ListLength

nodePtr listClass::PtrTo(int Position) const
// --------------------------------------------------
// Locates a specified node in a linked list.
// Precondition: Position is the number of the desired node.
// 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 > ListLength()) )
    return NULL;

  else  // count from the beginning of the list
  {  nodePtr Cur = Head;
    for (int Skip = 1; Skip < Position; ++Skip)
      Cur = Cur->next;
    return Cur;
  }  // end if
}  // end PtrTo

void listClass::ListRetrieve(int Position, listItemType& DataItem, bool& Success) const
{
  Success = bool( (Position >= 1) && (Position <= ListLength()) );

  if (Success) // get pointer to node, then data in node
  {
    nodePtr Cur = PtrTo(Position);
    DataItem = Cur->item;
  }  // end if
} // end ListRetrieve

void listClass::ListInsert(int NewPosition, listItemType NewItem,  bool& Success)
{
  int NewLength = ListLength() + 1;

  Success = bool( (NewPosition >= 1) &&  (NewPosition <= NewLength) );

  if (Success) // create new node and place NewItem in it
```

```
  {
    nodePtr NewPtr = new Node;
    Success = bool(NewPtr != NULL);
    if (Success)
    {
      Size = NewLength;
      NewPtr->item = NewItem;

      // attach new node to list
      if (NewPosition == 1) // insert new node at beginning of list
      {
        NewPtr->next = Head;
        Head = NewPtr;
      }

      else
      {  nodePtr Prev = PtrTo(NewPosition-1);   // insert new node after node to which Prev points
        NewPtr->next = Prev->next;
        Prev->next = NewPtr;
      }  // end if
    }  // end if
  }  // end if
} // end ListInsert

void listClass::ListDelete(int Position,   bool& Success)
{
  nodePtr Cur;

  Success = bool( (Position >= 1) &&  (Position <= ListLength()) );

  if (Success)
  {  --Size;
    if (Position == 1) // delete the first node from the list
    {
      Cur = Head;  // save pointer to node
      Head = Head->next;
    }
    else
    {  nodePtr Prev = PtrTo(Position-1);         // delete the node after the node to which Prev points
      Cur = Prev->next;  // save pointer to node
      Prev->next = Cur->next;
    }  // end if

    // return node to system
    Cur->next = NULL;
    delete Cur;
    Cur = NULL;
  }  // end if
}  // end ListDelete
```

7. Sorted Linked list

a. What if the list is sorted? Assuming the list is sorted in ascending order, how to insert a node with *value 40* into the list at the appropriate spot in the list? (This time, we assume that we don't know ahead of time what is the correct position for this value, it is to be determined by the code itself)

Step 2: decide if the list is empty

```
if (head == NULL)
        head = newNode
else if (40 < head→data) // add the newNode as the new head
{
        … // change link
}
```

Step 3:

```
prev=head;
curr=head;
while (curr!=NULL && 40<curr→data)
{
        prev=curr;
        curr=curr→next;
}
// change link to insert
```

Does this code handle the situation where we want to insert a value 15?
Or insert a value 75?

b. delete a node with *data* equal to 50.

Step 1 : search for node, position pointers

```
prev=head;
curr=head;
while (curr!=NULL && curr→data !="Mary")
{
        prev=curr;
        curr=curr→next;
}
// change link to delete
…
```

Step 2: three cases:

<case 1> delete at the front of the list
<case 2> delete in the middle or at the end
<case 3> item not in the list

how about empty list situation?

c. Make a copy of an entire list – deep copy vs. shallow copy
(Copy constructor of a listclass with pointer implementation)

d. Delete an entire list
(Destructor of a listclass with pointer implementation)

Linked list -- Sorted list (Ascending order)

```
void List::insert(ListItemType toAdd)
{
   nodePtr   prev, curr;
   nodePtr   newNode;

   // create new node
   newNode = new Node;
   assert(newNode);
   newNode->item = toAdd;

   prev=NULL;
   curr=head;
   while ((curr!=NULL)&&(curr->item < toAdd))
   {
       prev = curr;
       curr = curr->next;
   }

   // <case 1> insertion at the beginning of the list
   if (curr == head)
   {
      // add code here to perform insertion
      // at the head of the list
      newNode->next = head;
      head = newNode;
   }
   else // case2:insertion in the middle or end of list
   {
      // add code here
      newNode->next = curr;
      prev->next = newNode;
   }
}

void List::delete(ListIemType  toDelete)
{
     nodePtr curr, prev;

     if (head == NULL)
         cout << "The list is empty." << endl;
     else
     {
         prev= head;
         curr = head;
         while ((curr!=NULL) && (curr->item < toDelete))
         // can you switch the order of the two conditions ??
```

```
        {
                prev= curr;
                curr = curr->next;
        }

        if  ((curr == head) && (curr->item == toDelete))
        // delete from the head of the list
        {
              curr = head;
              head  = head->next;
              curr->next = NULL;
              delete curr;
              curr = NULL;

              size --;
        }
        else  if ((curr!=NULL)&&(curr != head) && (curr->item == toDelete))
        // found the node, prev points to the node in front of “foundNode”,
        // curr points to the “foundNode”
        {
              prev->next = curr->next;              // remove curr from the list
              curr->next = NULL;                    // delete the memory space
              delete curr;
              curr=NULL;

              size --;
        }
        else  // curr == NULL case
        {
              cout << toDelete << " is not in the list." << endl;
              cout << "Deletion operation not performed. " << endl;
        }
     }
}
```