CSCI 2170 **Lecture Notes (10)**

Linked List (1)

```
struct  Node
{       string name;
        string      phone;
        Node *    next;
 };
```

or

```
struct  Person
{     string   name;
      string     phone; };
struct Node
{     Person   data;
      Node*   next;
};
```

typedef Node* NodePtr;

Examine a linked list of N nodes:

- The 1st element in the list is special. It is of type NodePtr, not Node
  It is the only name by which the list nodes may be accessed
- When head==NULL, the list is empty
- The **next** field of a middle node contains the memory address of the next node in the list → that 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
- **Advantages** of using linked list, instead of array, to store data:
  - exact amount of memory is allocated for the data → more memory efficient
  - insertion, deletion into a list are more efficient

How to create a linked list of data items?

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

```
NodePtr    head = new  Node;
if (head != NULL)
{         head->name=”mary”;
          head->phone=”893-0983”;
          head->next = NULL;
}

// create a new node for insertion
NodePtr    cur = new Node;
if ( cur != NULL)
{         cur->name = “John”;
          cur->phone = “983-9987”;
          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 and put it at the beginning of the list

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

   (2.1) **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->name << " " << curr->phone << endl;
        curr= curr->next;   // important! This is how to get from one
}                             // node to the next node
```

   (2.2) Given a list of n nodes, print out the information of the node at position "*position*"

```
NodePtr curr=head;
int  i=1;
while (curr !=NULL && i<position)  // detecting end of list should
{                          // always be the first condition because of
                curr = curr->next;       // C++ "short circuit evaluation"
                i++;
};
if (curr!=NULL)
     cout << curr->name << " " << curr->phone;
```

   (2.3) Given a list of n nodes, print out the phone number of the contact whose name is NameGiven

```
bool found=false;
NodePtr  curr=head;
while (curr !=NULL)
{       if (curr->name == nameGiven)
        {       cout << curr->name << " " << curr->phone << endl;'
                found=true;
        }
    curr=curr->next;
}
if (!found)
     cout << "the person is not in the list" << endl;
```

practice : how to print out the content of the last node in the list?

(2.4) insert a node at position "position" in the list in "unsorted list"

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 inserted at the beginning:

Step 2a: newNode->next = head

Step 2b: head = newNode;

Questions: Does it take care of empty list situation?

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

Step 3a: traverse down the list and find the insertion point

```
curr = head;
prev = head;
count = 1;
while (curr!=NULL && count != position)
{
        prev= curr;
        curr = curr->next;
        count++;
 }
```

Step 3b: at this point, **curr** points at the position of insertion, **prev** points to the node right before the insertion location.

Insert the node by:

```
newNode->next = curr;
prev->next = newNode;
```

Step 4: update the size of the list.

Question: Does this work for end of list insertion?

(2.5) What if the list is sorted? Assuming the list is sorted based on name, how to insert a node with *name* "John" into the list at the appropriate spot in the list?

Step 2: decide if the list is empty

```
if (head == NULL)
        head = newNode
else if ("John" < head->name) // add the newNode as the new head
{
        … // same as (2.4) Step 2
}
```

Step 3a:

```
while (curr!=NULL && "John" <curr->name)
{  …
```

```
}
```

(2.6) 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 of end of list → list traversal

Step 1: case 1 – position is 1

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

```
curr = head;
head = head->next;
curr->next = NULL;
delete curr;
curr = NULL;
```

Step 2: case 2 – position is not 1, so:

Step 2a traverse down the list and find the insertion point

```
curr = head;
prev = head;
count = 1;
while (curr!=NULL && count != position)
{
        prev= curr;
        curr = curr->next;
 }
```

Step 2b: at this point, **curr** points at the position of deletion, **prev** points to the node right before the deletion location.

delete the node by: detach and relink

```
prev->next = curr->next;
curr->next = NULL;
delete curr;
curr= NULL;
```

Step 3 (optional) : release the node
Update the size of the list

(2.7) delete a node with *name* equal to "Mary".

Step 2a : while (curr!=NULL && curr->name !="Mary")

Step 2b: add one more case: "Mary" not in list

```
if (curr !=NULL)
{
    …   // same as in (2.5) step 2b
}
else
     cout << "This person not in list";
```

(2.8) Make a copy a list

(2.9) Delete a list

Linked list (unsorted)
Header file

```
// ***********************************
typedef desired-type-of-list-item listItemType;

struct Node        // a node on the list
{  listItemType item;  // a data item on the list
   nodePtr      next;  // pointer to next node
};  // end struct
typedef Node* nodePtr;  // pointer to node

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
// End of header file.
```

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
```

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 != 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
          {
                    cout << toDelete <<
                         " is not in the list." << endl;
                    cout << "Deletion operation
                              not performed. " << endl;
          }
     }
}
```