LAB SESSION 1

# BASIC OPERATIONS ON LINKED LIST

**1. OBJECTIVE**

The objectives of Lab 1 are (1) to introduce on how to represent a linked list as an OO class; (2) to implement some basic operations on linked list and (3) to use an pre-implemented class of linked list;

**2. EXAMPLE**

This section gives an example on how to represent a linked list as a class, as well as implement some basic operations of the linked list. The way to use an implemented linked list is also introduced.

Listing 1 gives an example of a class representing a node in a linked list. As stated in the lectures, a node in a linked list consists of two parts: data and the link to the next node.

|  |
| --- |
| class Node {  public:  int data;  Node\* next;  } |

**Listing 1**

Listing 2 illustrates a simple way to create a linked list using the *Node* class. The list consisted of 3 elements: {2,3,5}. We use the pointer *pHead* to keep track of the first element of the list, meanwhile *count* reflecting the number of list elements. We also use a temporary pointer *pTemp* for our work. To check that our code runs correctly, we use a loop to print out all of elements’ data.

|  |
| --- |
| void main() {  //declaration  Node \* pHead = NULL;  int count = 0;  Node\* pTemp = NULL;  //make the list  pTemp = new Node;  count++;  pTemp->data = 5;  pTemp->next = NULL; //  pHead = pTemp; // the list now is {5}  pTemp = new Node;  count++;  pTemp->data = 3;  pTemp->next = pHead;  pHead = pTemp; // the list now is {2,3}  pTemp = new Node;  count++;  pTemp->data = 2;  pTemp->next = pHead;  pHead = pTemp; // the list now is {2,3,5}  //print them out  while (pTemp!=NULL) {  cout << pTemp->data;  pTemp = pTemp->next;  }  } |

**Listing 2**

In that way, our program looks ill-organized. In Listing 3, we then improve it by combining *pHead*, *count* into a class named List. Then the operation of inserting a new element into the list will be implemented as the method *addFirst*. We also implement method *display* to print out the list elements.Note that we use constructor to initialize the head of the list as NULL, and destructor to free the memory allocated.

|  |
| --- |
| class List{  private:  int count;  Node\* pHead;  public:  List() {pHead=NULL; count = 0;}  void addFirst(int newdata) {  Node\* pTemp = new Node;  pTemp->data = newdata;  pTemp->next = pHead;  pHead = pTemp;  count++;  }  void display() {  Node\* pTemp = pHead;  while (pTemp!=NULL) {  cout << pTemp->data;  pTemp = pTemp->next;  }  }  ~List() {  Node\* pTemp = pHead;  while (pTemp!=NULL) {  pTemp = pTemp->next;  delete pHead;  pHead = pTemp;  }  }  } |

**Listing 3**

Having the List class implemented, the *main* function can be rewritten far simpler as depicted in Listing 4.

|  |
| --- |
| void main(){  List aList;  aList.addFirst(5);  aList.addFirst(3);  aList.addFirst(2);  aList.display();  } |

**Listing 4**

The implementation of a class for linked list as above-described in this section is just an example. There are many other variations of implementations for linked list as discussed in the Appendix section. Students can choose what they feel appropriate when working with exercises and assignments.

**3. EXERCISES**

**Required exercises:**

* 1. Rewrite the *main* function in Listing 4 to build and display a linked list as follows

{12, 6, 29, 12, 51, 35, 83, 35, 78}.

**Solution:**

|  |
| --- |
| void main() {  List aList;  aList.addFirst(78); aList.addFirst(35); aList.addFirst(83); aList.addFirst(35); aList.addFirst(51); aList.addFirst(12);  aList.addFirst(29); aList.addFirst(6); aList.addFirst(12);  aList.display();  } |

* 1. Consider the following function

|  |
| --- |
| List\* buildPosLinkedList()  {  List pList = new List;  int valid=1;  char choice;  int num;  while (valid) {  cout << “Do you want to enter a number? (Y/N):”;  cin >> choice;  if ((choice == ‘Y’) || (choice == ‘y’)) {  cin >> num;  if (num>0) pList.addFirst(num);  } else valid = 0;  }  return pList;  } |

1. Rewrite the *main* function in Exercise 3.1 to do the following tasks:

* use the *buildPosLinkedList* function to create a list of positive numbers based on input from user.
* display the list
* free the memory allocated to the list

**Solution:**

|  |
| --- |
| void main() {  List \*aList;  aList = buildPosLinkedList ();  aList->display();  delete aList; //free the aList  } |

1. Modify *buildPosLinkedList* into *buildEvenLinkedList* to create a list of even numbers (the numbers can be positive or negative).

Solution:

|  |
| --- |
| List\* buildEvenLinkedList(){  List\* pList = new List();  int valid=1;  char choice;  int num;  while (valid) {  cout << "Do you want to enter a number? (Y/N):";  cin >> choice;  if ((choice == 'Y') || (choice == 'y')){  cin >> num;  if (num%2 == 0) pList->addFirst(num);  } else valid = 0;  }  return pList;  } |

* 1. Write for the class List in Listing 3 an additional method int addFirstIfPerfectSquare (int n) which adds *n* to the list if *n* is a square number. In that case the returned result is 1, otherwise 0.

Solution:

|  |
| --- |
| int addFirstIfFibo(int n){  if(n < 0) return 0;  if(checkFibo(n)){  addFirst(n);  return 1;  }  return 0;  } |
| bool checkFibo(int n){  if (n < 0) return false;  if (n == 0 || n == 1) return true;  int a,b,c;  a = 0; b = 1; c = a + b;  while(c <= n){  if(c == n) return true;  a = b; b = c; c = a + b;  }  return false;  } |

* 1. Write for the class List in Listing 3 an additional method void addLast(int n) which adds *n* to the last position of the list.

**Solution:**

|  |
| --- |
| void addLast(int n){  Node\* pTail = pHead;  Node\* pTemp = new Node;  pTemp->data = n;  pTemp->next = NULL;  if(pHead == NULL){  pHead = pTemp;  } else{  while(pTail->next != NULL){  pTail = pTail->next;  }  pTail->next = pTemp;  }  count ++;  } |

* 1. Write for the class List in Listing 3 an additional method void addEvenFirst(int n) which
     + adds *n* to the first position of the list if *n* is an even number (n mod 2 == 0),
     + adds *n* to the last position of the list if *n* is an odd number (n mod 2 != 0).
  2. Write for the class List in Listing 3 an additional method int addPost(int n, int index) which adds *n* to the *ith* position of the list. The index of the first position in the list is 1. Assume that the list is having more than *i* elements when called.

**Solution:**

|  |
| --- |
| int addPost(int n, int index){  if(index < 1 || index > count + 1) return 0;  Node\* pIns = pHead;  Node\* pTemp = NULL;  if(index == 1){  addFirst(n);  }  else{  while(index > 2){  pIns = pIns->next;  index --;  }  pTemp = new Node;  pTemp->data = n;  pTemp->next = pIns->next;  pIns->next = pTemp;  count ++;  }  return 1;  } |

**Advanced exercises:**

* 1. Write for the class List in Listing 3 an additional method int addIfOrdered(int n) which adds *n* to the list if it has been sorted and there is no element which has value *n*. *n* is added to the list so that the ordering of the list is maintained (a list containing 1 element or an empty list is considered as an ascending ordered list). This method will return:
* 0 if the list is unsorted.
* 1 if the list is sorted in ascending order but there existed one element which has value *n*.
* 2 if the list is sorted in ascending order and *n* is added successfully.
* 3 if the list is sorted in descending order but there existed one element which has value *n*.
* 4 if the list is sorted in descending order and *n* is added successfully.
* **Solution:**

|  |
| --- |
| int addPost(int n, int index){  if(index < 1 || index > count + 1) return 0;    return 1;  } |

* 1. Write for the class List in Listing 3 an additional method void reverseList() that reverses the elements in List. Your solution should **contain only 1 loop**.
  2. Write for the class List in Listing 3 an additional method int countIncreLists() which counts the number of the ascending ordered sublists in the list. For example, when called with the list as built in Exercise 3.1, we have 4 ascending ordered sublists {12}, {12, 79, 82}, {21, 43}, {31, 35, 57}. Therefore, the returned result is 4.
  3. Write for the class List in Listing 3 an additional method void removeMidElement() that removes the middle element of the list. If the list has *2n* elements, remove the *nth* position of the list, otherwise if the list has *(2n + 1)* elements, remove the *(n+1)th* position of the list. Your solution should fulfill the following requirements:
* Only 1 loop allowed
* Do NOT use the count field
* Do NOT call other supplementary methods of the class List
* It is NOT a recursive algorithm.
  1. Assuming that we have two ordered list a and b that are arranged in ascending order. Write for the class List in Listing 3 an additional method void concatenateOrderedList(List a, List b) that concatenates these lists to create a new ordered list. After executing this method, a will point to this new list and b will point to NULL.

-- End --