Data Structures in C++

CMPE226- Data Structures

Linked Lists as ADT

Doubly Linked Lists

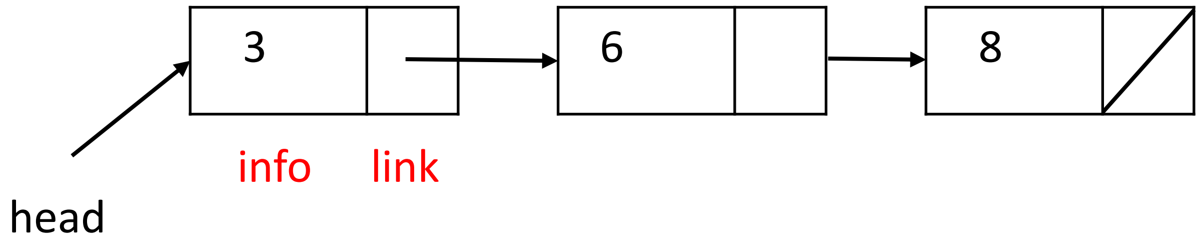
Circular Linked Lists

Adapted from Textbook (D. S. MALIK), Lecture notes (C. TURHAN) and Lecture Notes (H. QU)

Recall: Linked Lists

- Collection of components (nodes)
  - Every node (except last)
    - Contains address of the next node
- Node components
  - **Info:** stores relevant information
  - **Link:** stores address (Pointer to the next node in the list)

Example:



Head stores address of first node

```
struct Node{
    int info;
    Node *link;
};
int main(){
    //declare head pointer of type node:
    Node *head;
}
```

Linked Lists as an ADT

Generic definition of Linked Lists by using templates

Any type of list: int, double, string, object..

```
#include<iostream>
using namespace std;
template <class T>
struct node {
  T info; // T can be of any type
  node<T> *link;
};
int main() {
  node<int> *head = new node<int>;
  head->info=23;
  head->link=NULL;
  node<string> *p = new node<string>;
  p->info="cmpe226";
}
```

Abstract data type: A datatype that specifies the logical properties without the implementation details

Linked Lists as an ADT

An ADT defines a data type and a set of operations on that type.

- The name of ADT: type name
- The set of values belonging to ADT: domain
- The set of operations on the data.

Typename

Linked List

Domain

head (pointer to the first node)

last (pointer to the last node)

count of nodes in list.

Operations

Initialize the list

Check if the list is empty

Print the List

Length of a List

Search List

Insert Node

Delete Node

Destroy List

Copy List

Get First Element (Head node’s info)

Get Last Element

Linked Lists as an ADT

```
template <class T>
class LinkedList
{
protected:
    int count;
    Node<T> *head;
    Node<T> *last;
public:
    LinkedList();//constructor
    bool emptyList();//empty list
    int length();   //length
    Node<T>* search(T& val);//search
    void insertFirst(T& item);//Insert at first position
    void insertLast(T& item);//Insert at last position
    void deleteNode(T& item);//Delete a node
    void destroyList();//Destroy List
    void copyList(LinkedList<T> &);//Copy List
    ~LinkedList();//Destructor
    template <class S>
    friend std::ostream & operator<< (std::ostream &, const LinkedList<S> &);
    LinkedList & operator = (LinkedList &);
    T front();//first element
    T back();//last element
};
```

Linked List as an ADT

- Default constructor
  - Initializes list to an empty state

```
//constructor
template <class T>
LinkedList<T>::LinkedList()
{
    head = NULL;
    last = NULL;
    count = 0;
}
```

- Length of a list
  - Number of nodes stored in the variable `count`
  - Function `length`
    - Returns value of variable `count`

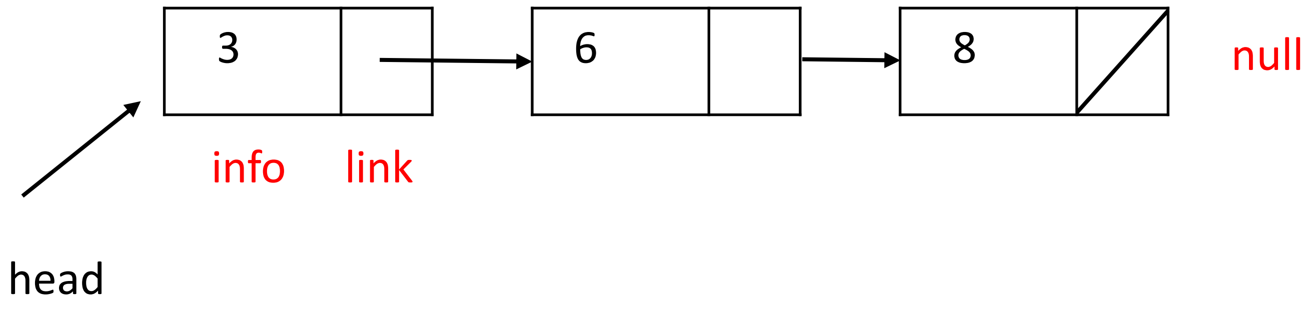
```
template <class T>
int LinkedList<T>::length()
{
    return count;
}
```

Linked List as an ADT

- Destroy the list
  - Deallocates memory occupied by each node
  - Must traverse list starting at first node
  - Delete each node in the list
- Destructor
  - Calling `destroyList` destroys list

```
//Destroy List
template <class T>
void LinkedList<T>::destroyList()
{
    Node<T> *p;

    while (head != NULL)
    {
        p = head;
        head = head->link;
        delete p;
    }

    last = NULL;
    count = 0;
}
```

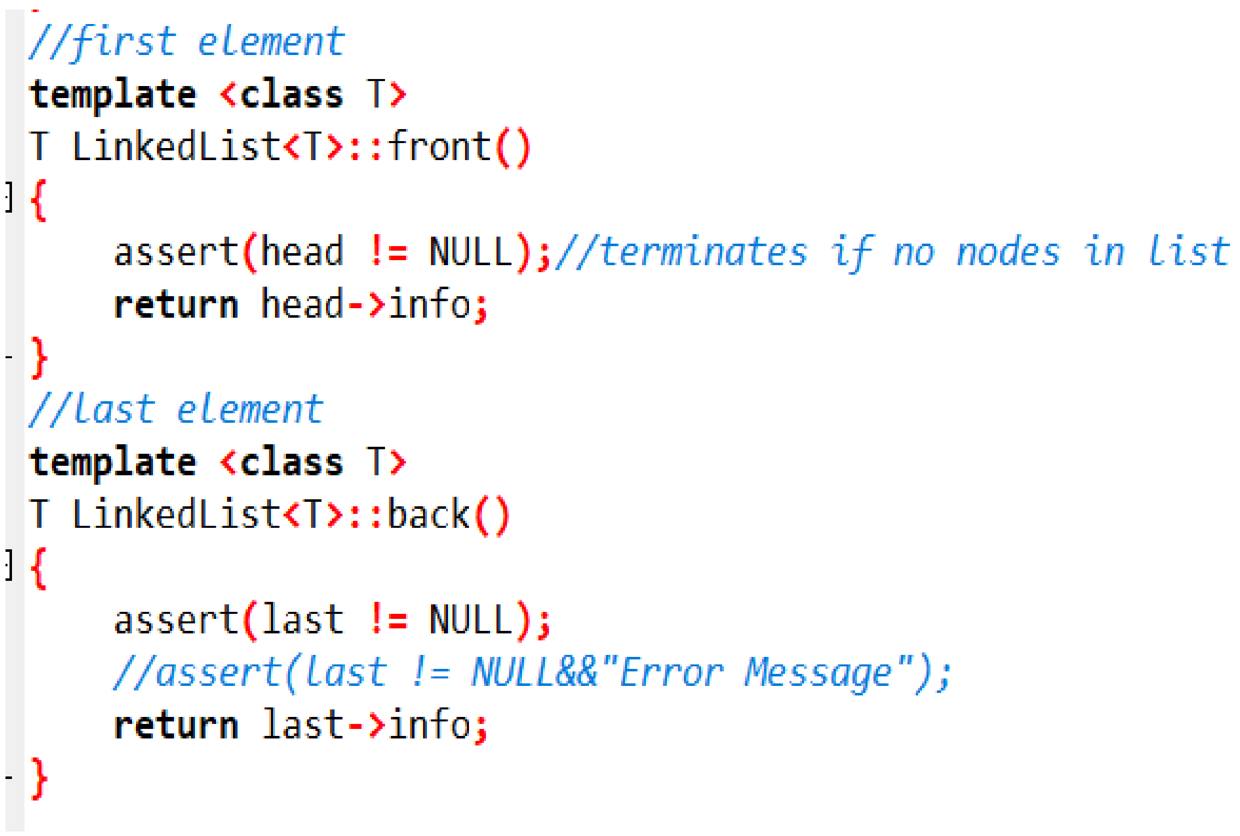
Example:



```
//Destructor
template <class T>
LinkedList<T>::~LinkedList()
{
    destroyList();
}
```

Linked List as an ADT

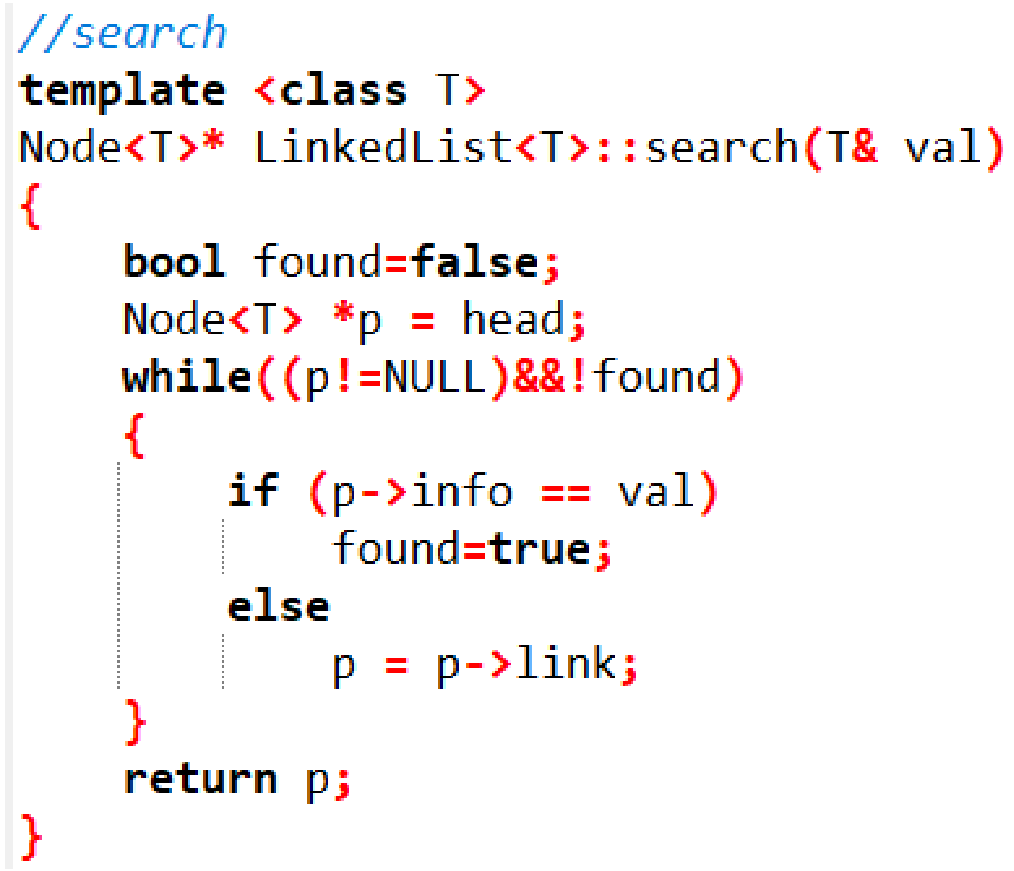
- Retrieve the data of the first node
  - Function `front`
    - Returns the info contained in the first node
    - If list is empty, `assert` statement terminates program
- Retrieve the data of the last node
  - Function `back`
    - Returns info contained in the last node
    - If list is empty, `assert` statement terminates program



```
//first element
template <class T>
T LinkedList<T>::front()
{
    assert(head != NULL);//terminates if no nodes in list
    return head->info;
}
//last element
template <class T>
T LinkedList<T>::back()
{
    assert(last != NULL);
    //assert(last != NULL&&"Error Message");
    return last->info;
}
```

Linked List as an ADT

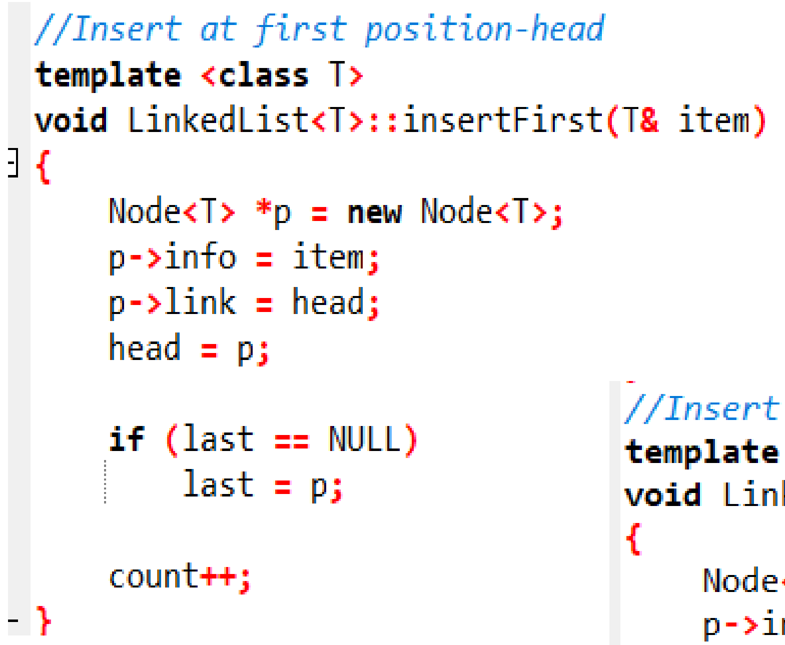
- Search the list
    - Steps
    - Step one: Compare the search item with the current node in the list. If the info of the current node is the same as the search item, stop the search; otherwise, make the next node the current node.
    - Step two: Repeat Step one until either the item is found or no more data is left in the list to compare with the search item.



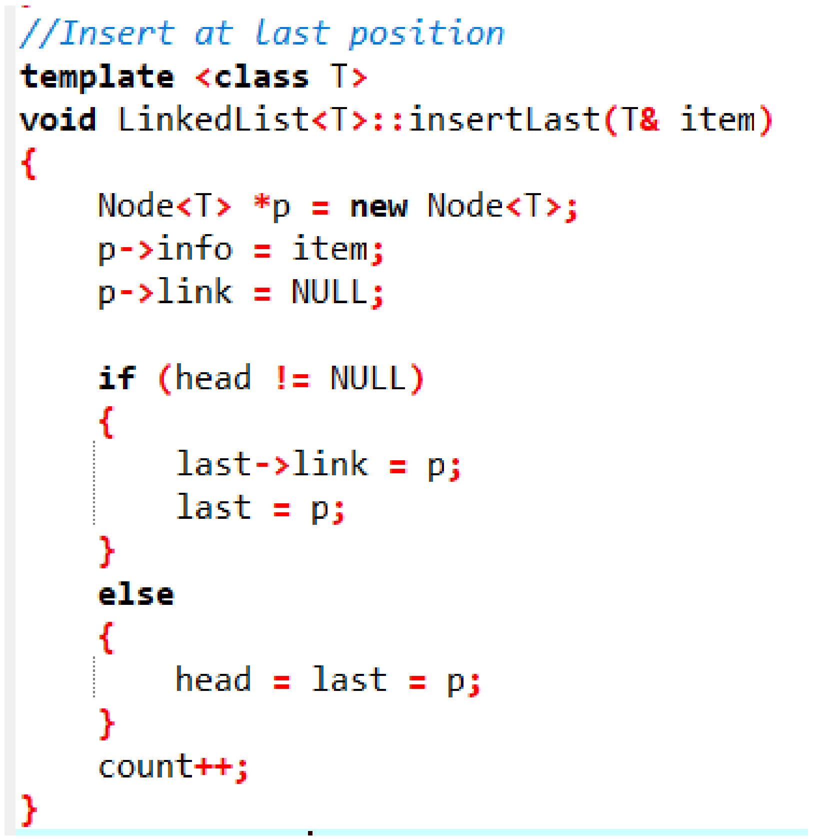
```
//search
template <class T>
Node<T>* LinkedList<T>::search(T& val)
{
    bool found=false;
    Node<T> *p = head;
    while((p!=NULL)&&!found)
    {
        if (p->info == val)
            found=true;
        else
            p = p->link;
    }
    return p;
}
```

Linked List as an ADT

- Insert the first node
  - Steps
    - Create a new node
    - If unable to create the node, terminate the program
    - Store the new item in the new node
    - Insert the node before first
    - Increment count by one
- Insert the last node
  - Similar to definition of member function `insertFirst`
  - Insert new node after last



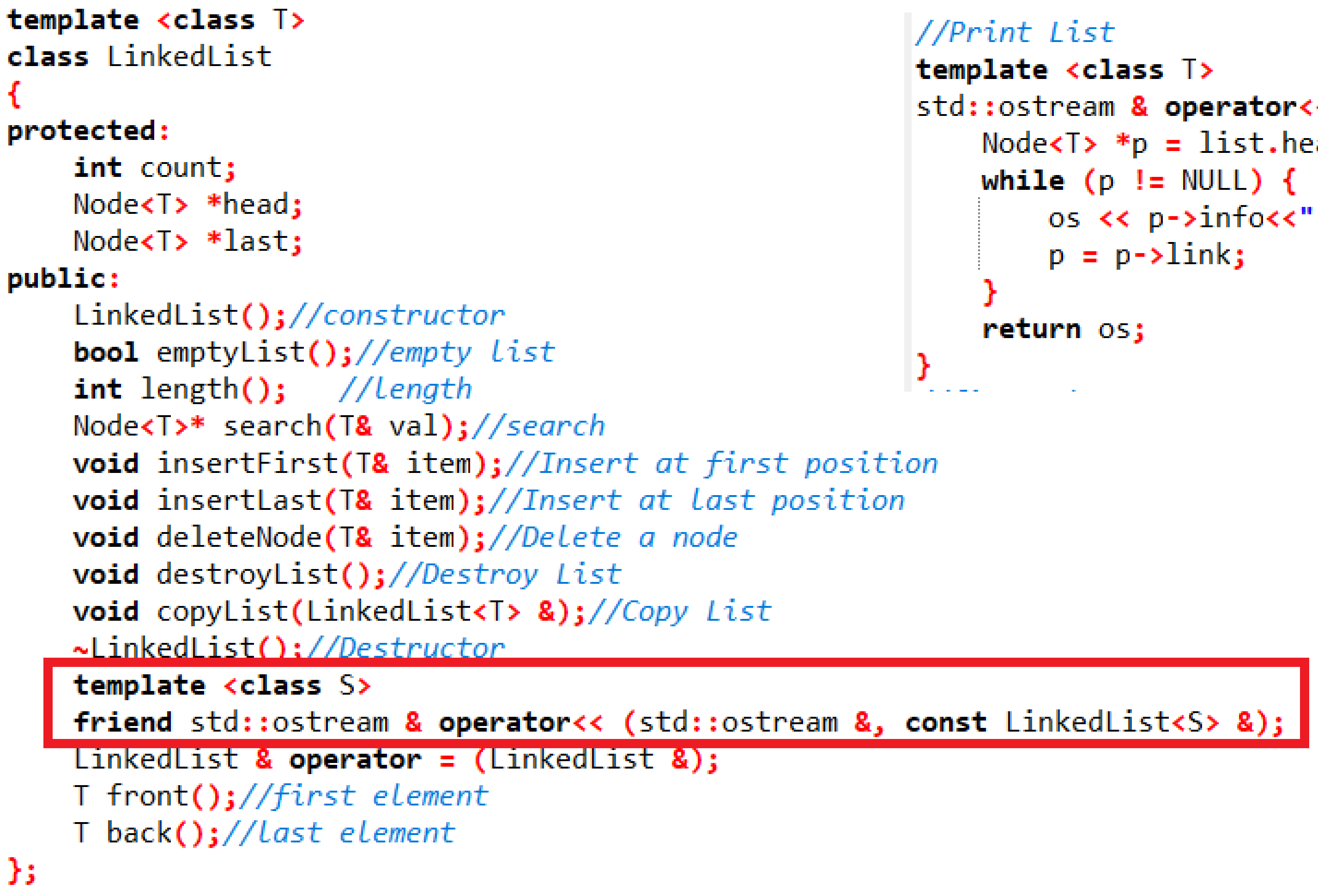
```cpp
//Insert at first position-head
template <class T>
void LinkedList<T>::insertFirst(T& item)
{
    Node<T> *p = new Node<T>;
    p->info = item;
    p->link = head;
    head = p;

    if (last == NULL)
        last = p;

    count++;
}
```



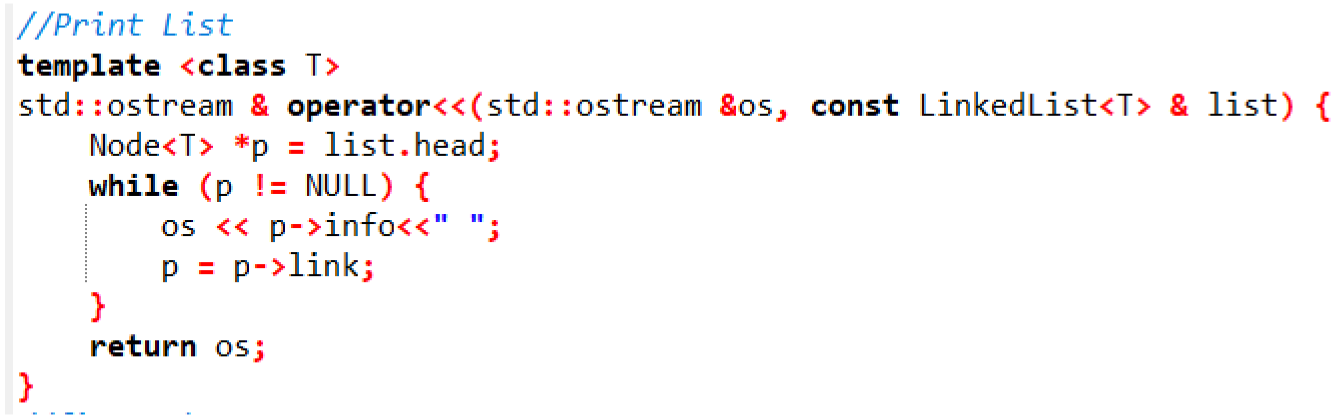
```cpp
//Insert at last position
template <class T>
void LinkedList<T>::insertLast(T& item)
{
    Node<T> *p = new Node<T>;
    p->info = item;
    p->link = NULL;

    if (head != NULL)
    {
        last->link = p;
        last = p;
    }
    else
    {
        head = last = p;
    }
    count++;
}
```

Linked List as an ADT

- Print the list
  - Must traverse the list starting at first node



```
template <class T>
class LinkedList
{
protected:
    int count;
    Node<T> *head;
    Node<T> *last;
public:
    LinkedList();//constructor
    bool emptyList();//empty list
    int length();   //length
    Node<T>* search(T& val);//search
    void insertFirst(T& item);//Insert at first position
    void insertLast(T& item);//Insert at last position
    void deleteNode(T& item);//Delete a node
    void destroyList();//Destroy List
    void copyList(LinkedList<T> &);//Copy List
    ~LinkedList();//Destructor
    template <class S>
    friend std::ostream & operator<< (std::ostream &, const LinkedList<S> &);
    LinkedList & operator = (LinkedList &);
    T front();//first element
    T back();//last element
};
```



```
//Print List
template <class T>
std::ostream & operator<<(std::ostream &os, const LinkedList<T> & list) {
    Node<T> *p = list.head;
    while (p != NULL) {
        os << p->info<<" ";
        p = p->link;
    }
    return os;
}
```

Linked List as an ADT

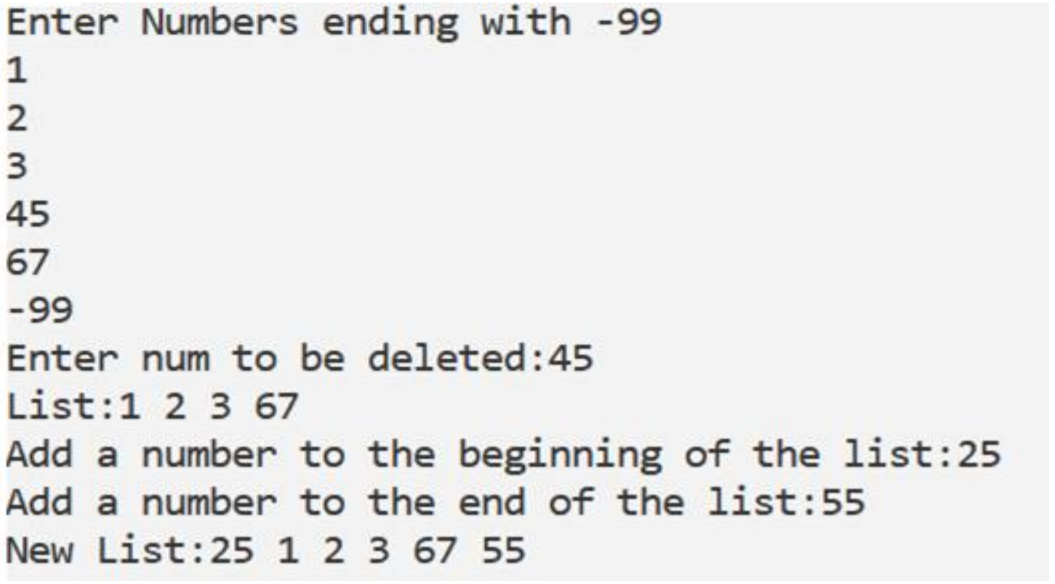
- Delete a node
  - The list is empty
  - The node is nonempty and the node to be deleted is the first node
  - The node is nonempty and the node to be deleted is not the first node, it is somewhere in the list
  - The node to be deleted is not in the list.

Check from LinkedList.h file provided to you on Moodle.

Linked List as an ADT: **Example 1**

Using the LinkedList.h file provided to you, create a list with integers (until user enters -99), delete a node and print result. Then add a number to the beginning and another number to the ending of the list and print result.

Sample Run:



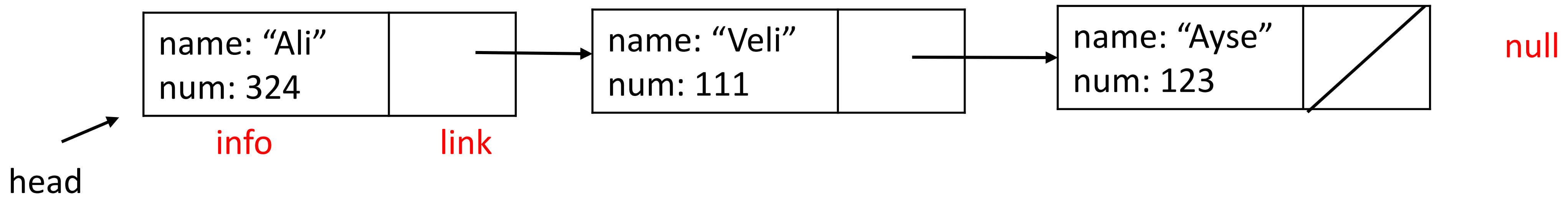
```
Enter Numbers ending with -99
1
2
3
45
67
-99
Enter num to be deleted:45
List:1 2 3 67
Add a number to the beginning of the list:25
Add a number to the end of the list:55
New List:25 1 2 3 67 55
```

Ex 1. Solution

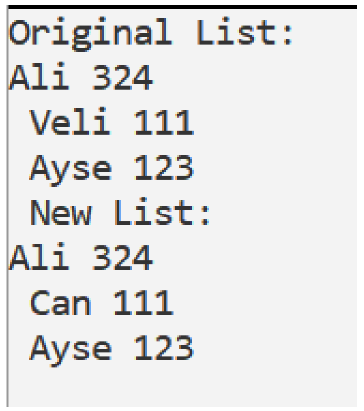
```
int main() {
        LinkedList<int> list;
        int num;
        cout << "Enter Numbers ending with -99" <<endl;
        cin >> num;
        while (num != -99) {
                list.insertLast(num);
                cin >> num;
        }
        cout << "Enter num to be deleted:";
        cin >> num;
        list.deleteNode(num);
        cout << "List:";
        cout << list <<endl;

        int h,t;
        cout << "Add a number to the beginning of the 
list:";
        cin >> h;
        list.insertFirst(h);
        cout << "Add a number to the end of the list:";
        cin >> t;
        list.insertLast(t);
  cout << "New List:";
        cout << list;
        return 0;
}
```

Linked List as an ADT: **Example 2**

Using the LinkedList.h file provided to you, from a text file input the names and id numbers of people and create a linked list of objects and output the result. Change the name of person numbered 111 to “Can”.

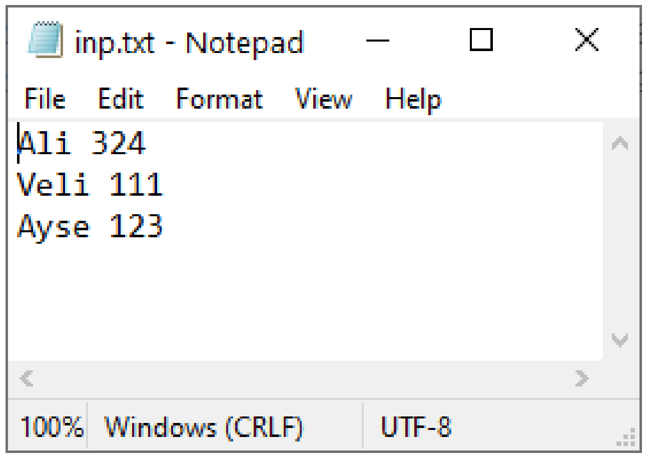


Sample Run:



```
Original List:
Ali 324
 Veli 111
 Ayse 123
 New List:
Ali 324
 Can 111
 Ayse 123
```

Sample Input File:



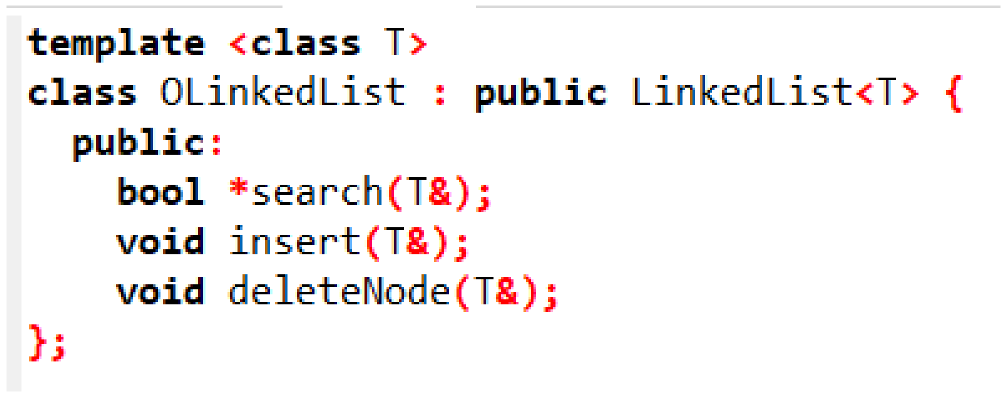
```
Ali 324
Veli 111
Ayse 123
```

```
#include <iostream>
#include <fstream>
#include <string>
#include "LinkedList.h"
using namespace std;
class Person {
public:
  int num;
  string name;
  Person () {};
  Person (int x) {num=x; }
  friend ostream & operator << (ostream &os, Person &per){
                        os << per.name << " " << per.num << endl;
                        return os;  }
  friend istream & operator >> (istream &is, Person &per){
                         is >> per.name >> per.num;
                        return is;  }
  bool operator == (Person &per) {
    return num==per.num;
//overloaded as a member function
in search func.-- - > if (p->info == val)
  }
};
main(){
  ifstream inp ("inp.txt") ;
  LinkedList <Person> per;
  Person p;
  inp>>p;
  while( !inp.eof() ){ //Create Linked List of objects
    per.insertLast(p);
    inp >> p;
  }
  cout <<"Original List:"<<endl; //Output the result
  cout << per;
  Person t(111);
  Node <Person> *np = per.search(t); //Search for a person with num 111.
  if ( np!= NULL)
                        np->info.name = "Can" ;
  cout <<"New List:"<<endl;
  cout<<per;
}
```

Ordered Linked Lists

- Derived From Linked List Class
- Assume elements are arranged in ascending order
- OPERATIONS
  - Initialize
  - Empty list
  - Destroy
  - Search *
  - Insert *
  - Delete *
  - Length

* are different, rest will be inherited.



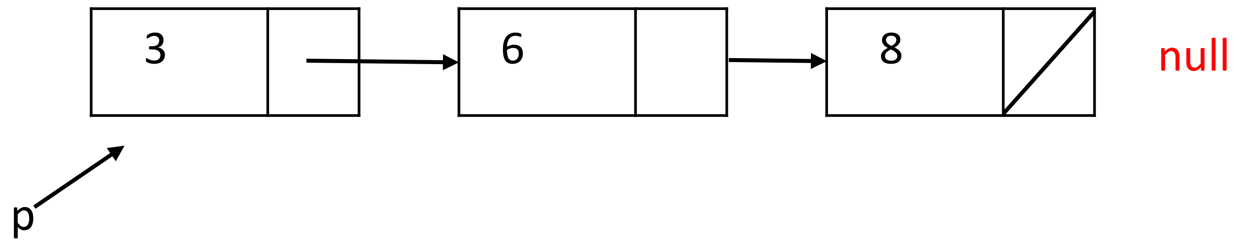
```
template <class T>
class OLinkedList : public LinkedList<T> {
  public:
    bool *search(T&);
    void insert(T&);
    void deleteNode(T&);
};
```

Ordered Linked Lists (cont'd.)

- Search the list
  - Steps describing algorithm
    - Step one: Compare the search item with the current node in the list. **If the info of the current node is greater than or equal to the search item, stop the search; otherwise, make the next node the current node**
    - Step two: Repeat Step one until either an item in the list that is greater than or equal to the search item is found, or no more data is left in the list to compare with the search item

```cpp
template <class T>
class OLinkedList : public LinkedList<T> {
  public:
    bool *search(T&);
    void insert(T&);
    void deleteNode(T&);
};

template <class T>
bool * OLinkedList<T>::search(T &item){
  bool found=false;
  // is a pointer to traverse the list
  node<T> *p = head;// Start search at first node
  while( p!=NULL && !found) {
    // Stop searching when data item becomes smaller than the node's data
    if( p->info >= item )
        found=true;
    else
      p = p->link;
    }
    if(found)
        found=(p->info==item);//test if equal
    return found;
}
```

Assume that searched item is 6.

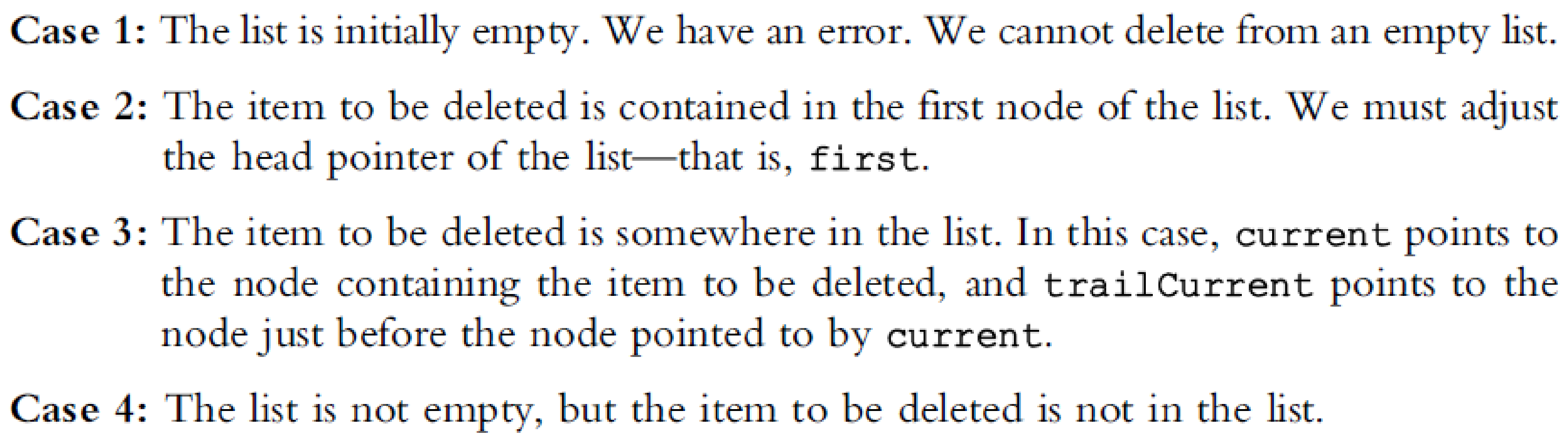


Ordered Linked Lists (cont'd.)

- Try to write insertion & deletion by yourself!
- Insert a node: Find place where new item goes
- *Hint: Use two pointers:*p for current and *q for node just before current (trailer)*
    - Case 1. List is initially empty. The new item will be the first node in list.
    - Case 2. The new item is smaller than the smallest element in the list (smaller than head). Adjust list's head pointer.
    - Case 3a. The new item is larger than all the items in list- insert to the end. In this case p is NULL and the item should be added after q.
    - Case 3b. The new item is to be inserted in the middle of the list- new item is inserted between p and q.
- After adding the node increment count by 1.
    - See code on page 304
        - Definition of the function `insert`

Ordered Linked Lists (cont'd.)

- Try to write insertion & deletion by yourself!
- Delete a node
  - Several cases to consider
  - See function `deleteNode` code on page 306



Case 1: The list is initially empty. We have an error. We cannot delete from an empty list.

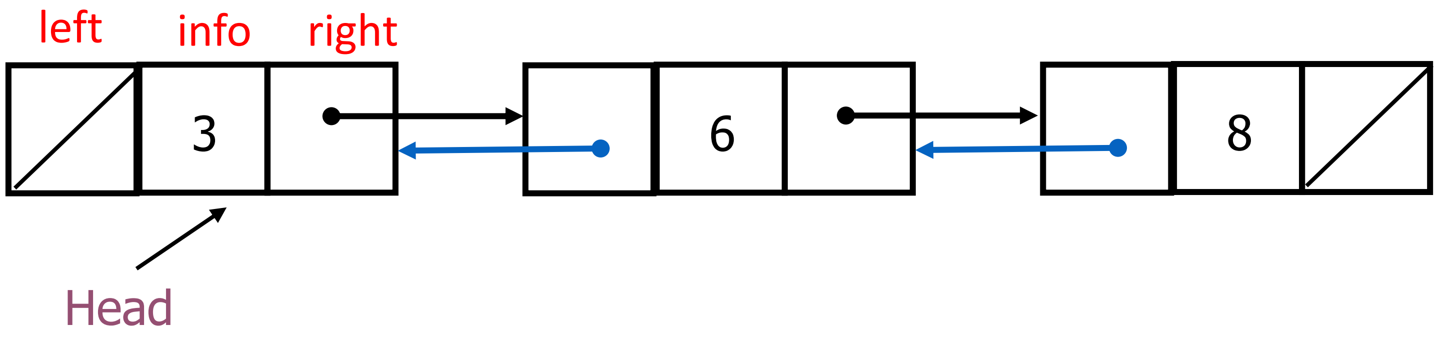
Case 2: The item to be deleted is contained in the first node of the list. We must adjust the head pointer of the list—that is, `first`.

Case 3: The item to be deleted is somewhere in the list. In this case, `current` points to the node containing the item to be deleted, and `trailCurrent` points to the node just before the node pointed to by `current`.

Case 4: The list is not empty, but the item to be deleted is not in the list.

Doubly Linked Lists

- In a Linked List:
  - If you want to delete a node, you need to find previous node.
  - It is difficult to traverse the list backward.
- In a Doubly Linked List:
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Next Lecture

Doubly Linked Lists - Implementation

Circular Linked Lists

Doubly Linked Lists

```
template <class T>
struct Node {
  T info;
  Node<T> *left, *right;
  //left points previous, right points next
};

template <class T>
class DLinkedList {
  protected:
    int cnt;
    node<T> *first, *last;
  private:
    void copyList( DLinkedList<T> & );
  public:
    DLinkedList () {
      first=NULL;
      last=NULL;
    }
    ~DLinkedList();
    bool search(T &);
    void insertLast(T &);
    void reverse();
    void deleteNode(T &)  ;
};
```

Doubly Linked Lists

```
template <class T>
void DLinkedList<T>::reverse() {
  node<T> *p = last;
  while(p!=NULL){
    cout << p->info;
    p=p->left;
  }
}

template <class T>
void DLinkedList<T>::insertLast(T &item) {
  node<T> *p = new node<T>;
  p->info = item;
  p->right = NULL;
  if ( first != NULL) {
    last->right = p;
    p->left = last;
    last = p;
  } else {
    first=last=p;
    p->left = NULL;
  }
  count++;
}
```

References

- CMPE226- Lecture Notes by Cigdem Turhan
- Data Structures Using C++, D.S. Malik, Thomson Course Technology, 2nd Edition.
- Lecture Slides by Huamin Qu, The Hong Kong University of Science and Technology (2005)