Data Structures in C++

CMPE226- Data Structures

Linked Lists as ADT

Doubly Linked Lists

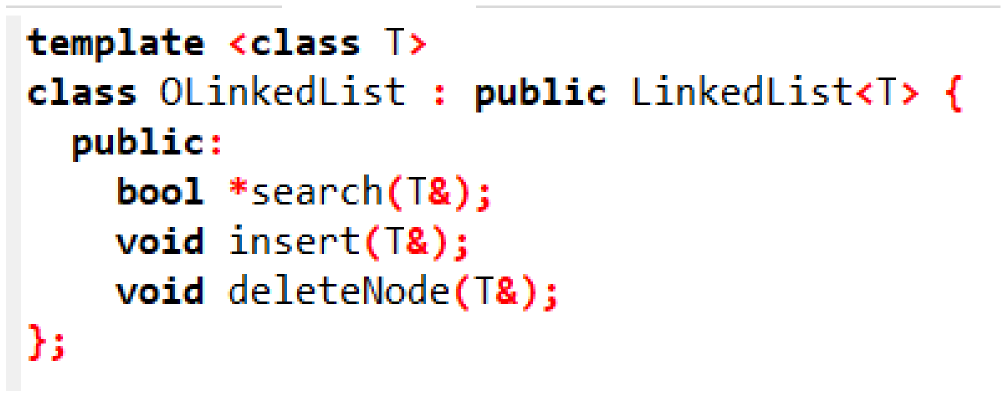
Circular Linked Lists

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

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&);
};
```

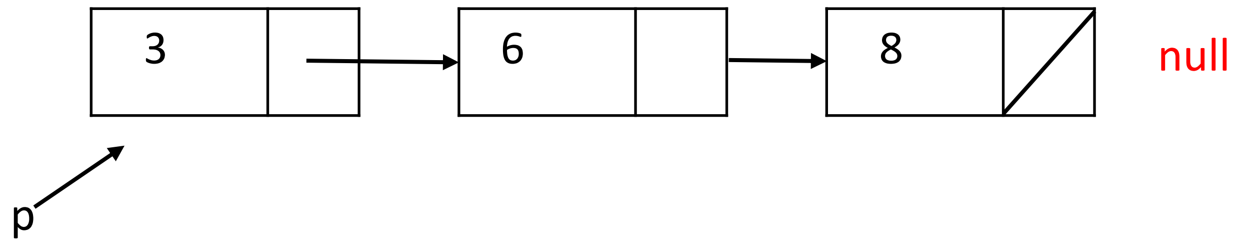
Ordered Linked Lists (cont'd.)

- 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

```
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`.

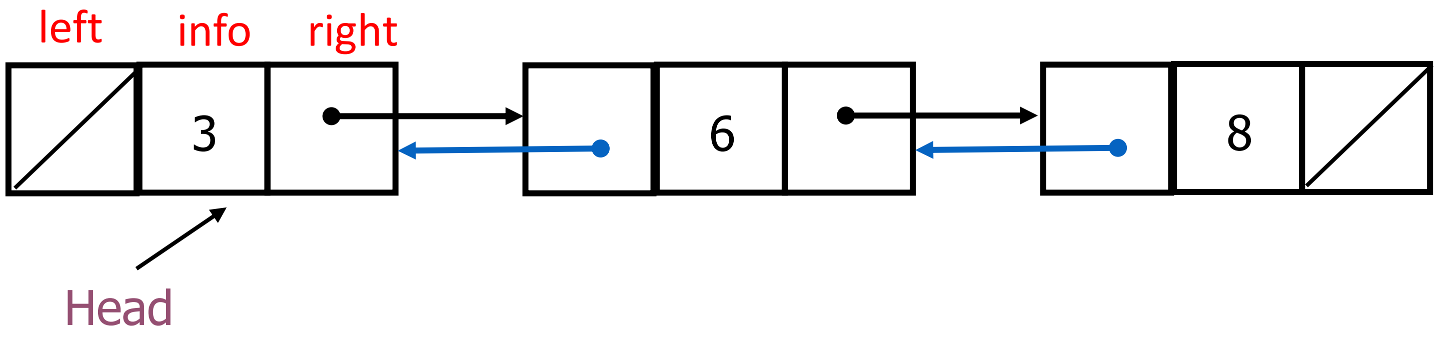
Case 3: The item to be deleted is somewhere in the list. In this case, `current` points to the node containing the item to be deleted, and `trailCurrent` points to the node just before the node pointed to by `current`.

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

Doubly Linked Lists

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

- 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



Doubly Linked Lists

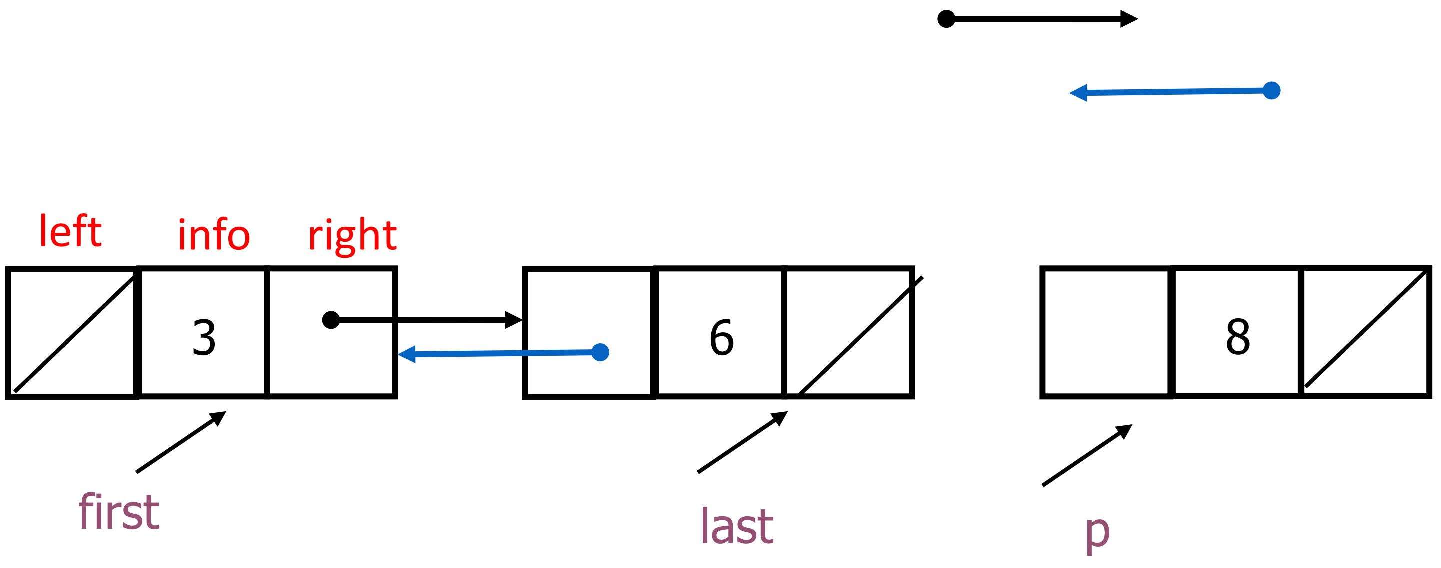
```
template <class T>
struct Node {
  T info;
  Node<T> *left, *right;
  //left points previous, right points next
};

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 &) ;
};
```

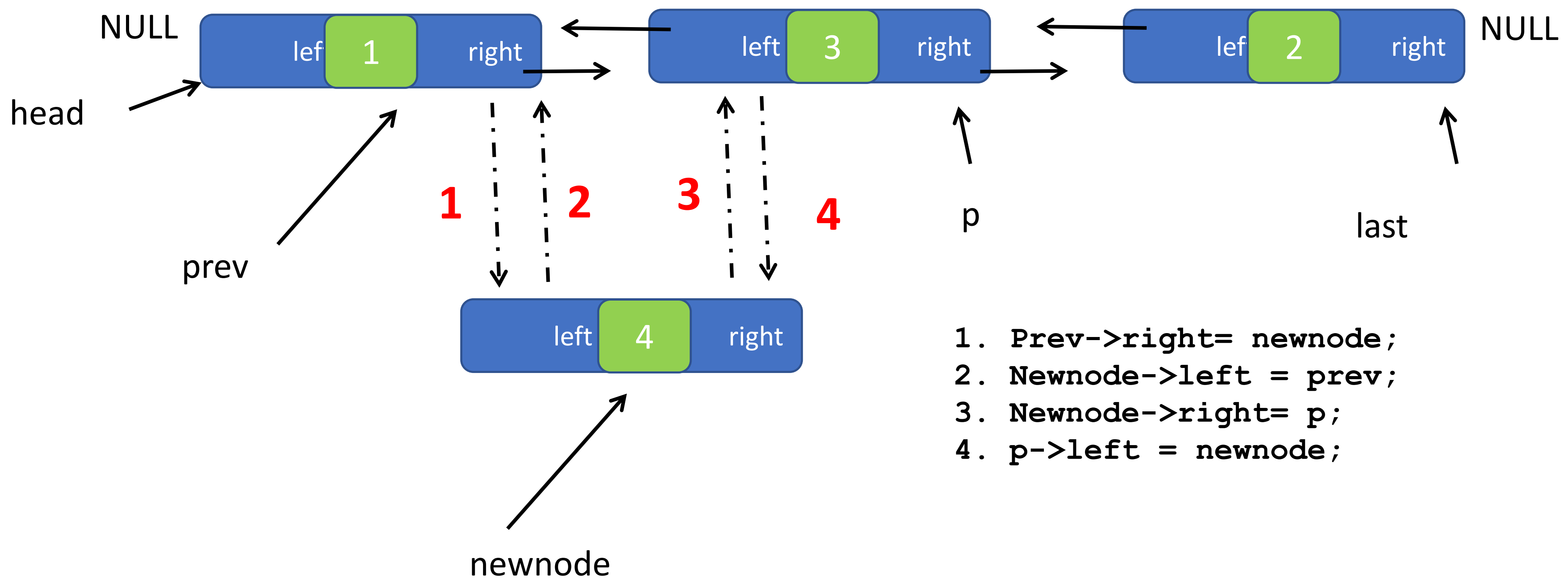
Doubly Linked Lists

```
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++;
}
```



Doubly Linked Lists: Insert in the Middle



```
1. Prev->right= newnode;
2. Newnode->left = prev;
3. Newnode->right= p;
4. p->left = newnode;
```

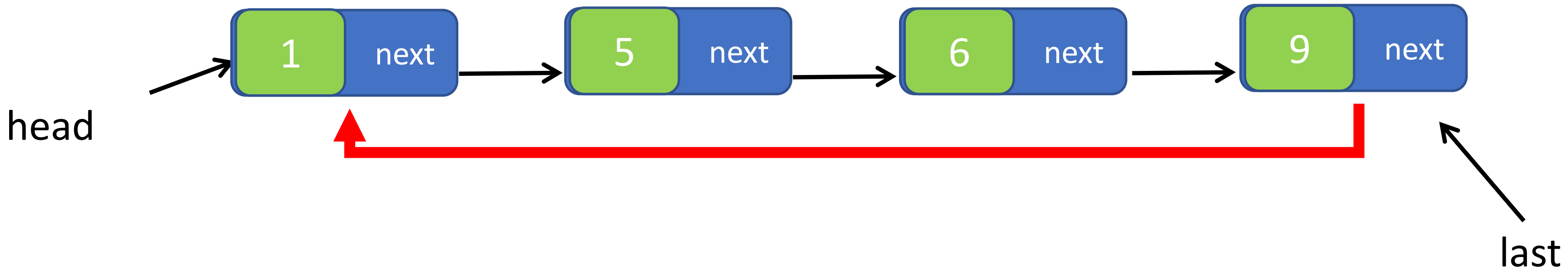
Circular Linked List

- A Linked List where last node points to the first node.
- There is no NULL at the end.
- Traversing the list:
- We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

```
Find the average of the list:

cnt=sum=0;
p=last;
do  {
  sum += p->info;
  cnt++;
  p=p->link;
} while (p!=last)

cout << sum/cnt;
```



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)