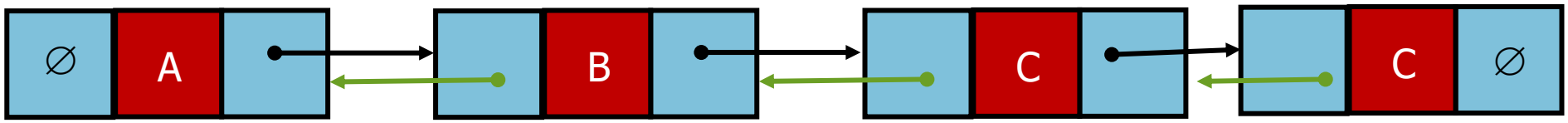
Doubly Linked Lists

Why ?

In singly link list the nodes contain only pointers to the successors; therefore, there is no immediate access to the predecessors

DL List NODE

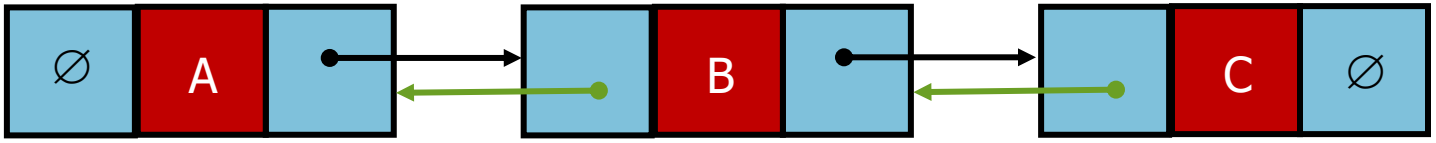


Each node points to not only successor but also to the predecessor

DL List NODE

```
template<class T>
class DLLNode {
public:
    DLLNode() {
        next = prev = 0;
    }
    DLLNode(const T & el,DLLNode *p = 0, DLLNode *n = 0) {
        info = el; next = n; prev = p;
    }
    Friend class List; /
private:
    T info;
    DLLNode<T> *prev, *next;
};
```

```
template<typename T>friend class List;
For visual studio use this syntax
```

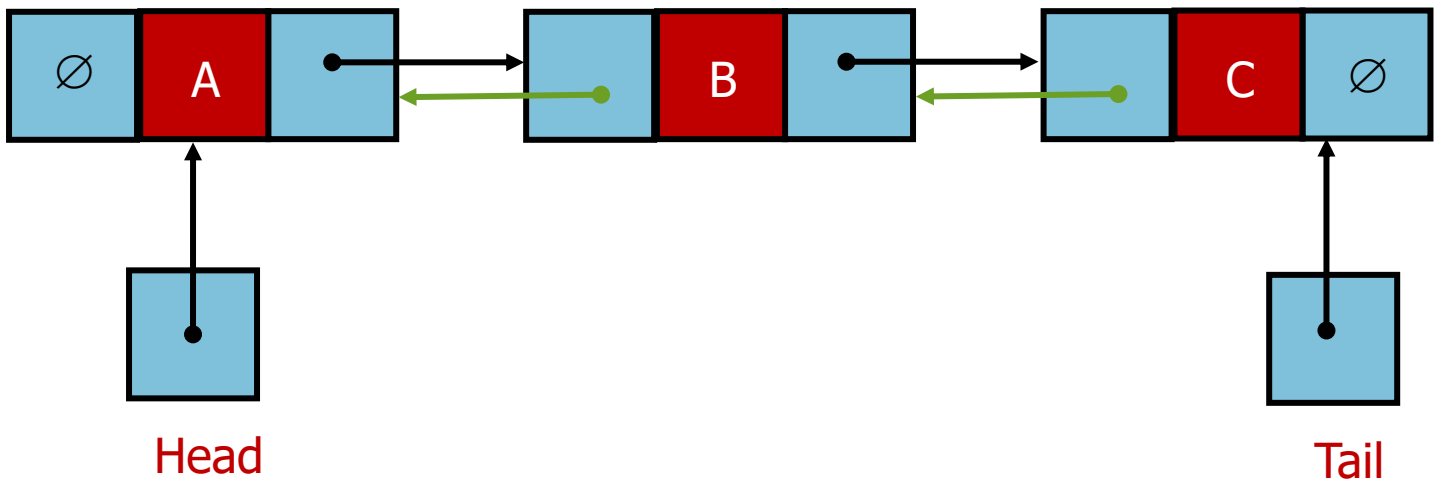


Each node points to not only successor but the predecessor

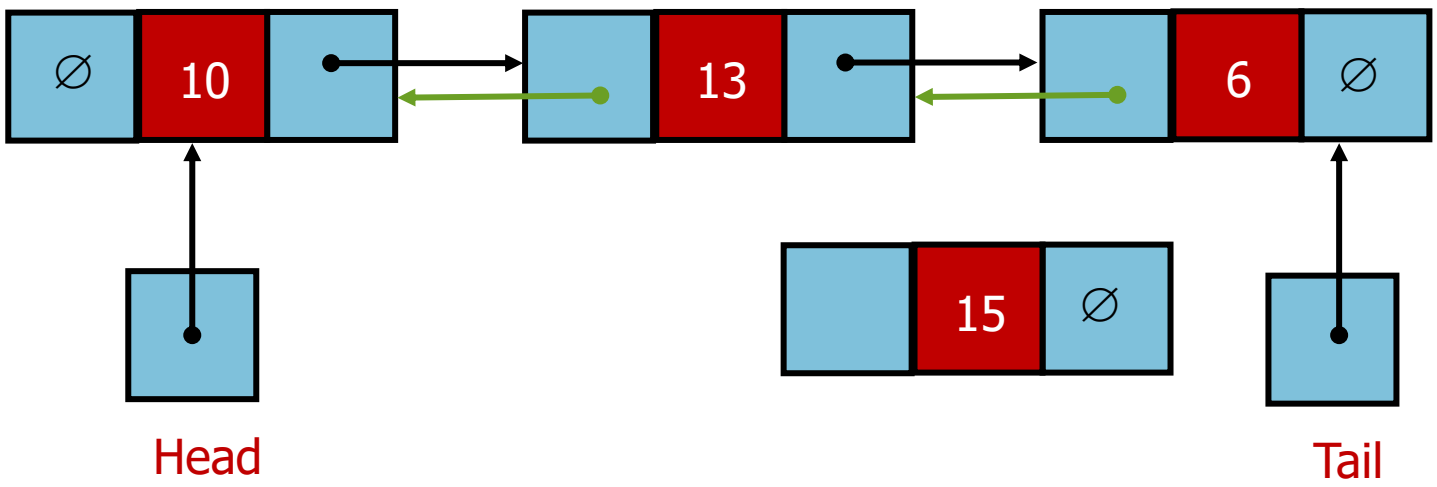
Doubly Linked Lists

- **There are two `NULL`:**
  - At the first and
  - At the last node
- **Advantage:**
  - It is easy to visit a predecessor.
  - Convenient to traverse lists backwards

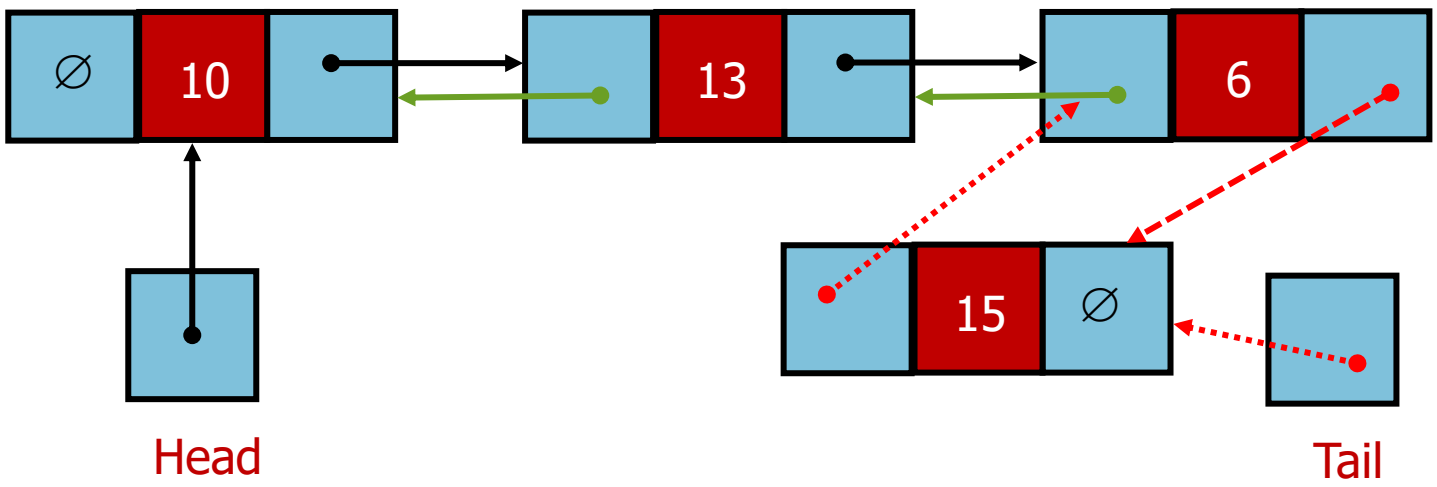
```
template<class T>
class DoublyLinkedList {
public:
    DoublyLinkedList() {
    head = tail = 0;
    }
    void addToDLLTail(const T&);
    T deleteFromDLLTail();

    private:
    DLLNode<T> *head, *tail;
};
```



DL Lists addToDLLTail

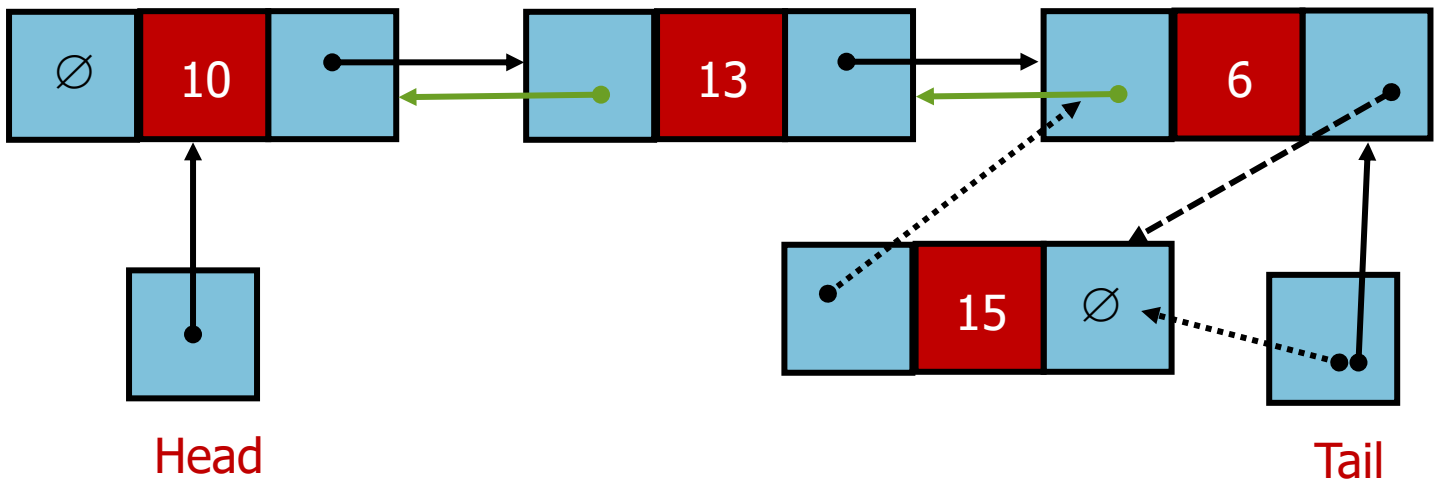


DL Lists addToDLLTail



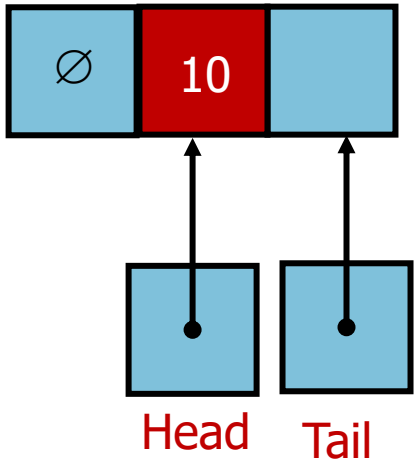
DL Lists addToDLLTail

```
template<class T>
void DLList<T>::addToDLLTail(const T& val) {
    if (tail != 0) {
        DLLNode<T> * newnode = new DLLNode<T>(val,tail, 0);
        \\ prev =tail, next=0
        tail->next = newnode;
        tail = newnode;
    }
    else
        head = tail = new DLLNode<T>(val);
}
```



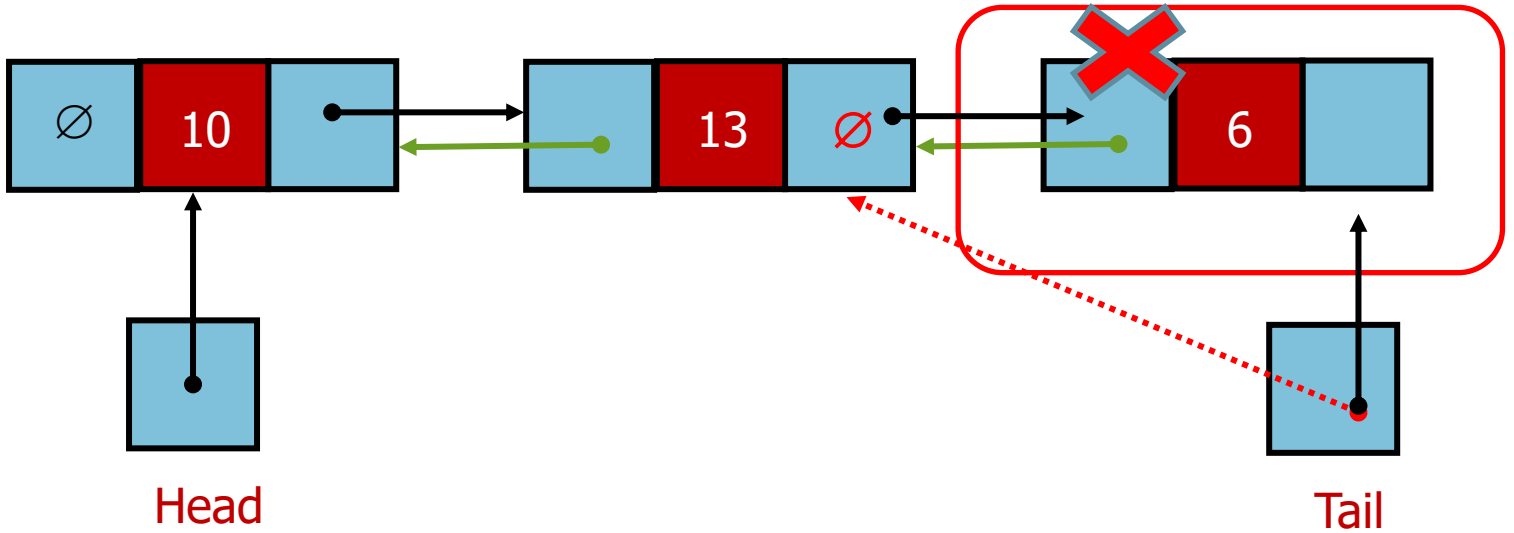
DL List deleteFromTail

```
template<class T>
void DLList<T>::deleteFromDLLTail() {
     if (head != NULL) {//not empty
          if (head == tail) { // if only one node in the list;
               delete head;
               head = tail = 0;
          }
          else { // if more than one node in the list;



          }
     }
}
```

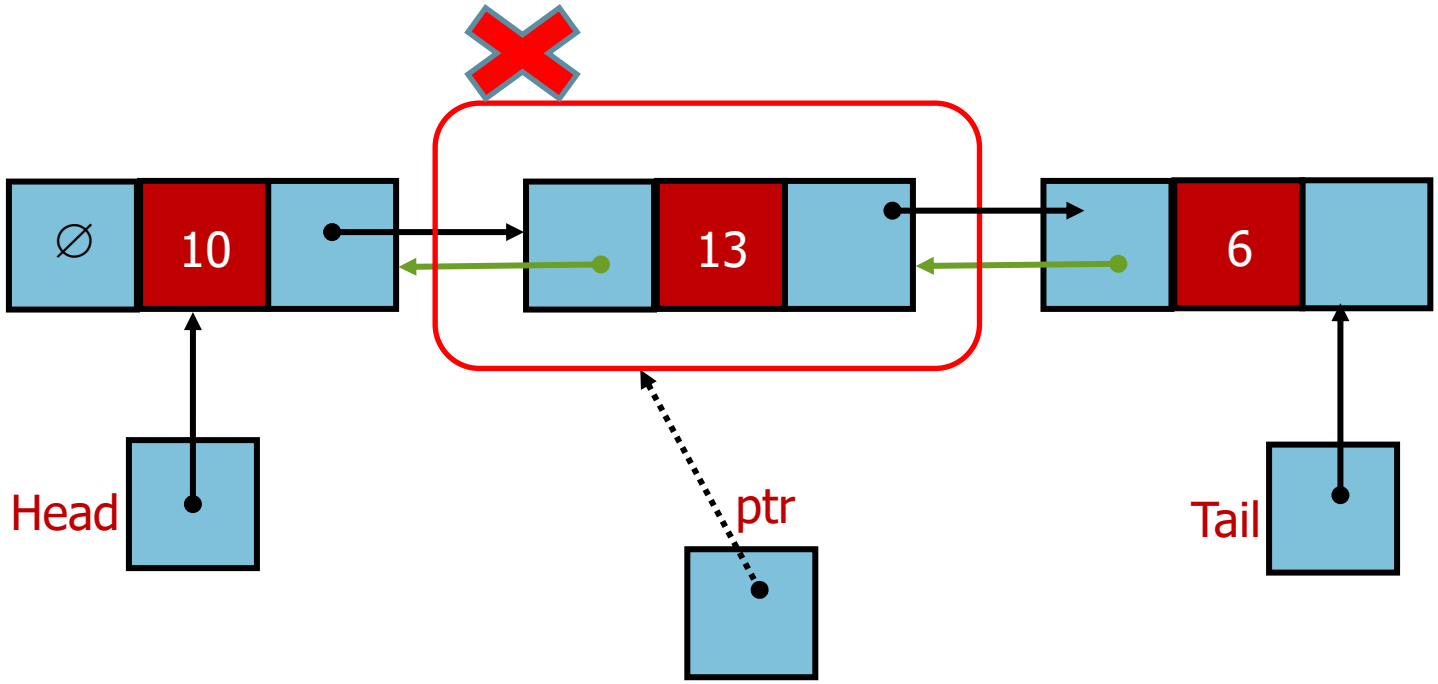


DL List deleteFromTail

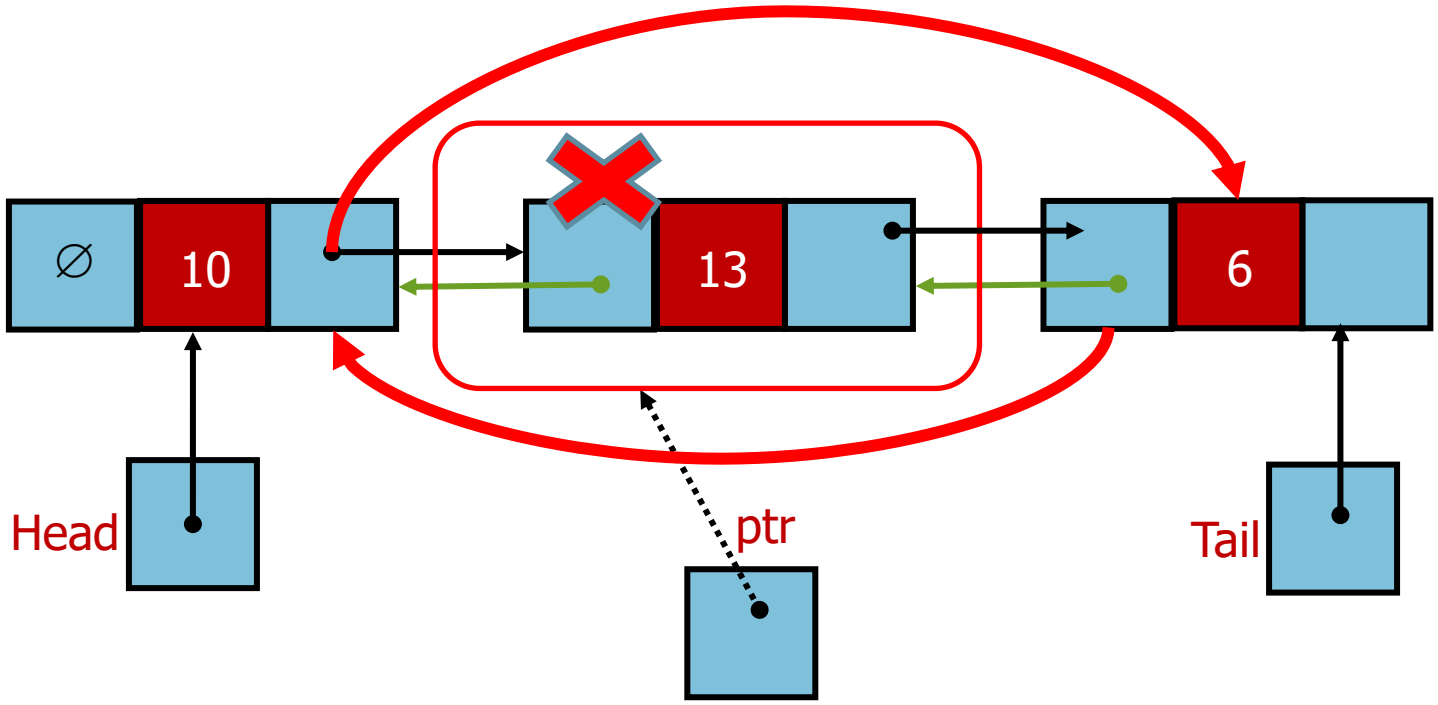
```
template<class T>
void DLList<T>::deleteFromDLLTail() {
     if (head != NULL) {//not empty
          if (head == tail) { // if only one node in the list;
               delete head;
               head = tail = 0;
          }
          else { // if more than one node in the list;
               tail = tail->prev;
               delete tail->next;
               tail->next = 0;
          }
     }
}
```



DL List deleteNode pointed by ptr

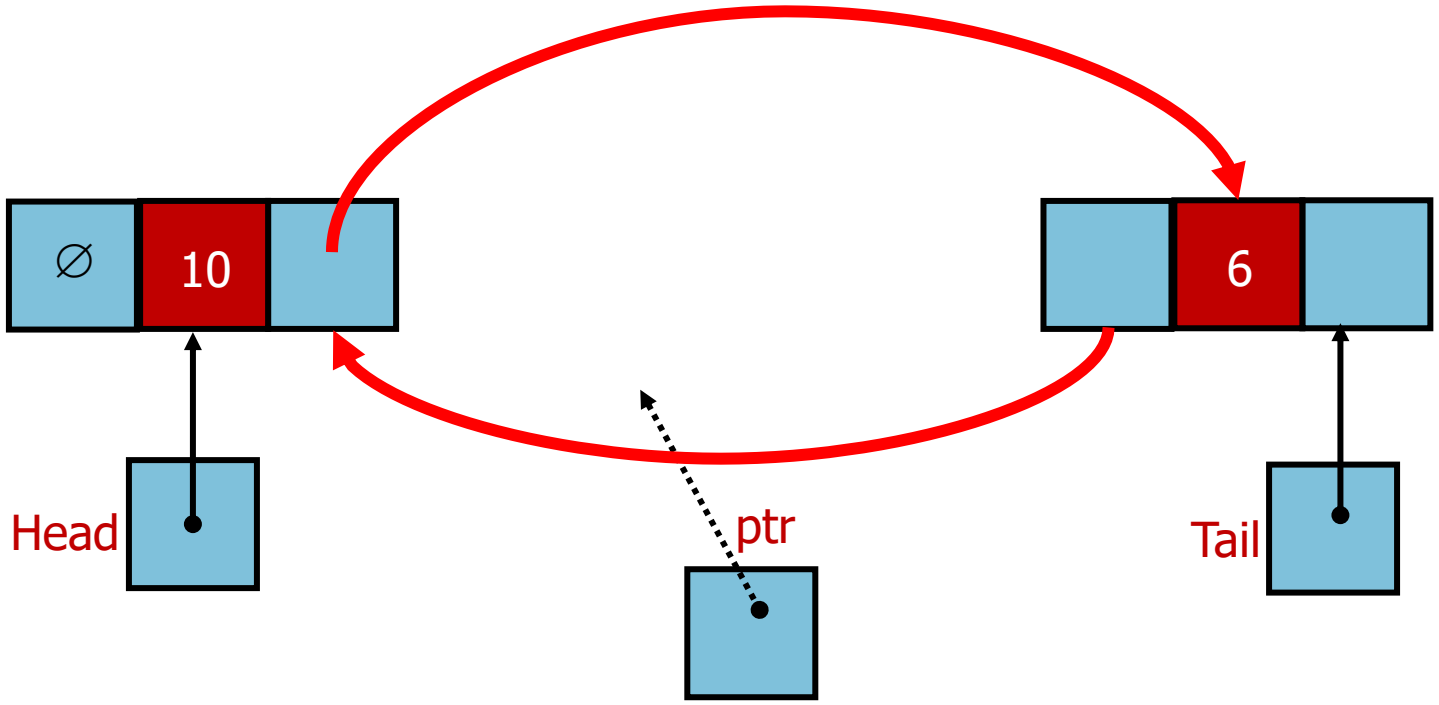


DL List deleteNode pointed by ptr




```
ptr->prev->next = ptr->next;

ptr->next->prev = ptr->prev;
```

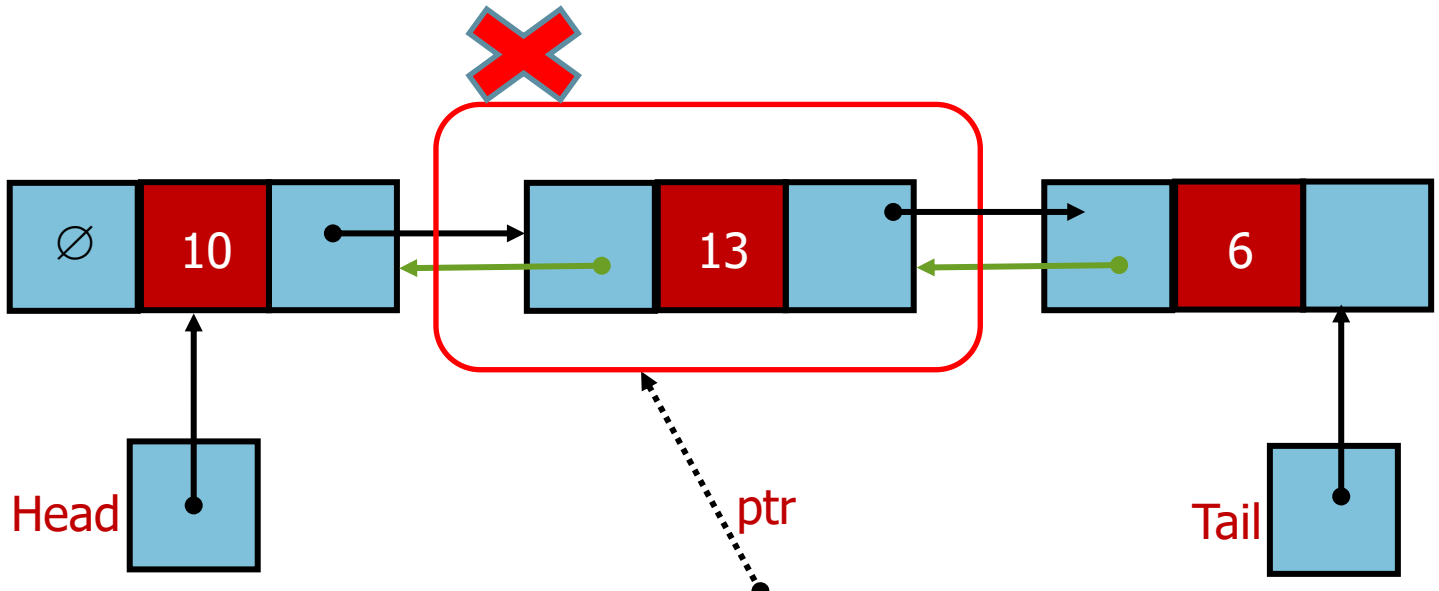
DL List deleteNode pointed by ptr




```
ptr->prev->next = ptr->next;

ptr->next->prev = ptr->prev;
```

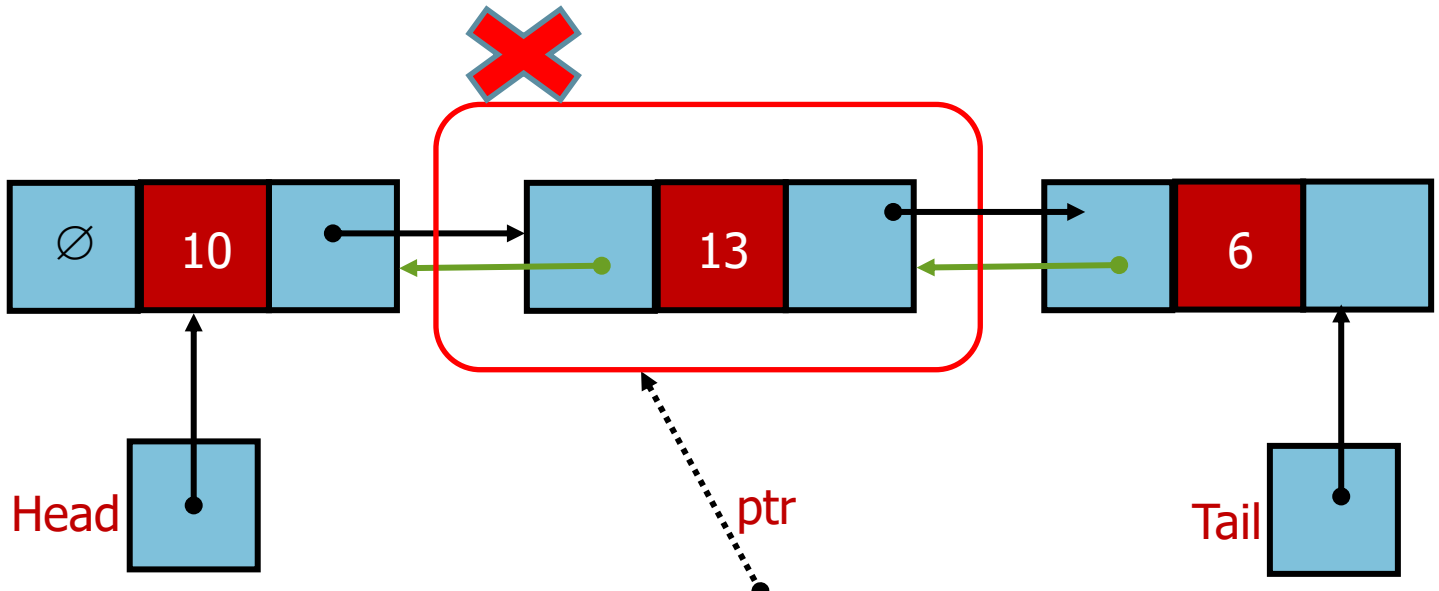
DL List deleteNode

```
template<class T>
void DLList<T>::deleteNode(DLLNode<T> * ptr) {
     if (head != NULL) {//not empty
          if (head == tail && head ==ptr){//if only one node in the list
               delete head;
               head = tail = 0;
          }
          else { // if more than one node in the list;




          }
     }
}
```



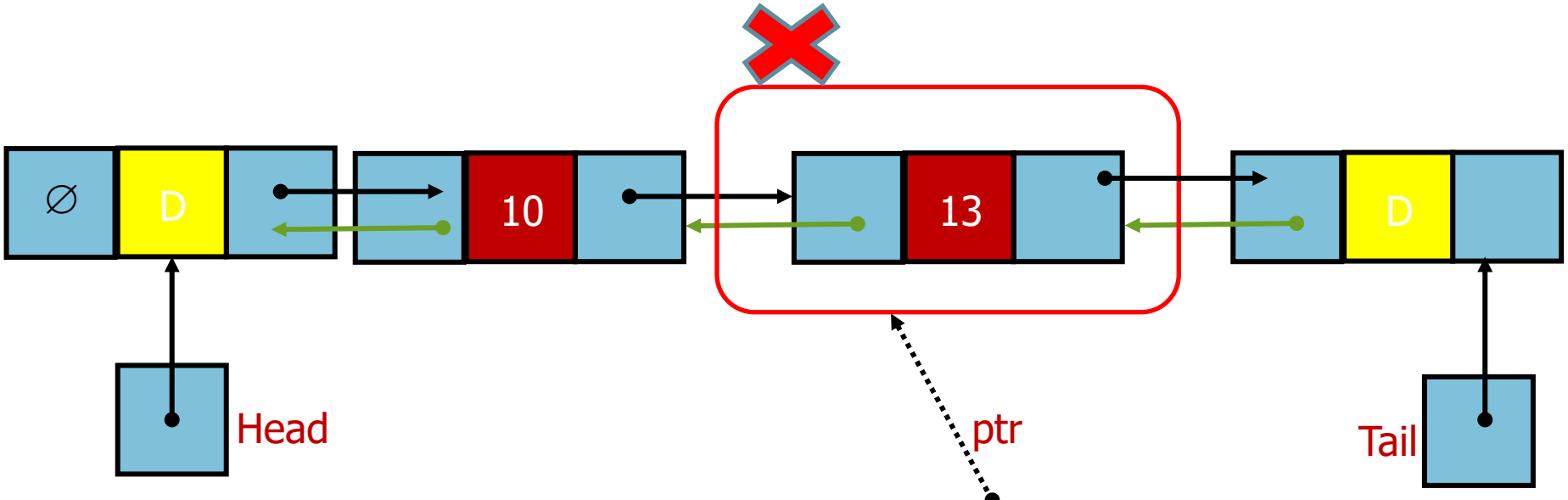
DL List deleteNode

```
template<class T>
void DLList<T>::deleteNode(DLLNode<T> * ptr) {
     if (head != NULL && ptr != NULL) {//not empty
          if (head == tail && head ==ptr){//if only one node in the list and delete it
               delete head;
               head = tail = 0;
          }
          else { // if more than one node in the list;
               if(ptr->prev != NULL)
                    ptr->prev->next = ptr->next; // not the first element
               else head = ptr->next; // the first element
               if(ptr->next != NULL)
                   ptr->next->prev = ptr->prev; // not the last element
               else tail = ptr->prev; // the last element
               delete ptr;
          }
     }
}
```



DLL deleteNode with Dummy

```
template<class T>
void DLList<T>::deleteNode(DLLNode<T> * ptr) {
    if (ptr != NULL && !Empty()) {//not empty

        ptr->prev->next = ptr->next;
        ptr->next->prev = ptr->prev;
        delete ptr;

    }
}
```



To Do DL List

- Implement the following functions
  - Delete node with particular data value
  - Destructor
  - Reverse the SL List
  - Remove duplicates in the list
  - Sort the list
  - Merge two sorted lists
  - Remove the given element
  - Remove all occurrences of the given element

  - TO DO SL List
  - TO DO CL List

TO DO

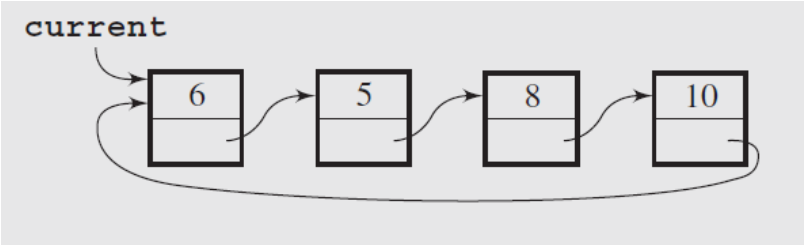
- Create a singly linked list of objects
  - STUDENT
    - Roll-no
    - Name
    - Address
- What need to be changed in the class methods?
- Do you need copy constructor?
- You should avoid creating unnecessary copies of objects
  - Use const &
  - Or you can keep a pointer to an object if it is needed in multiple List
    - Student may be needed in courses, library and account lists

```
template<class type>
void List<type>::addToStart(const type & val) {
    head = new Node<type>(val, head);
    if (tail == 0)tail = head;
}
```

CIRCULAR LINKED LIST

CIRCULAR LINKED LIST

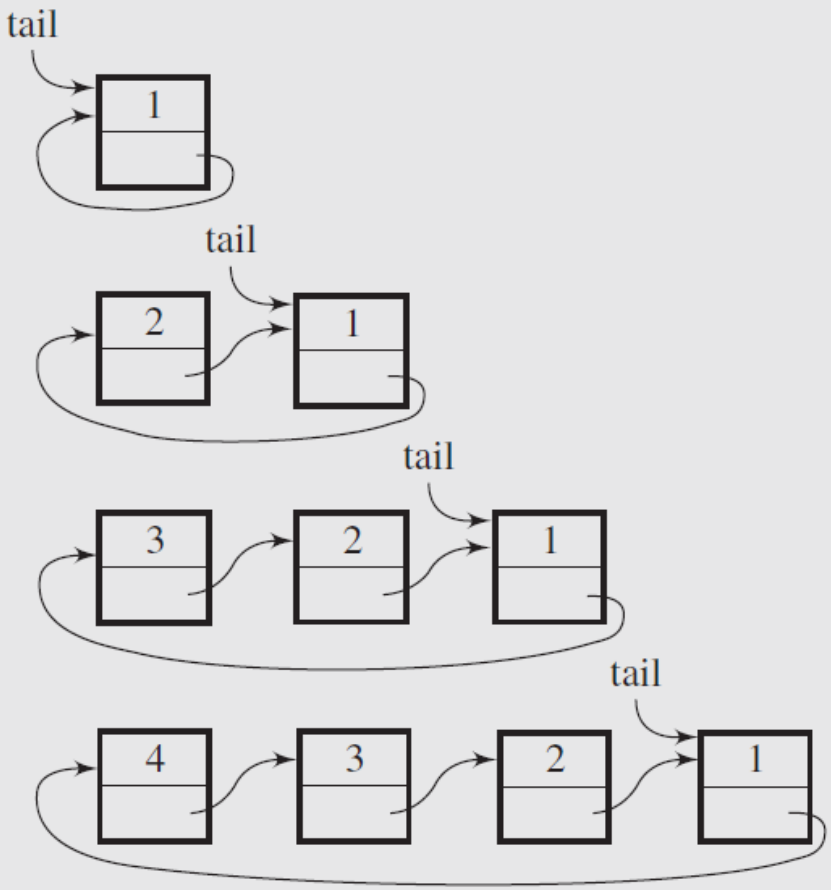
- In a *circular list* nodes form a ring:
  - the list is finite and
  - each node has a successor.



- Real World Example
  - when several processes are using the same resource for the same amount of time, and we have to ensure that each process has a fair share of the resource.
  - All processes are put on a circular list accessible through the pointer current.
  - After one node in the list is accessed and the process number is retrieved from the node to activate this process,
  - current moves to the next node so that the next process can be activated the next time.

SINGLY CL LIST

FIGURE 3.14 Inserting nodes (a) at the front of a circular singly linked list and (b) at its end.

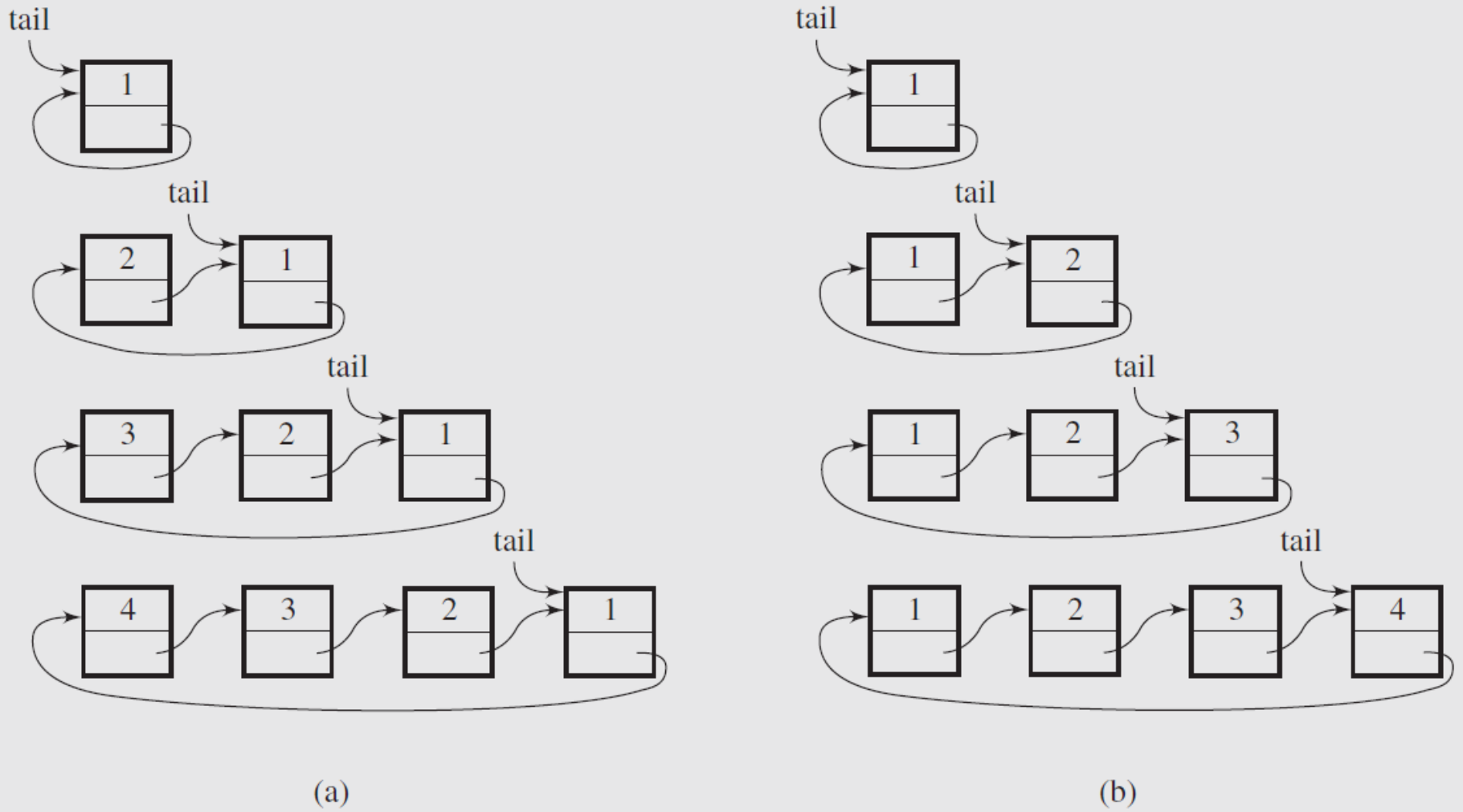


(a)

(b)

SINGLY CL LIST

FIGURE 3.14 Inserting nodes (a) at the front of a circular singly linked list and (b) at its end.



Insert at tail Singly CLList

```
void addToTail(int el) {
    if (isEmpty()) {
        tail = new Node(el);
        tail->next = tail;
    }
    else {
        tail->next = new Node(el, tail->next);
        tail = tail->next;
    }
}
```