LINKED LISTS

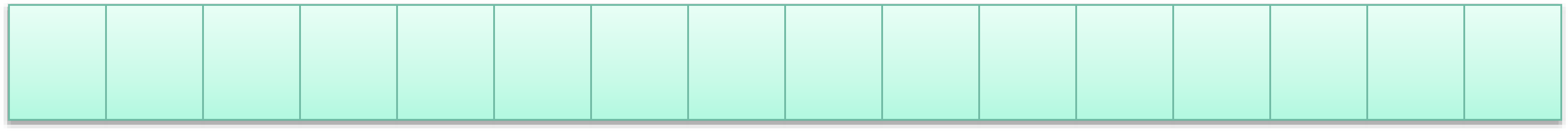
Singly Linked lists

Doubly linked lists

Circular linked lists

Issues with Arrays

- Array is no doubt a very useful data structure that provide fast access to elements

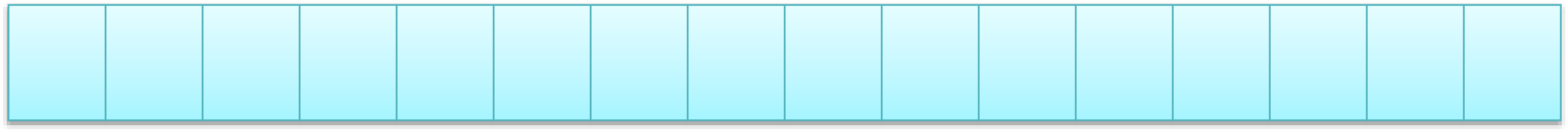


- But it has its limitation
  - Array is allocated contiguous memory, so insertion and deletion is a nightmare
    - Takes O(N) time for each Operation
  - What if Array size **grow** ?
    - In some case size has to be known at compilation time
    - We can declare dynamic arrays at runtime

This limitation can be overcome by using *linked structures.*

Why Linked List?

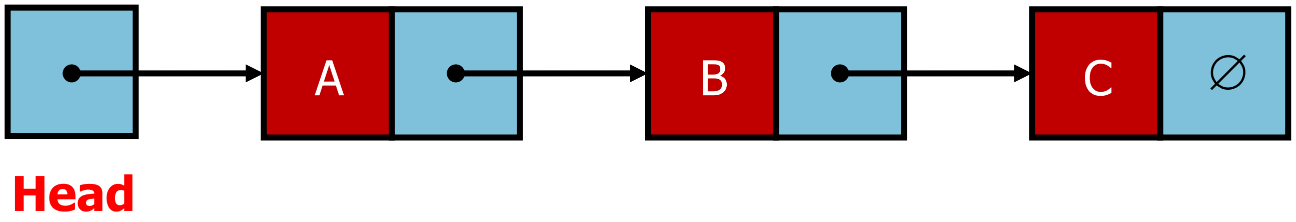
- Array is not useful
  - In case insertion and deletion is very common and
  - If the data has to be processed in a sequential order.



- If data is hardly processed randomly then
  - we can eliminate the need for contiguous memory
  - And store data elements at different places in the heap
  - Devise some mechanism to move from one element to the next.

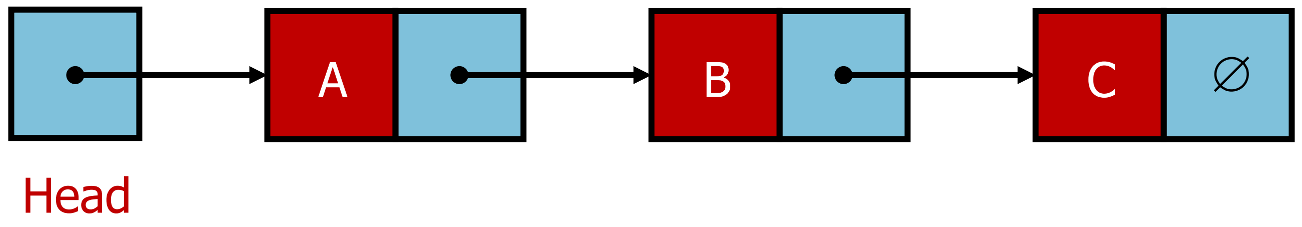
Arrays → Linked Lists

- So all we need is
  - a **starting point** and
  - a **link from one element to the next.**
    - is accomplished by storing the address



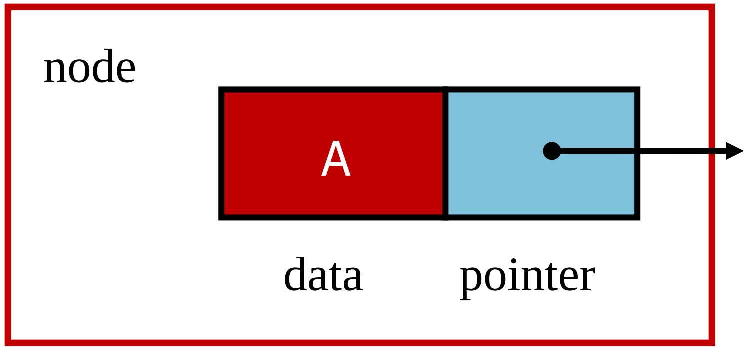
This gives the notion of **logical adjacency** as opposed to **physical adjacency**.

Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`

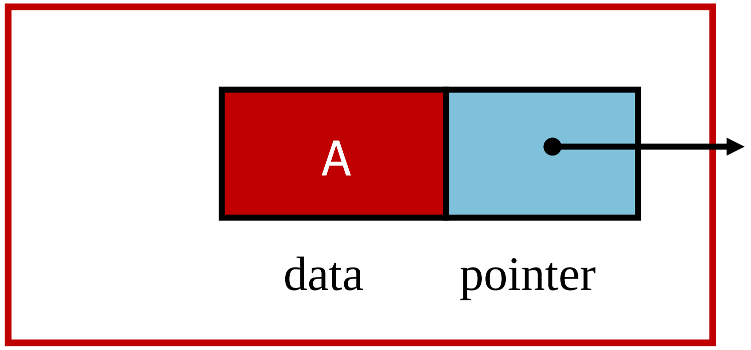
We can only travel in the direction of the link.



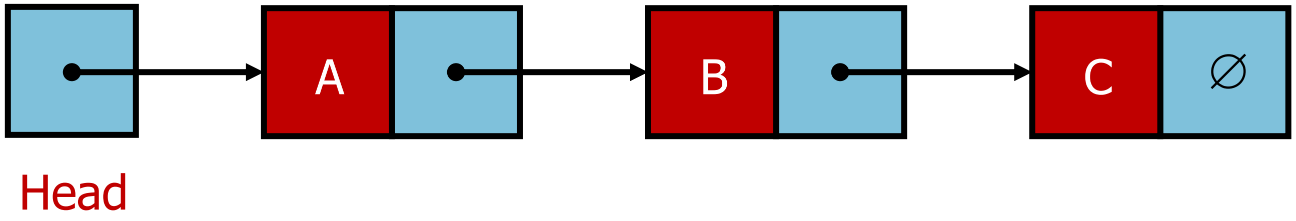
NODE Class

We need two classes:

Node struct



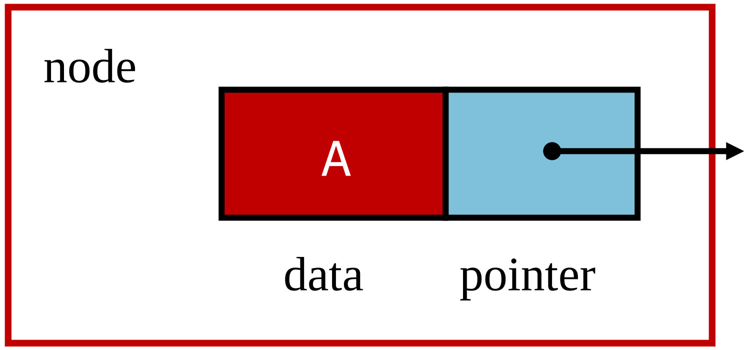
and
List class



NODE Class

- We need two things: **`Node`** and **`List`**
- Declare `Node` struct for the nodes
  - `data`: **`int`**-type data in this example
  - `next`: a pointer to the next node in the list

```
struct Node {
public:
   int data;
   Node *next;
};
```



NODE Class

- We use two classes: **Node** and **List**

```
struct Node {
public:
    int data;
    Node *next;
};
```

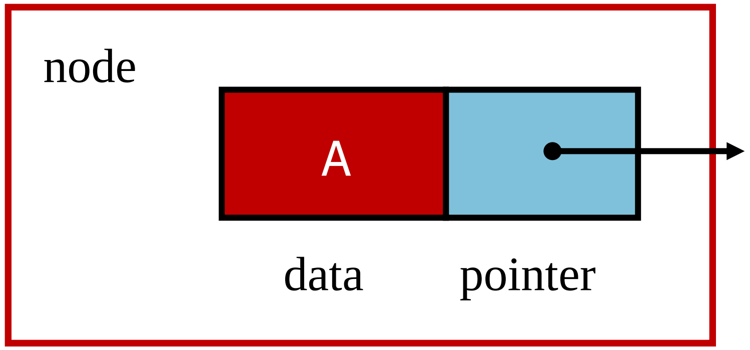
Issues ?

1. Missing Constructor
   - Next should be set to **NULL**
2. Do we need a node class for each kind of data item ? Any Easy way out …???
   - **Templates**

- Data is public …anyone can access
- If a func return a pointer to any list node then in main we can access list node data and nextptr
  - **Use friend class** concept
  - Nested class concept
  - Getter and setter concept

A **friend class** is a class that can access the private and protected members of a class in which it is declared as **friend**. This is needed when we want to allow a particular class to access the private and protected members of a class.

Node Class



```
struct Node {
public:
    int data;
    Node *next;
};
```

```
struct Node {
public:
    Node() { next = NULL; }
    Node(type val, Node<type> * nptr = 0) {
            data = val;
            next = nptr;
    }

    type data;
    Node * next;

};
```

NESTED CLASS

Nested Classes

- A nested class is a class which is declared in another enclosing class.

```
class outer {
private:

    class inner {
        public:
        private:

    };

public:

};
```

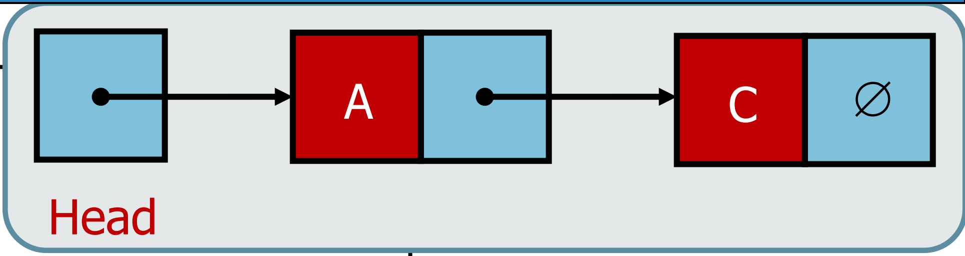
A nested class is a member and has the same access rights as any other member.

- The nested class ***can access public and private members*** of outer class *(if it create an object of outer class in it)*

The members of an enclosing class have no special access to members of a nested class.

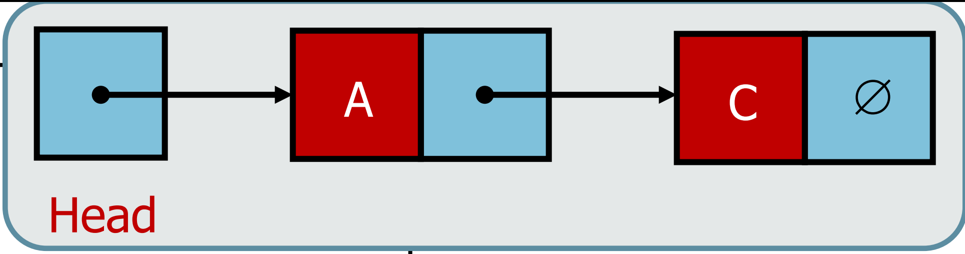
- The outer class **can only access public methods** of inner class

List Class



```
template<class type>
class List {
public:
    List() { head = 0; };
    ~List() ;
private:
    struct Node {
        public:
        Node() { next = NULL; }
        Node(type val, Node * nptr = 0) {
            data = val;
            next = nptr;
        }
        type data;
        Node * next;
    };

    Node * head;
};
```

List Class



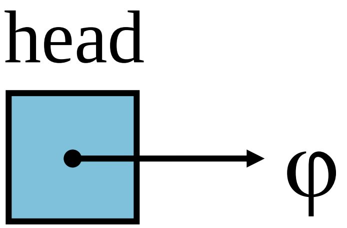
```
template<class type>
class List {
public:
    List() { head = 0; };
    ~List() ;
private:
   struct Node; // forward declaration
   Node * head;
};
```

```
template<class type>
struct List<type>::Node {
public:
    Node() { next = NULL; }
    Node(type val, Node * nptr = 0){
        data = val;
        next = nptr;
     }
    type data;
    Node * next;
};
```

Operations on `List` Class

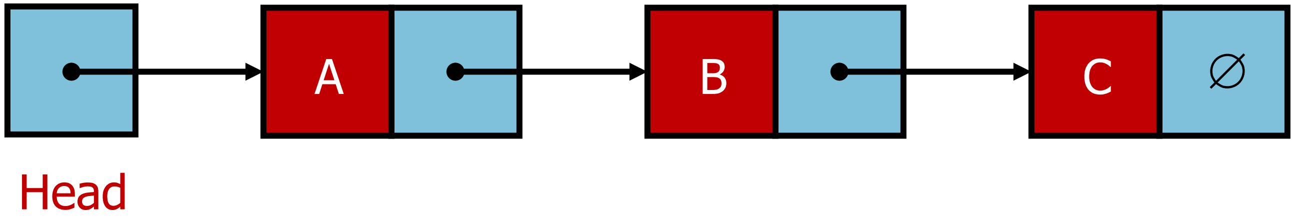
- `List` contains **head:** a pointer to the first node in the list.
  Since the list is empty initially, `head` is set to `NULL`

```
template<class type>
class List {
public:
    List() { head = 0; };
    ~List();
private:
     struct Node; // forward declaration
     Node * head;

};
```

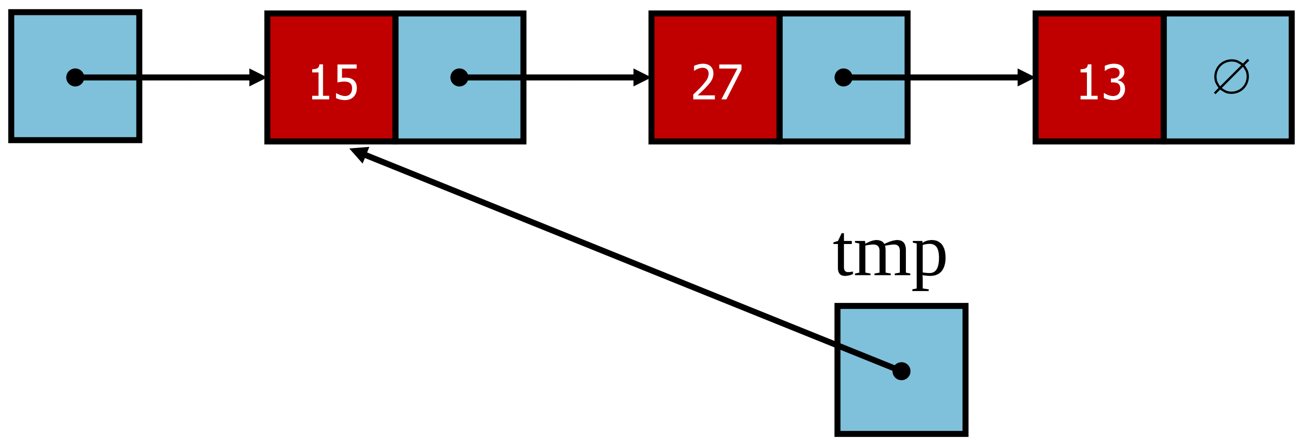


Is Empty

```
bool IsEmpty() {
    return head == 0;
}
```

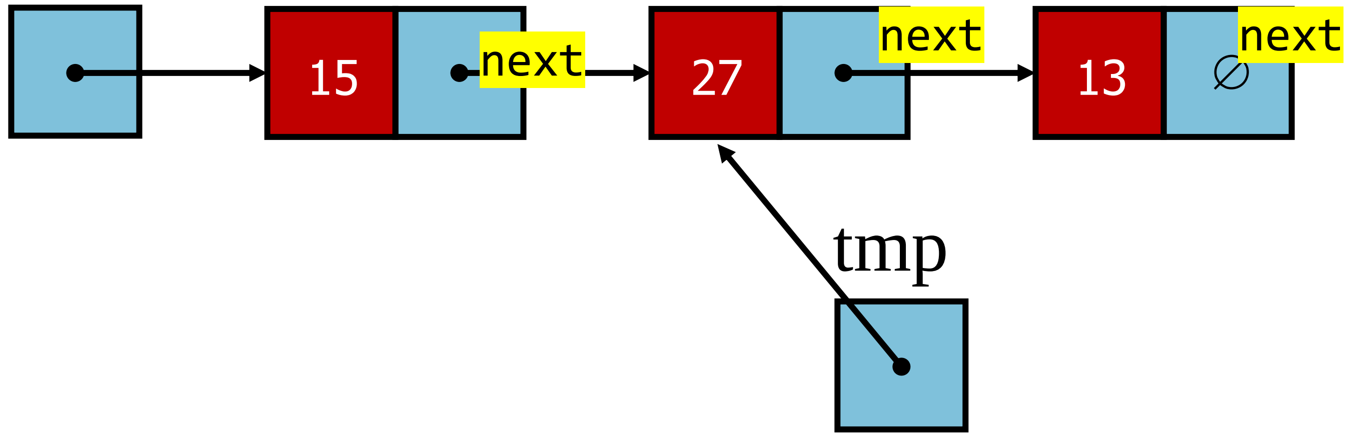


Find a data value in the List



```
template<class type>
bool List<type>::Find(type val) {
    Node * tmp = head;
    while (tmp != NULL && tmp->data != val)
        tmp = tmp->next;

    return tmp != NULL;
}
```

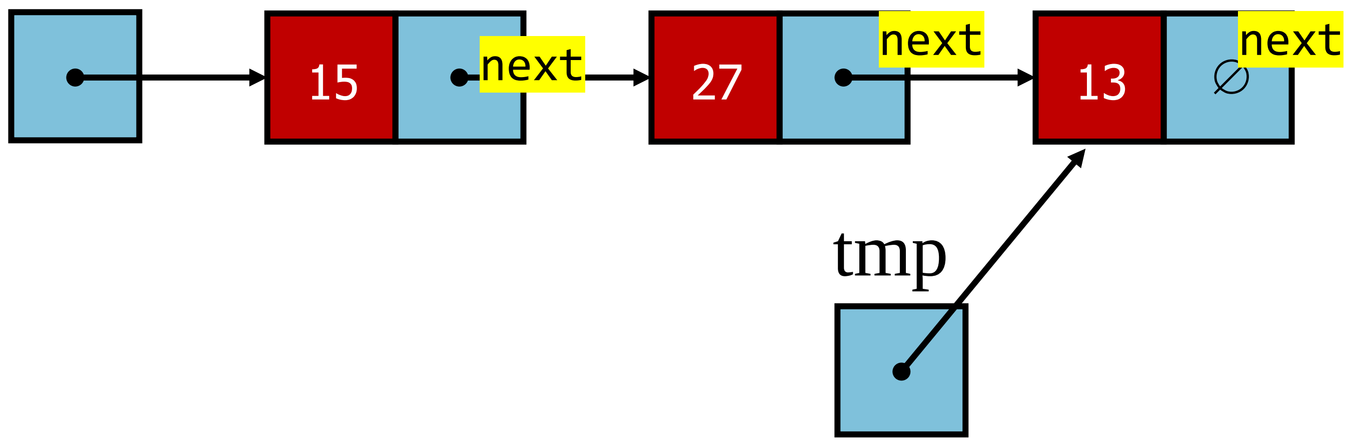
Find a node (data value) in List



```
template<class type>
bool List<type>::Find(type val) {
    Node * tmp = head;
    while (tmp != NULL && tmp->data != val)
        tmp = tmp->next;

    return tmp != NULL;
}
```

```
template<class type>
struct List<type>::Node {
public:
    Node(){next=NULL; }
    type data;
    Node * next;
};
```

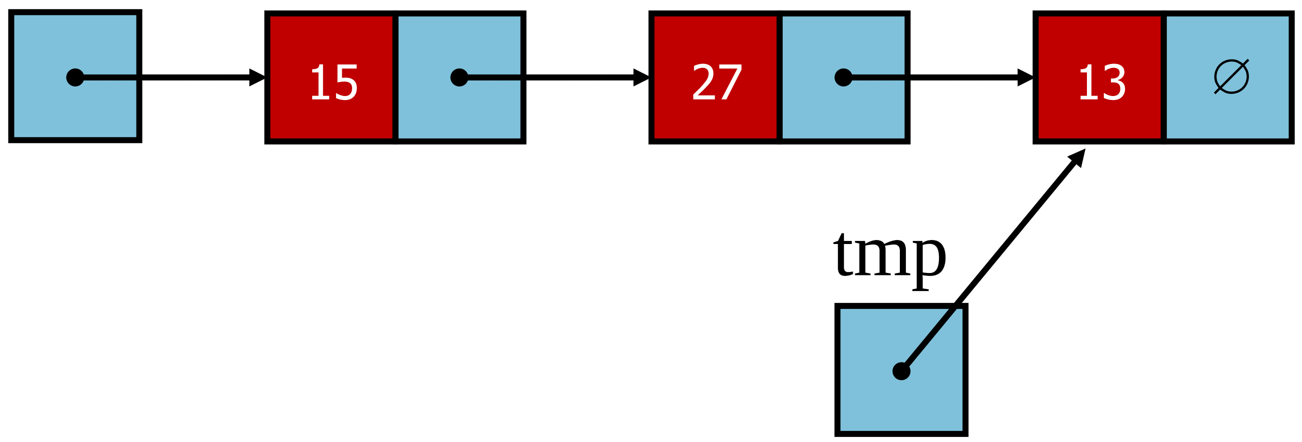
Find a node (data value) in List



```
template<class type>
bool List<type>::Find(type val) {
    Node * tmp = head;
    while (tmp != NULL && tmp->data != val)
        tmp = tmp->next;

    return tmp != NULL;
}
```

```
template<class type>
struct List<type>::Node {
public:
    Node(){next=NULL; }
    type data;
    Node * next;
};
```

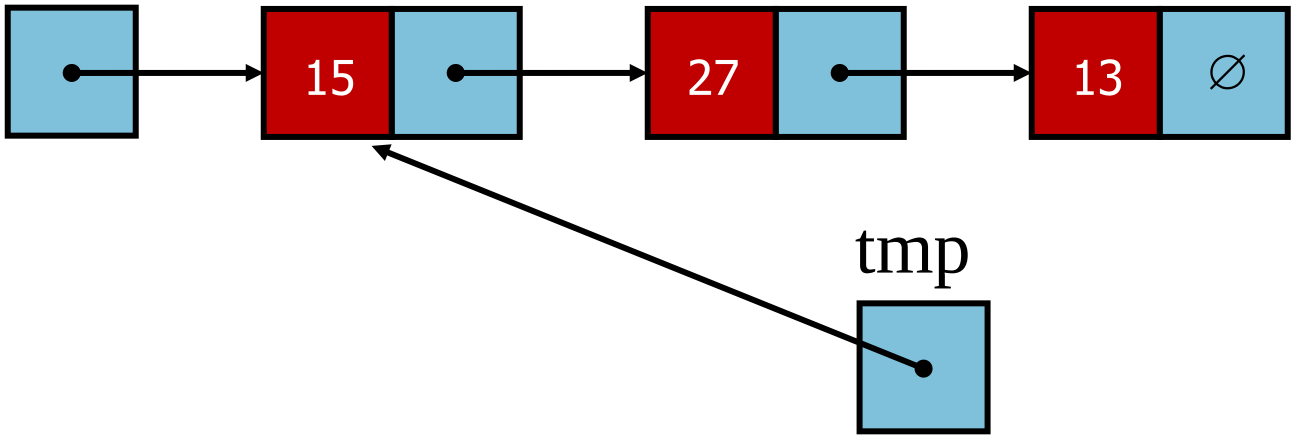
Find a node (data value) in List



```
template<class type>
bool List<type>::Find(type val) {
    Node * tmp = head;
    while (tmp != NULL && tmp->data != val)
        tmp = tmp->next;

    return tmp != NULL;
}
```

Takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases

Print SL List



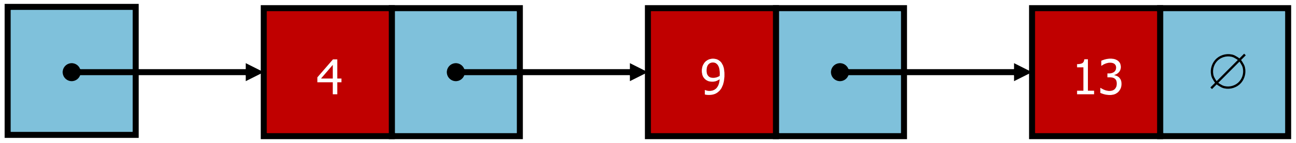
```
template<class type>
void List<type>::print() {
    Node * tmp;
    for (tmp = head; tmp != 0; tmp = tmp->next)
        cout << tmp->data << "  ";
    cout << endl;
}
```

Takes $O(n)$ time

SL List AddNode

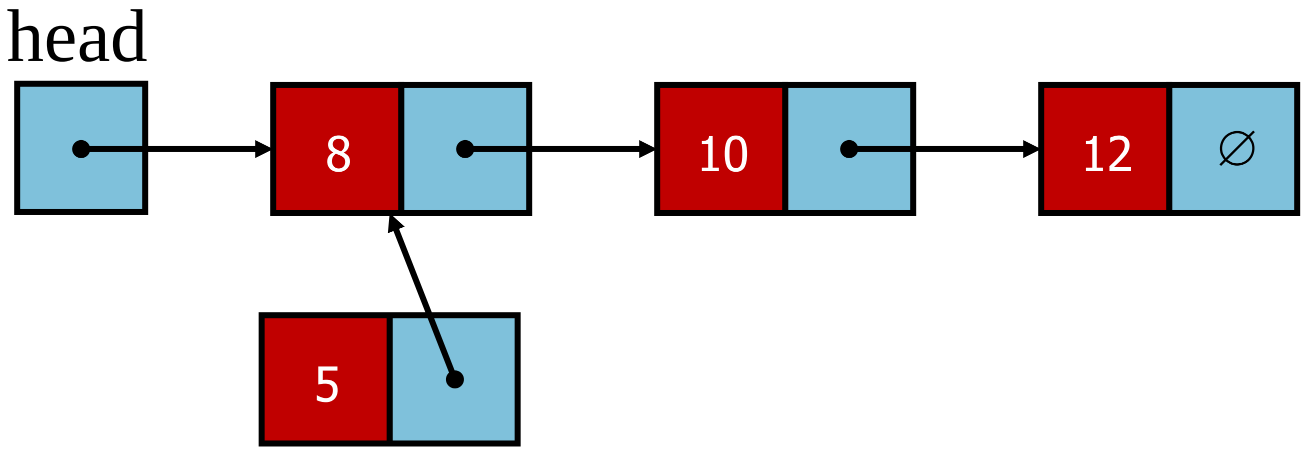
Let's implement some basic operations in class **List**

- **Add Node**
- **Where to** add Node
  - Start of the list
  - End of the list
  - Some where in the middle …after some particular data value (or in sorted list)
- **Which is most efficient ?**
- We provide all the options let user decide which to use



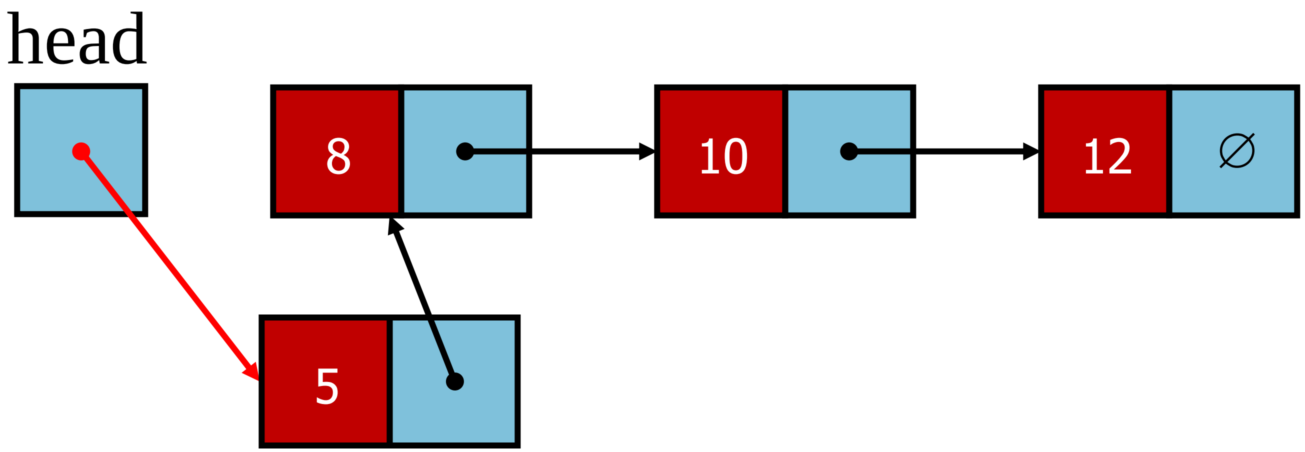
SL List AddNode

AddNode at start



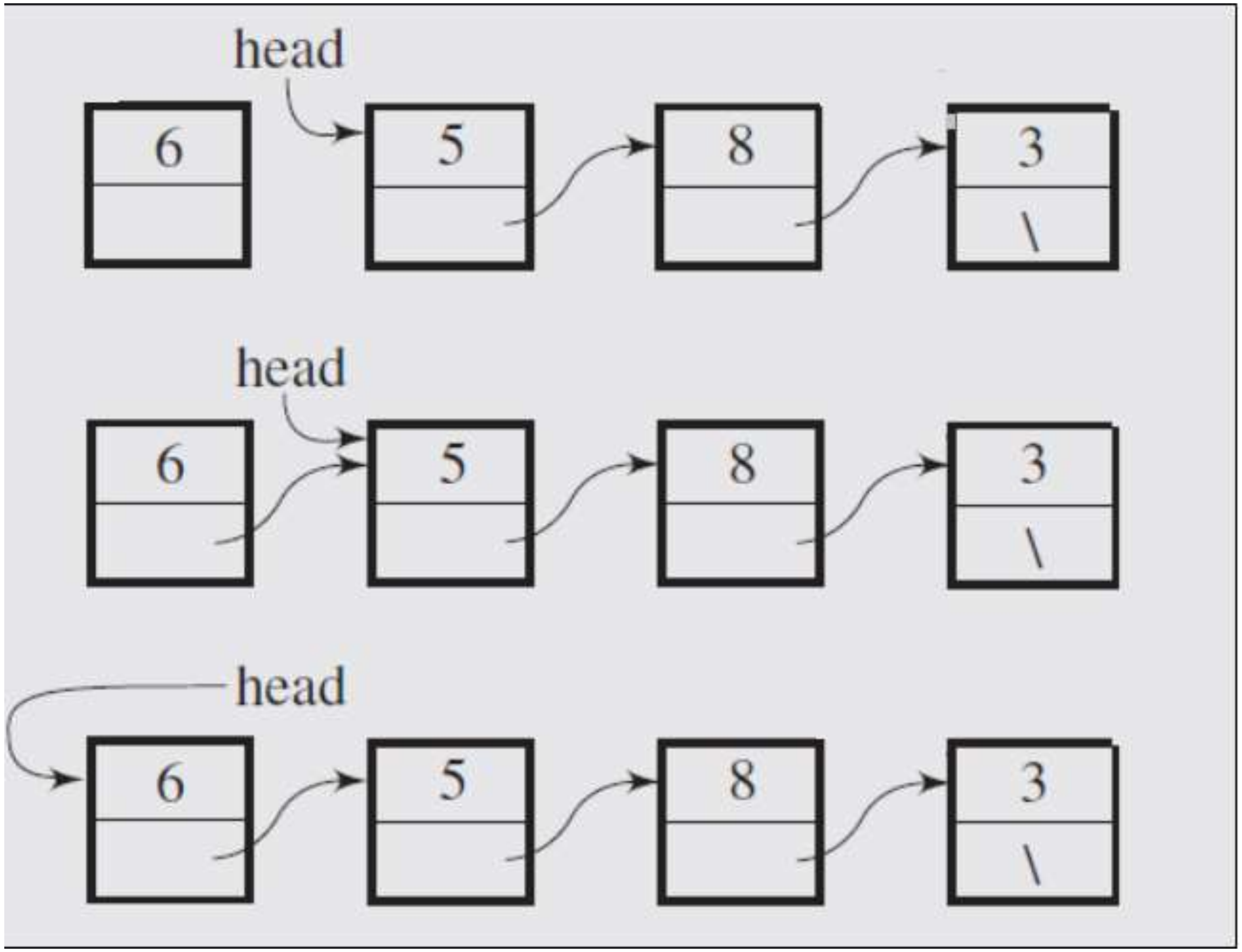
SL List AddNode

AddNode at start



SL List AddNode

AddNode at start

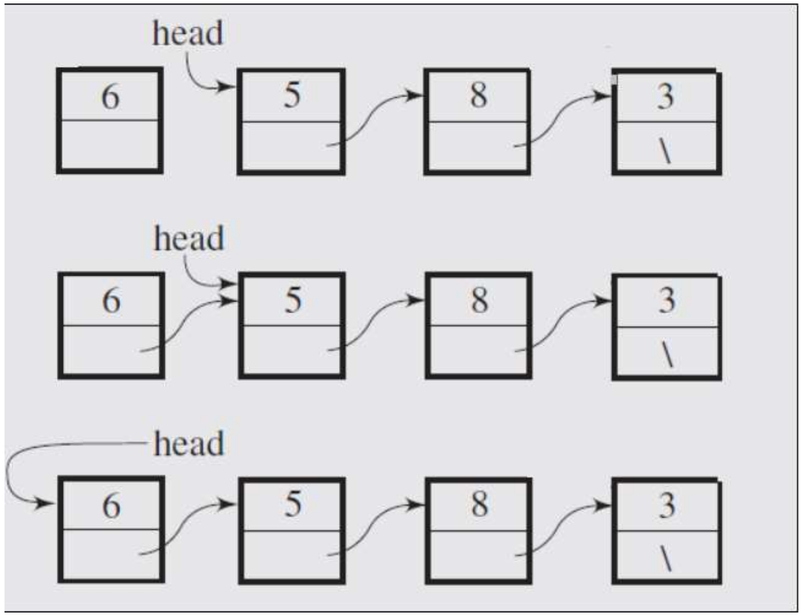


AddNode at start

```
template<class type>
void List<type>::addtoHead(type val)
{
    head = new Node(val, head);
}
```

TIME COMPLEXITY ?

```
template<class type>
struct List<type>::Node {
public:
    Node() { next = NULL; }
    Node(type val,Node * nptr = 0){
        data = val;
        next = nptr;
     }
    type data;
    Node * next;
};
```



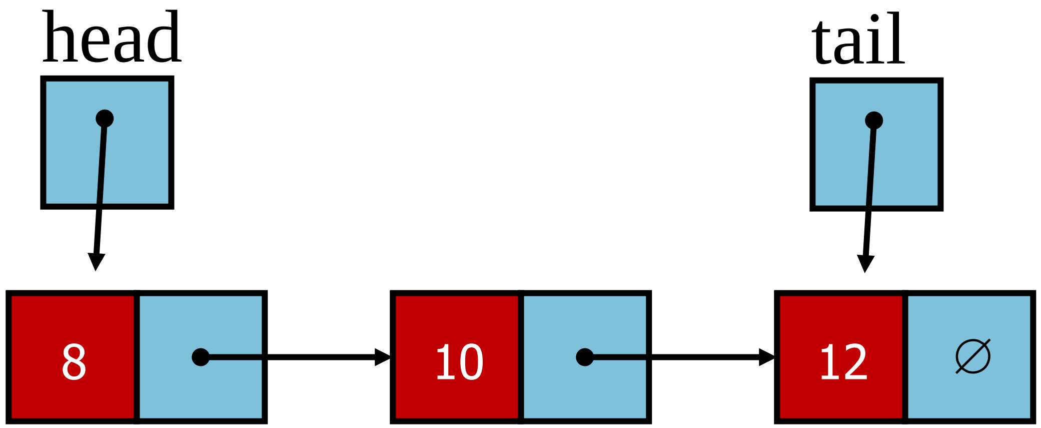
AddNode at start

```
template<class type>
class List {
public:
   List() { head = 0;
            tail= 0; };
   ~List() ;

   void addToStart(type val);

private:
      struct Node; // forward declaration
      Node * head;
      Node * tail;
};
```

AddNode at End

```
Not a good idea to traverse the entire list and insert
Keep a pointer to the tail of the list.
```

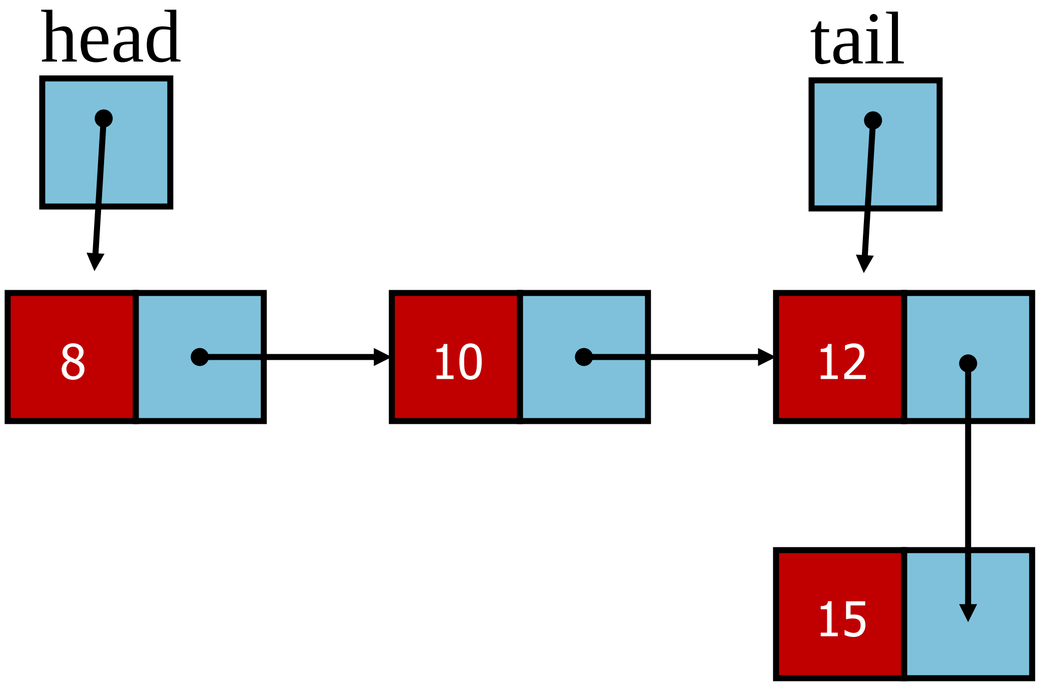
```
template<class type>
class List {
public:
    List() { head = 0;
                tail= 0; };
    ~List() ;

    void addToStart(type val);
    void addToTail(type val);

private:
        struct Node;
        Node * head;
        Node * tail;
};
```



AddNode at End

```
Not a good idea to traverse the entire list and insert
Keep a pointer to the tail of the list.
```

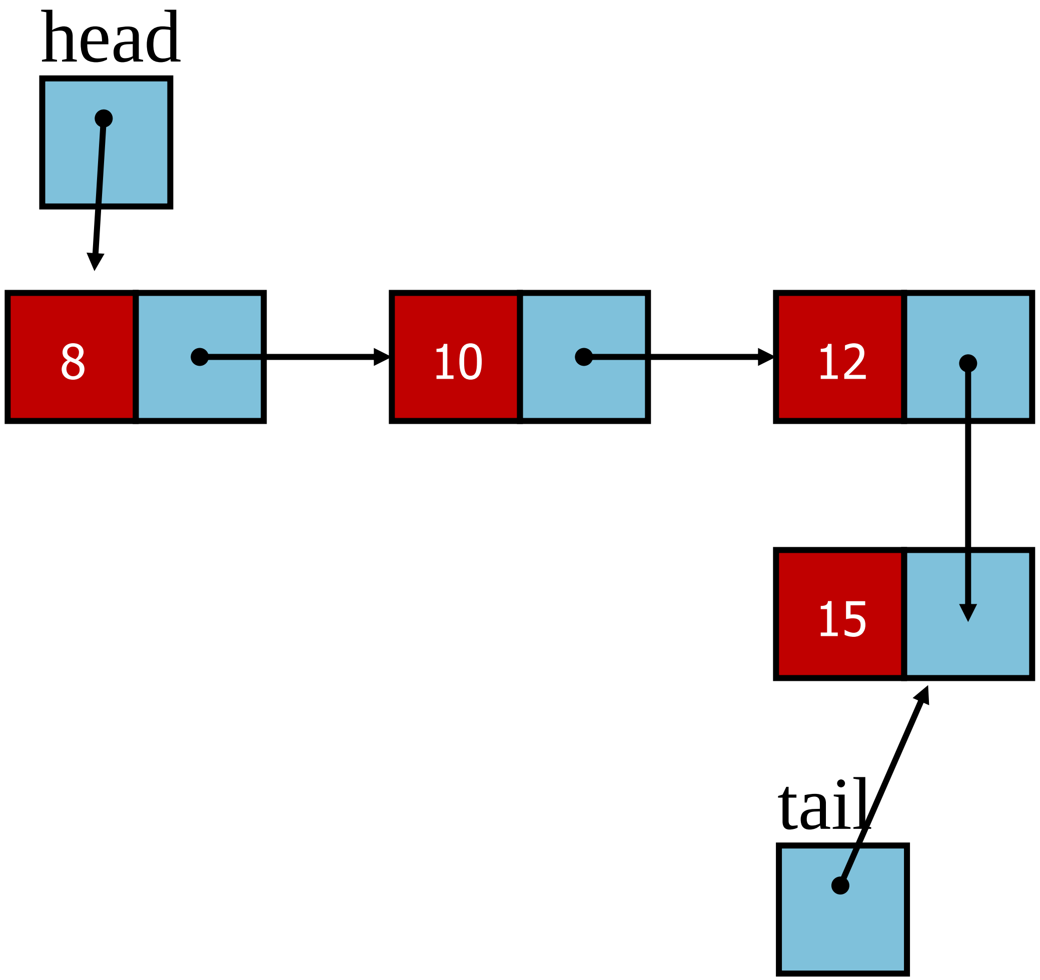
```
template<class type>
class List {
public:
    List() { head = 0;
                tail= 0; };
    ~List() ;

    void addToStart(type val);
    void addToTail(type val);

private:
        struct Node;
        Node * head;
        Node * tail;
};
```



AddNode at End

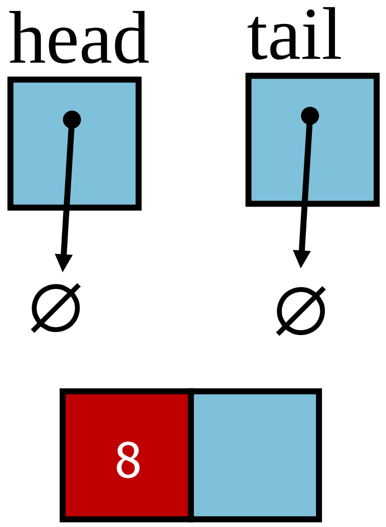
```
Not a good idea to traverse the entire list and insert
Keep a pointer to the tail of the list.
```

```
template<class type>
class List {
public:
    List() { head = 0;
             tail= 0; };
    ~List() ;

    void addToStart(type val);
    void addToTail(type val);

private:
        struct Node;
        Node * head;
        Node * tail;
};
```



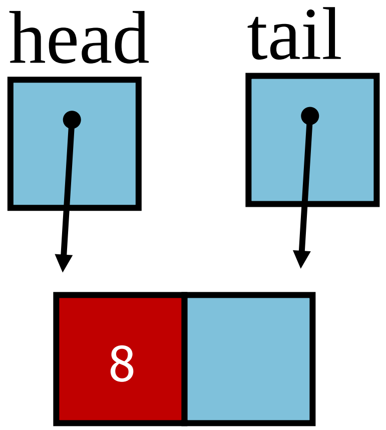
AddNode at End

```
template<class type>
void List<type>::addToTail(type val) {
    if (tail != NULL) {

    }
    else
        head = tail= new Node( val);
}
```



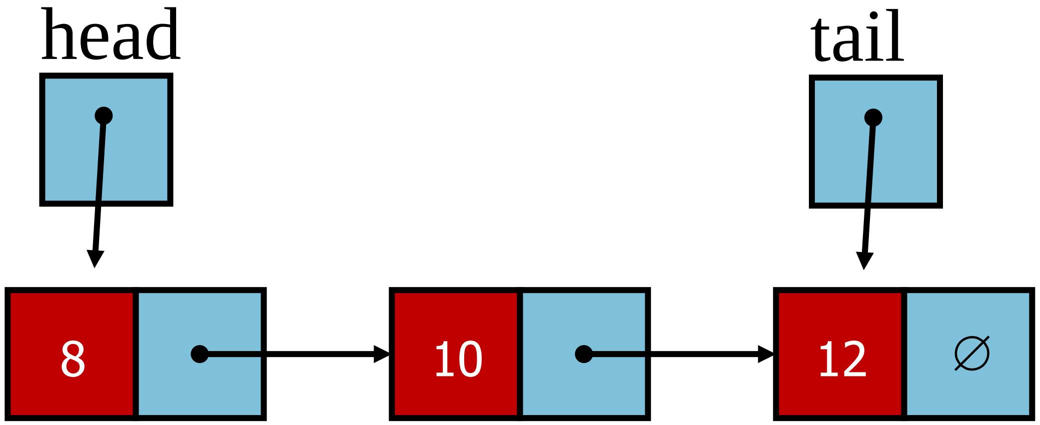
AddNode at End

```
template<class type>
void List<type>::addToTail(type val) {
    if (tail != NULL) {
    }
    else
        head = tail= new Node( val);
}
```



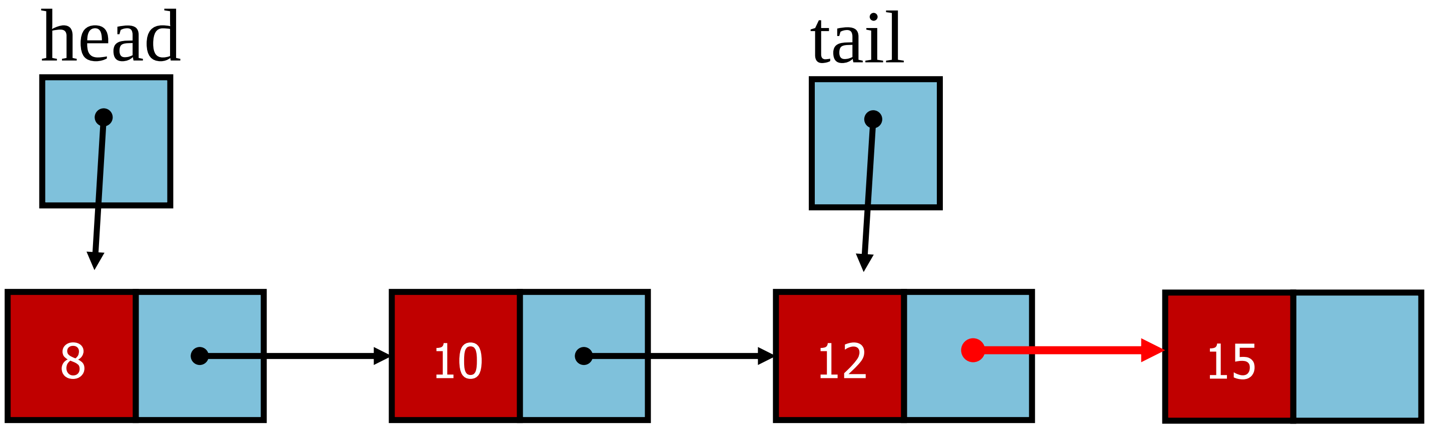
AddNode at End

```
template<class type>
void List<type>::addToTail(type val) {
    if (tail != NULL) {
    }
    else
        head = tail= new Node( val);
}
```



AddNode at End

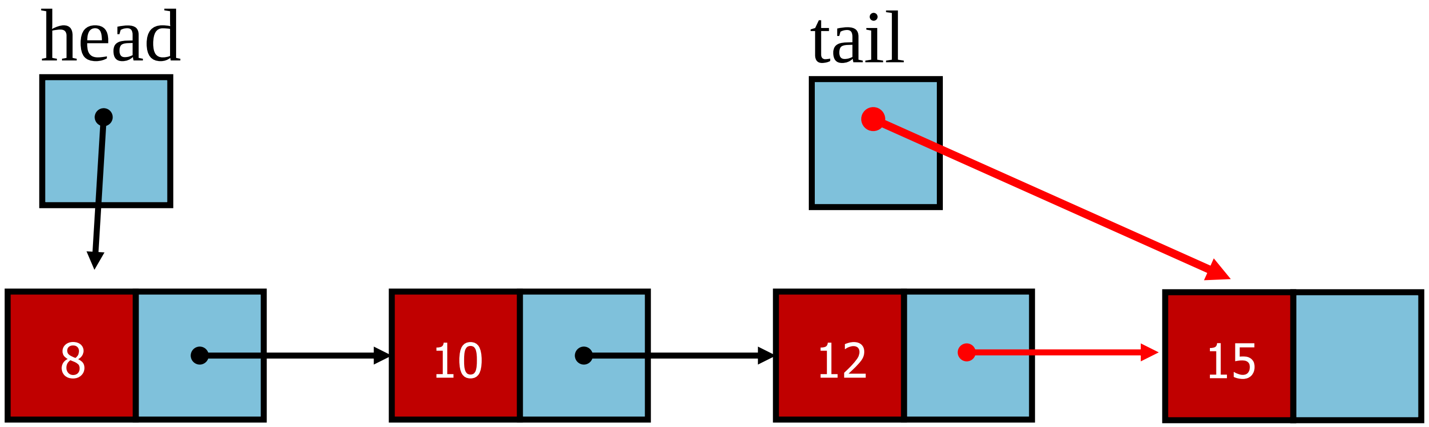
```
template<class type>
void List<type>::addToTail(type val) {
    if (tail != NULL) {
        tail->next = new Node(val);
    }
    else
        head = tail= new Node(val);
}
```



AddNode at End

```
template<class type>
void List<type>::addToTail(type val) {
    if (tail != NULL) {
        tail->next = new Node(val);
        tail = tail->next;
    }
    else
        head = tail= new Node(val);
}
```

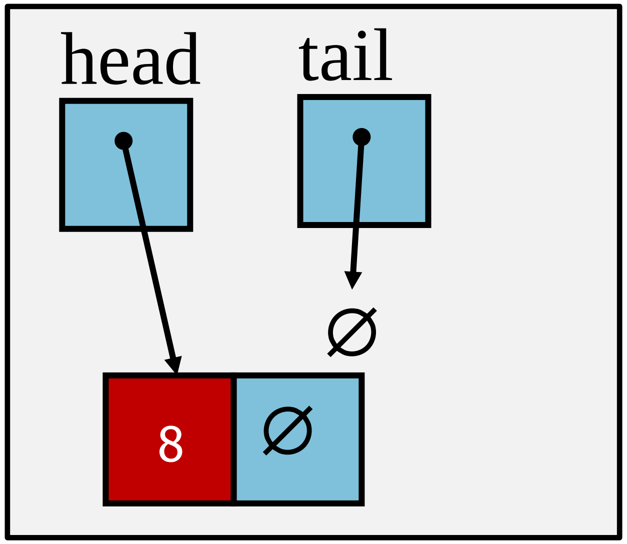
TIME COMPLEXITY ?

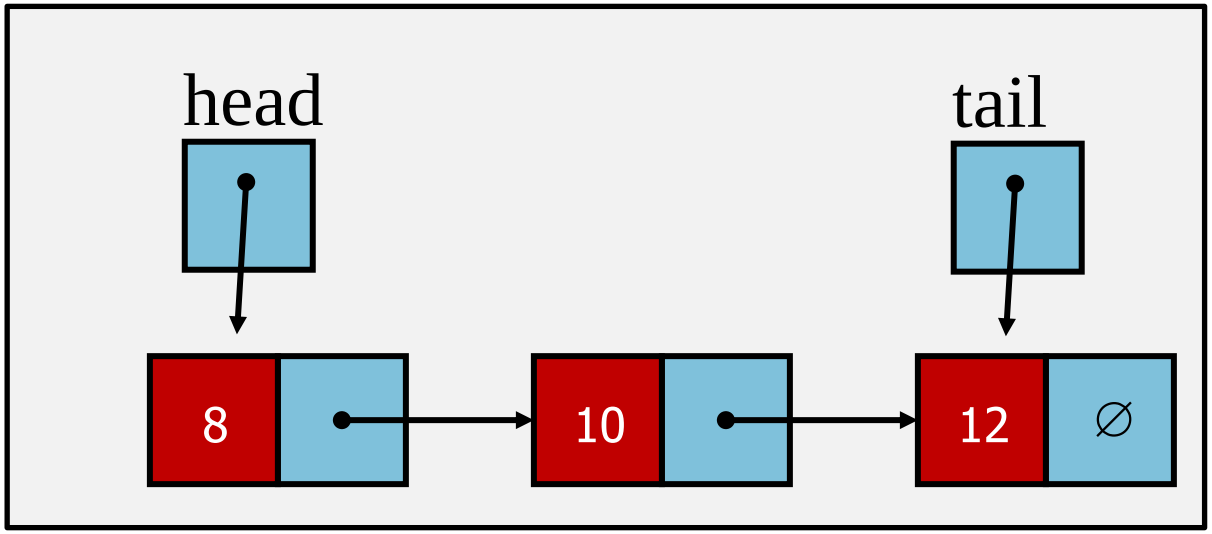


Update Method AddToHead

```
template<class type>
void List<type>::addtoHead(type val)
{
    head = new Node(val, head);
}
```

ISSUES ??



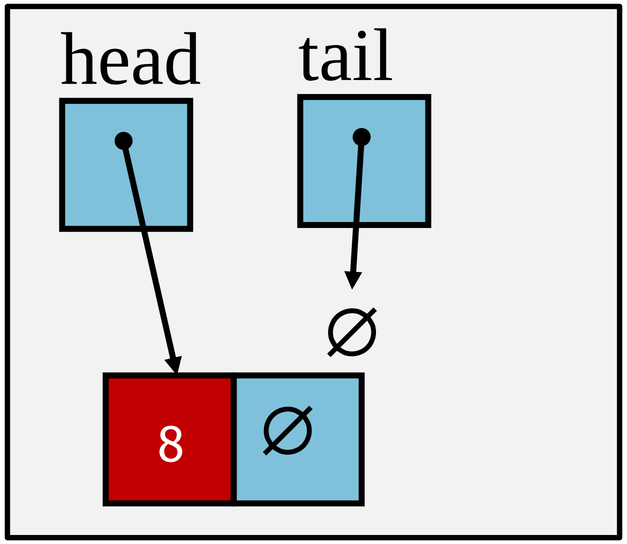


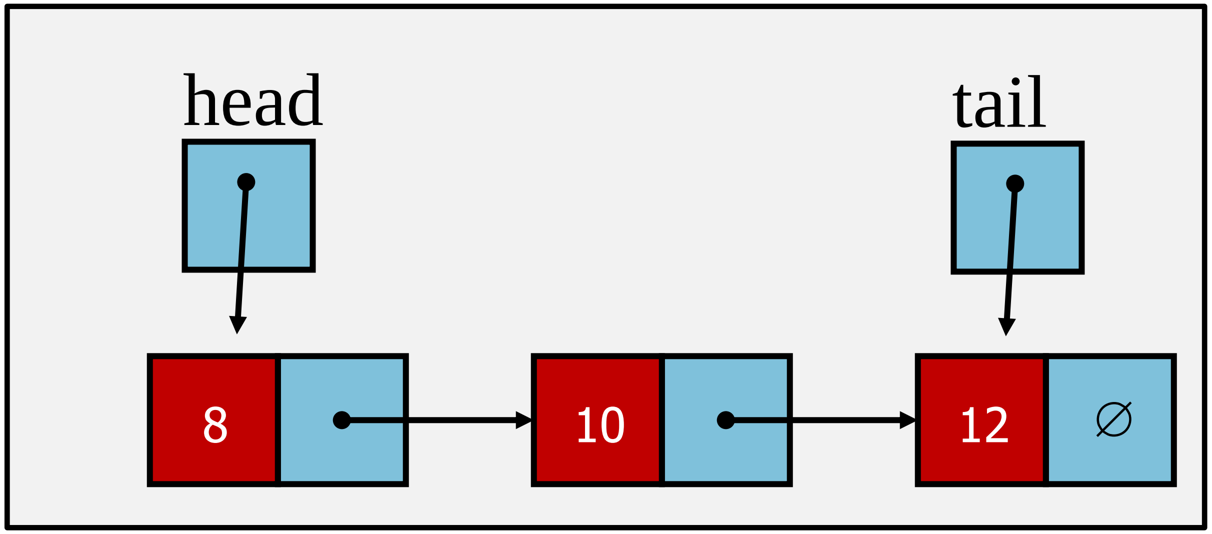
```
template<class type>
void List<type>:: addtoHead(type val){
   head = new Node(val, head);
    if (tail == 0)
         tail = head;
}
```

Update Method AddToHead

```
template<class type>
void List<type>::addtoHead(type val)
{
     head = new Node(val, head);
}
```

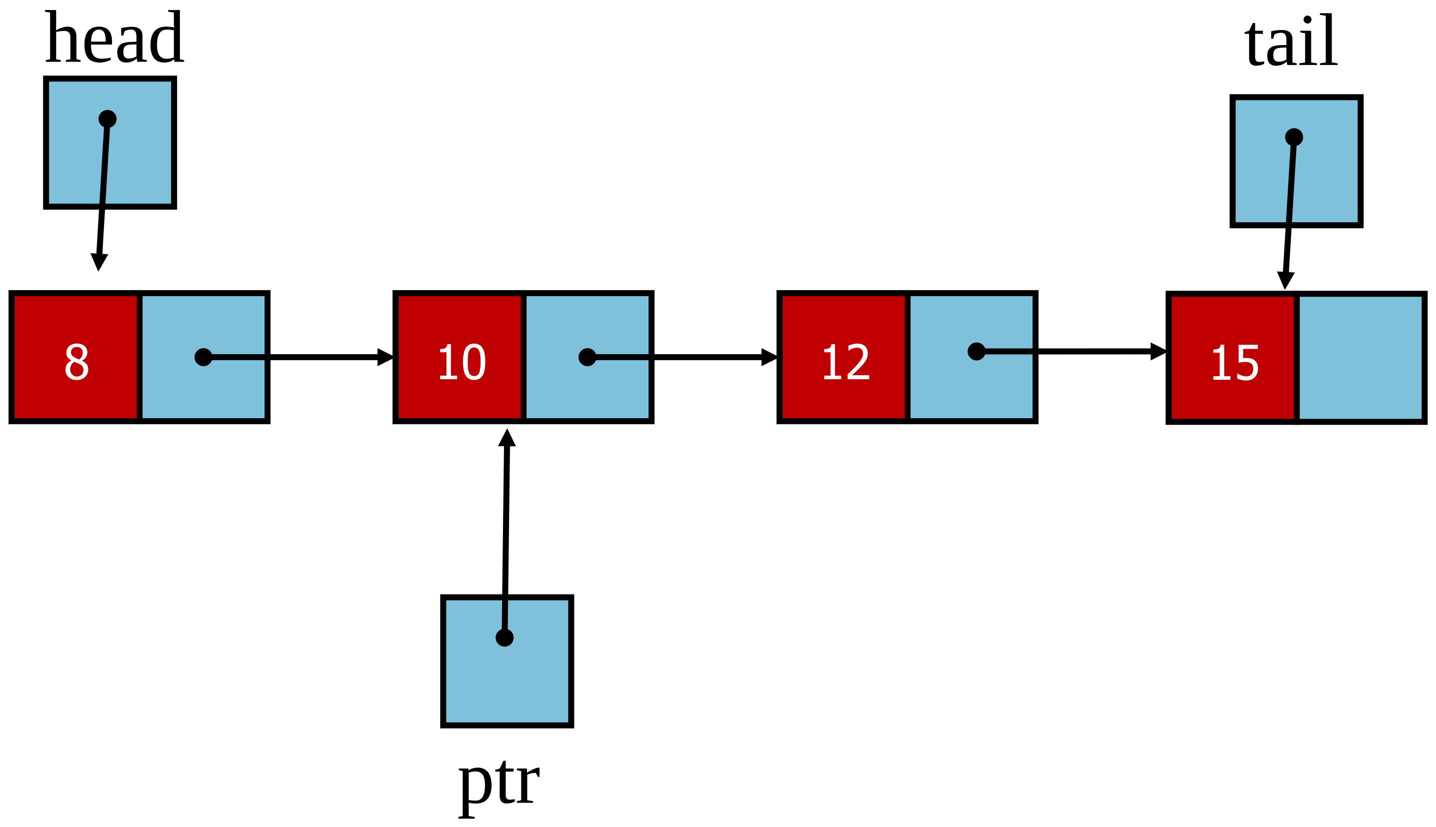
ISSUES ??





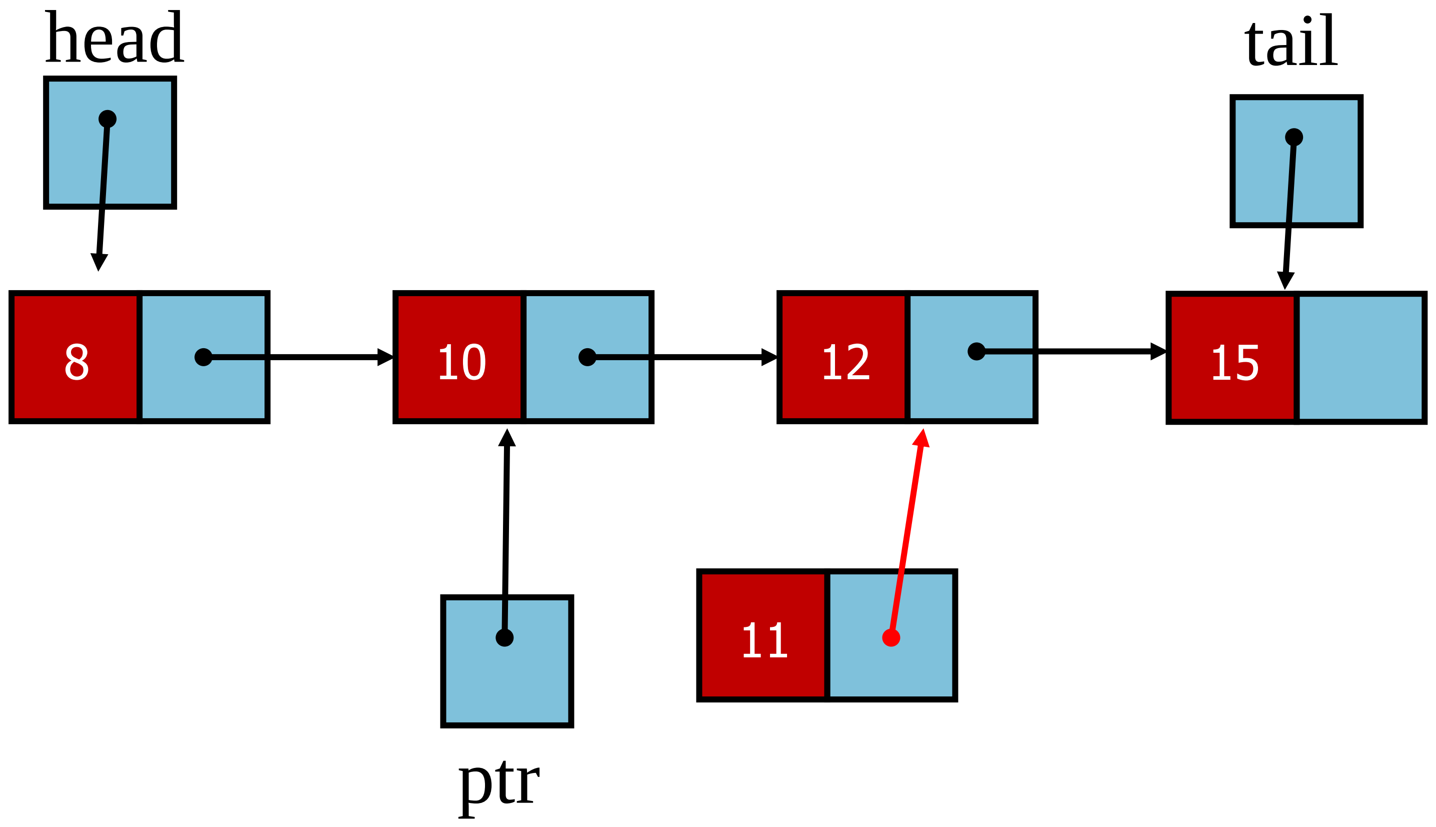
```
template<class type>
void List<type>:: addtoHead(type val){
   head = new Node(val, head);
    if (tail == 0)
         tail = head;
}
```

AddNode after a input pointer ptr



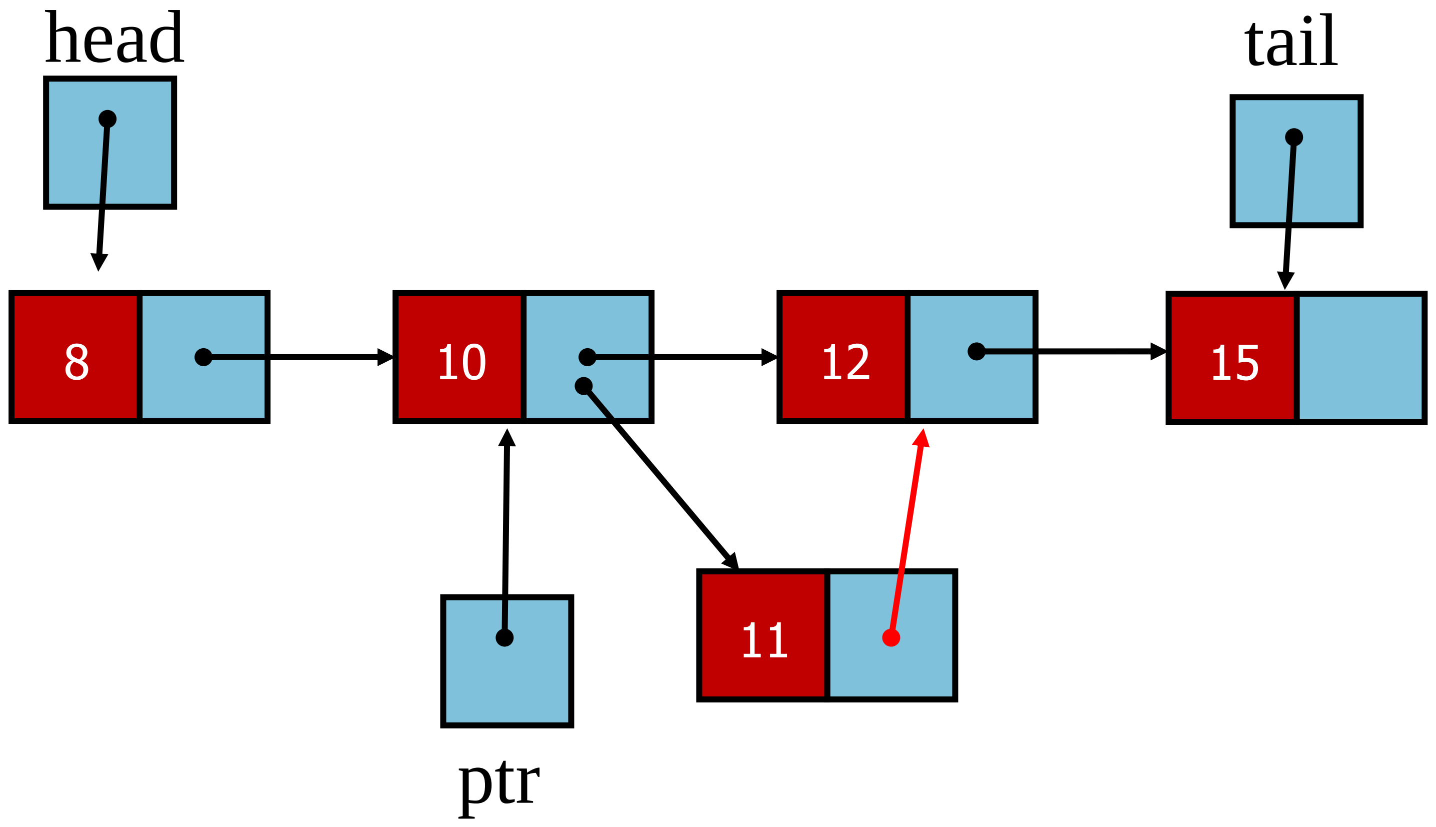
Add a new node after the given ptr

AddNode after a particular Data item



Add a new node with data=11 after the node (with data= 10)

AddNode after a particular Data item



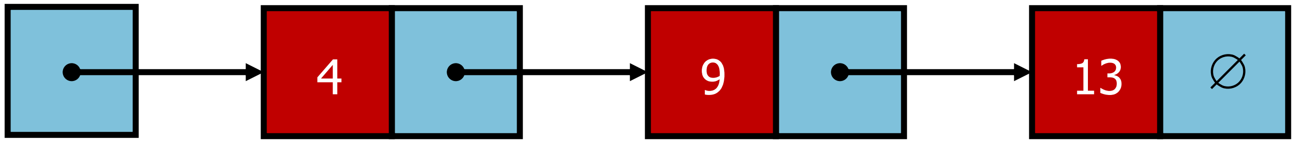
WRITE CODE

TIME COMPLEXITY ?

SL List Delete Node

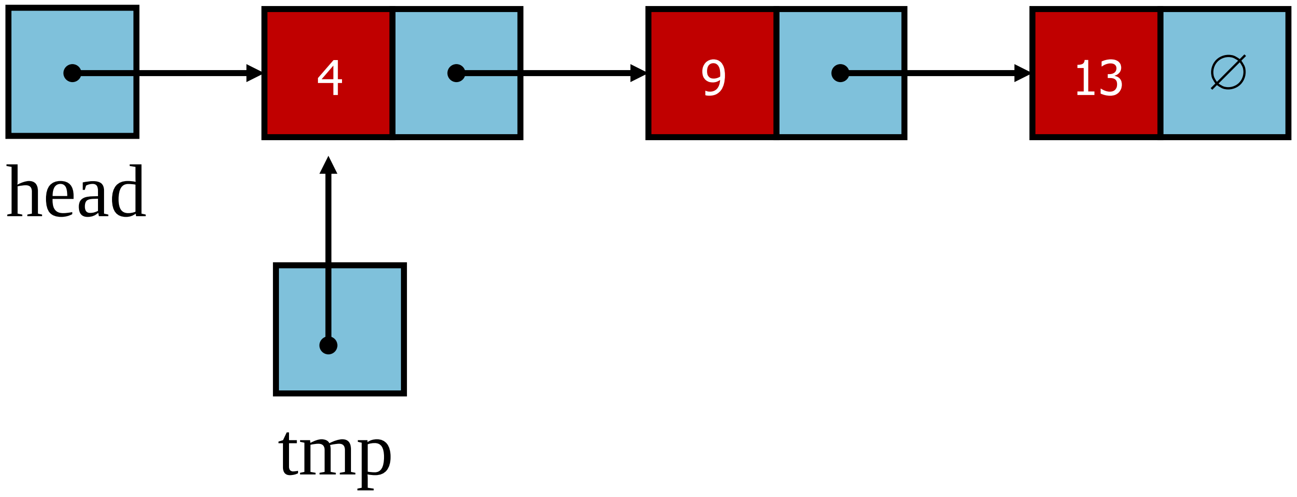
Delete Node

- **Which node to delete ?**
  - Start ?
  - End ?
  - Or the one with some particular data value ?
- **Which is most efficient ?**
- We provide all the options let user decide which to use



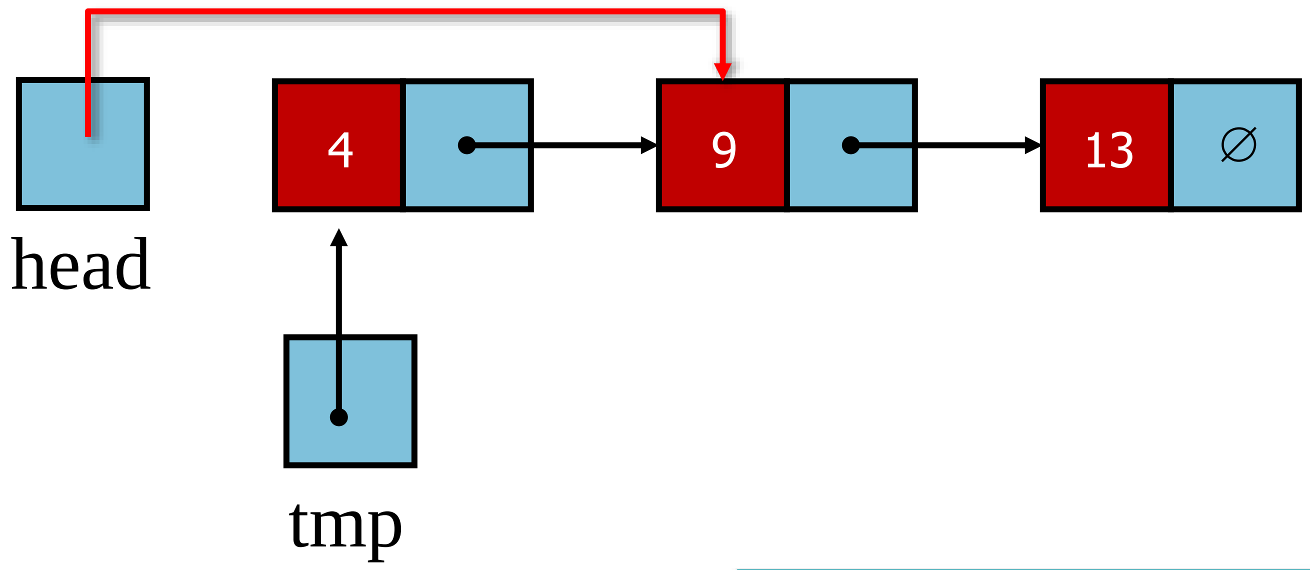
SL List Delete Node

DeleteNode From start



SL List Delete Node

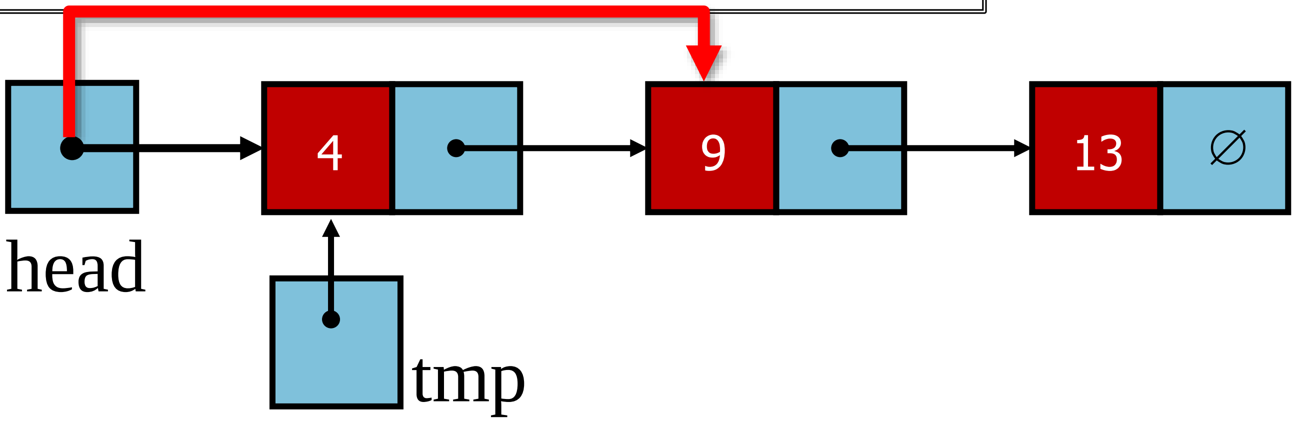
DeleteNode From start



TIME COMPLEXITY ?

DeleteNode From Start

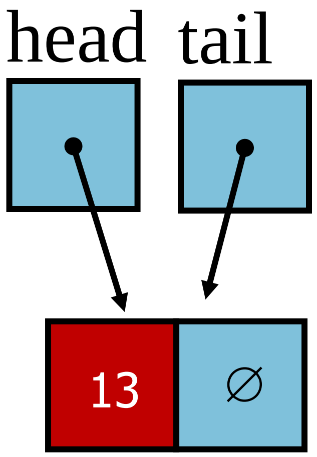
```
template<class type>
bool List<type>::deleteFromHead() {
    bool deleted = false;
    if (head != NULL) {//non empty list
        Node * tmp = head;
        if (head == tail) {// only one node in the list
                head = tail = NULL;
        }
        else  //more than one node
                head = head->next;
        delete tmp;
        deleted = true;
    }
    return deleted;
}
```



Delete Node From Tail

- **Can tail be helpful ?**

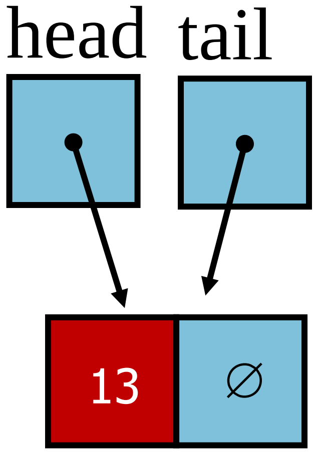
```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end





        }
    }
}
```

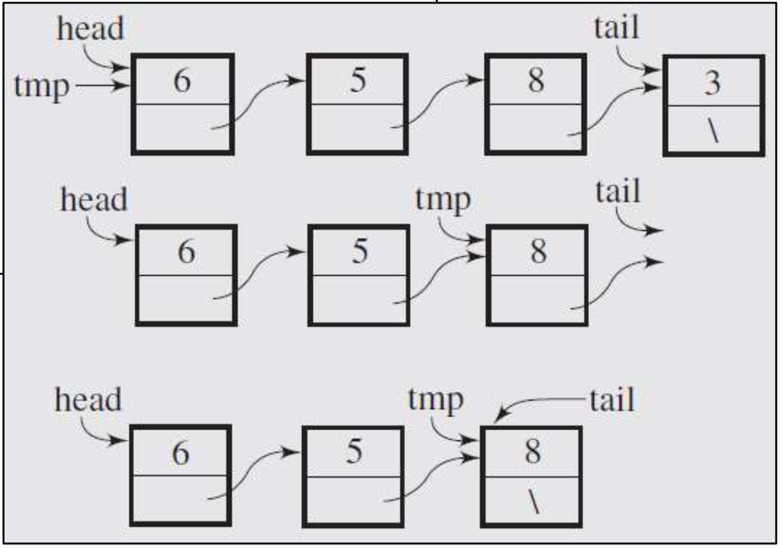


Delete Node From Tail

- Can tail be helpful ?

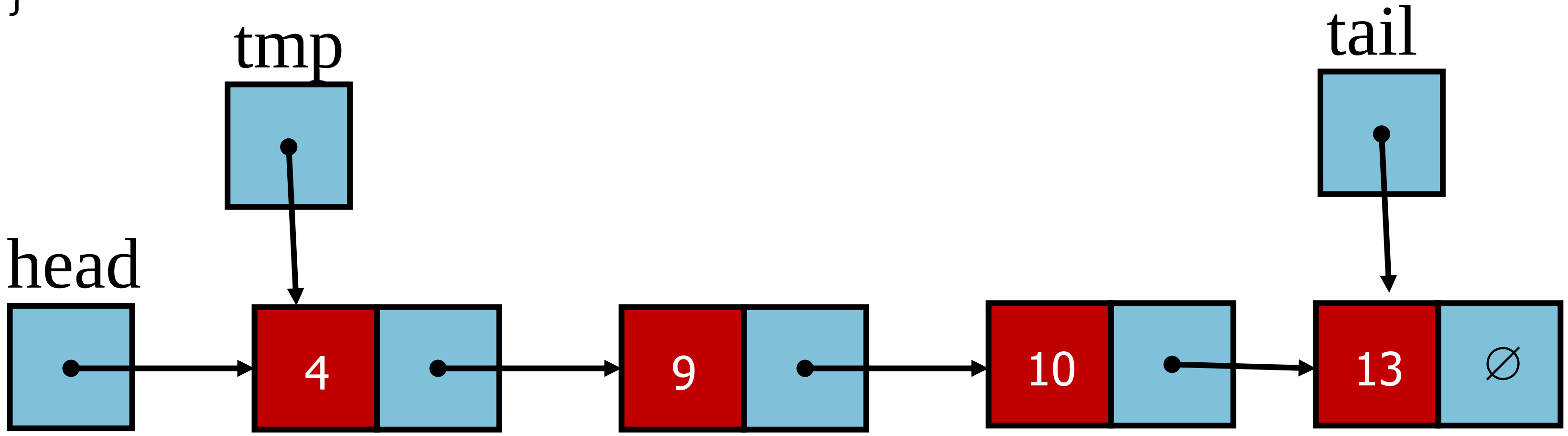
```
template<class type>
void List<type>::deleteFromTail() {
     if (head != NULL) {//non empty list
          if (head == tail) {// only one node in the list
               delete head;
               head = tail = NULL;
          }
          else { //more than one node transverse to the end
               Node * tmp = head;
               for (; tmp->next != tail; tmp = tmp->next);
               delete tail;
               tail = tmp;
               tail->next = NULL;
          }
     }
}
```





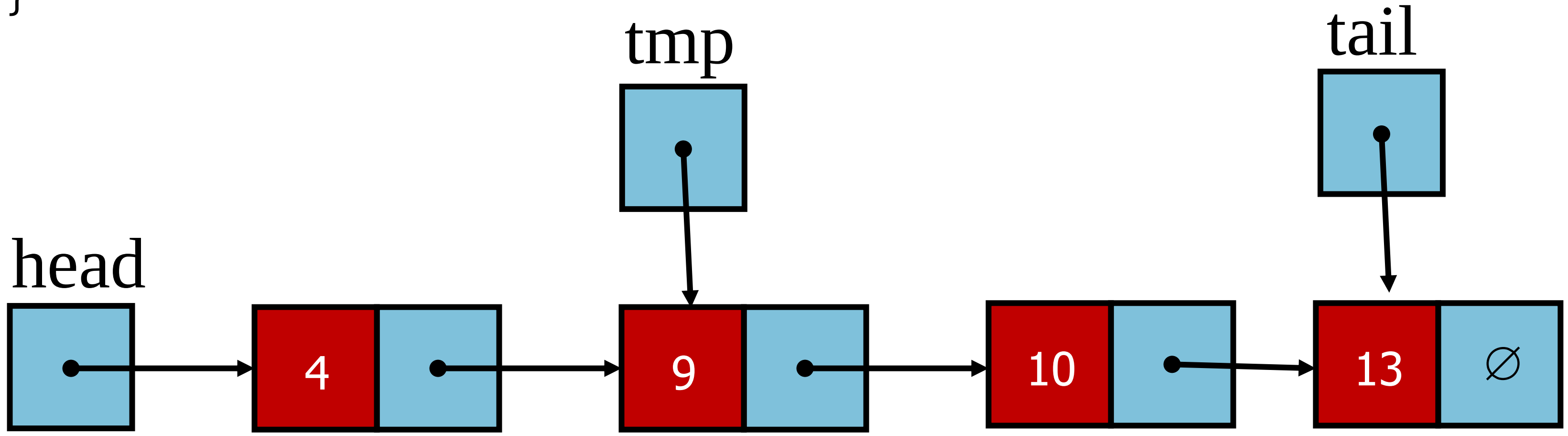
Delete Node From Tail

```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end
            Node * tmp = head;
            for (; tmp->next != tail; tmp = tmp->next);
            delete tail;
            tail = tmp;
            tail->next = NULL;
        }
    }
}
```



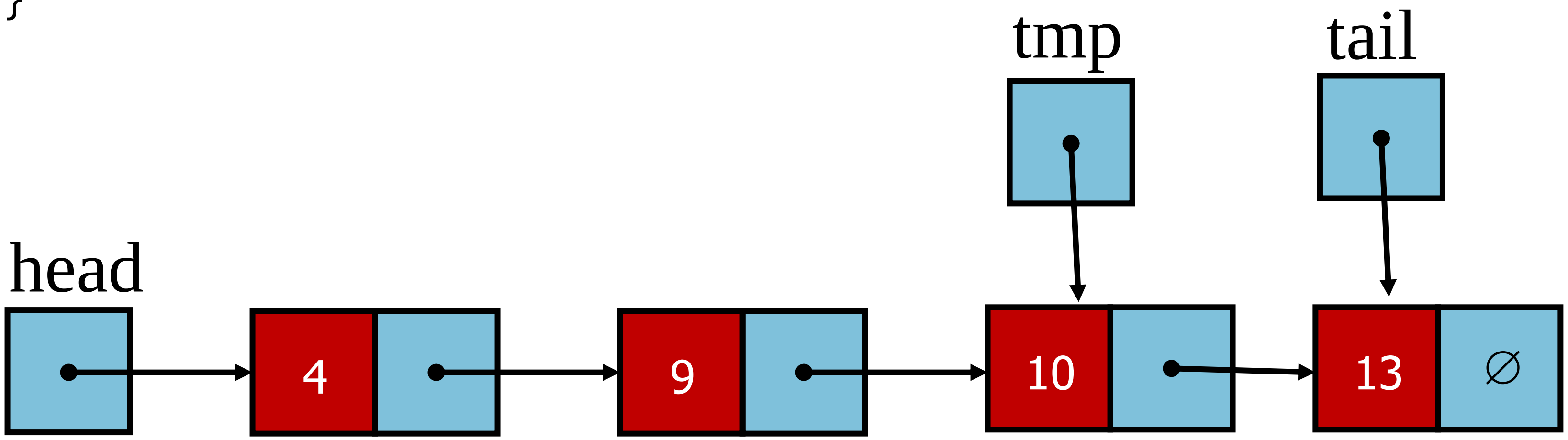
Delete Node From Tail

```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end
            Node * tmp = head;
            for (; tmp->next != tail; tmp = tmp->next);
            delete tail;
            tail = tmp;
            tail->next = NULL;
        }
    }
}
```



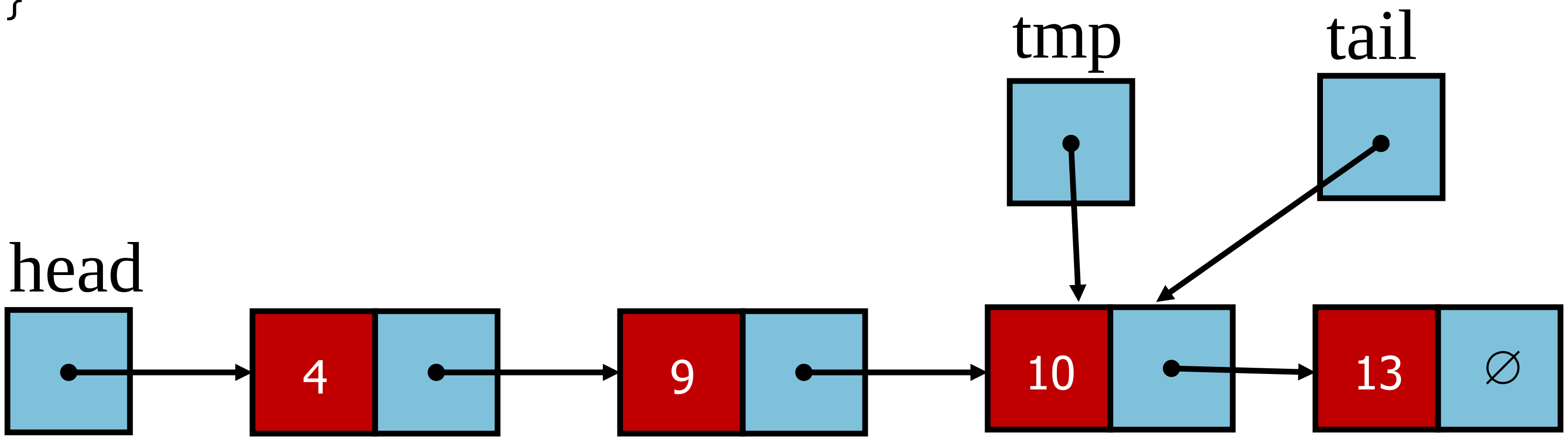
Delete Node From Tail

```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end
            Node * tmp = head;
            for (; tmp->next != tail; tmp = tmp->next);
            delete tail;
            tail = tmp;
            tail->next = NULL;
        }
    }
}
```



Delete Node From Tail

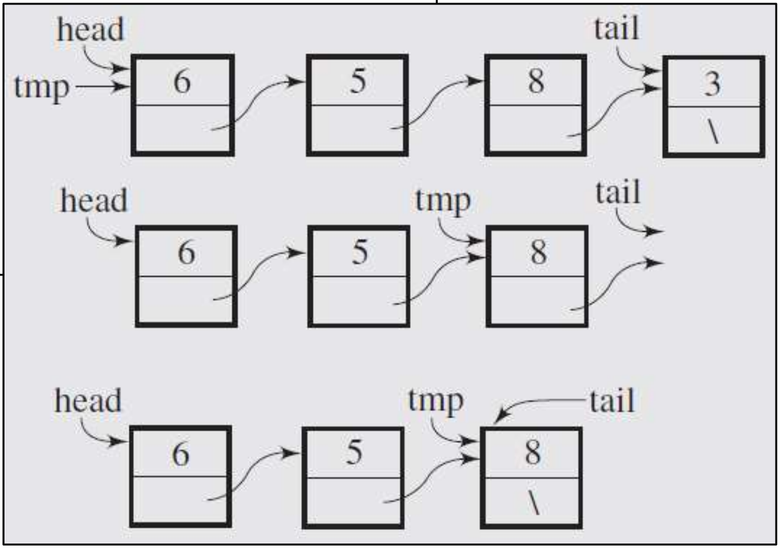
```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end
            Node * tmp = head;
            for (; tmp->next != tail; tmp = tmp->next);
            delete tail;
            tail = tmp;
            tail->next = NULL;
        }
    }
}
```



Delete Node From End

- Can tail be helpful ?

```
template<class type>
void List<type>::deleteFromTail() {
    if (head != NULL) {//non empty list
        if (head == tail) {// only one node in the list
            delete head;
            head = tail = NULL;
        }
        else { //more than one node transverse to the end
            Node * tmp = head;
            for (; tmp->next != tail; tmp = tmp->next);
            delete tail;
            tail = tmp;
            tail->next = NULL;
        }
    }
}
```

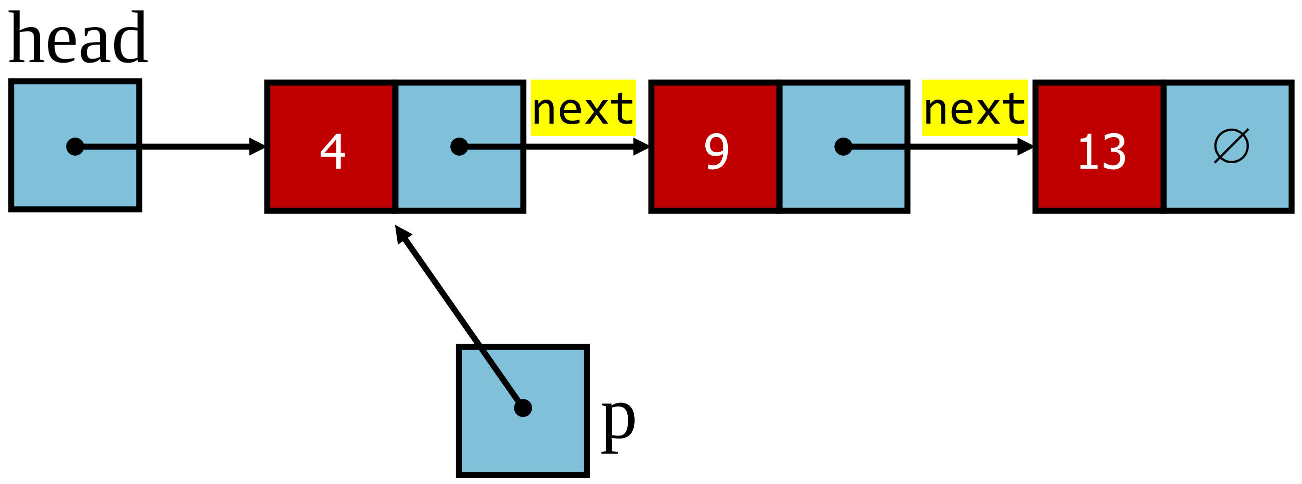


Delete Node with given input data

TRY IT YOURSELF

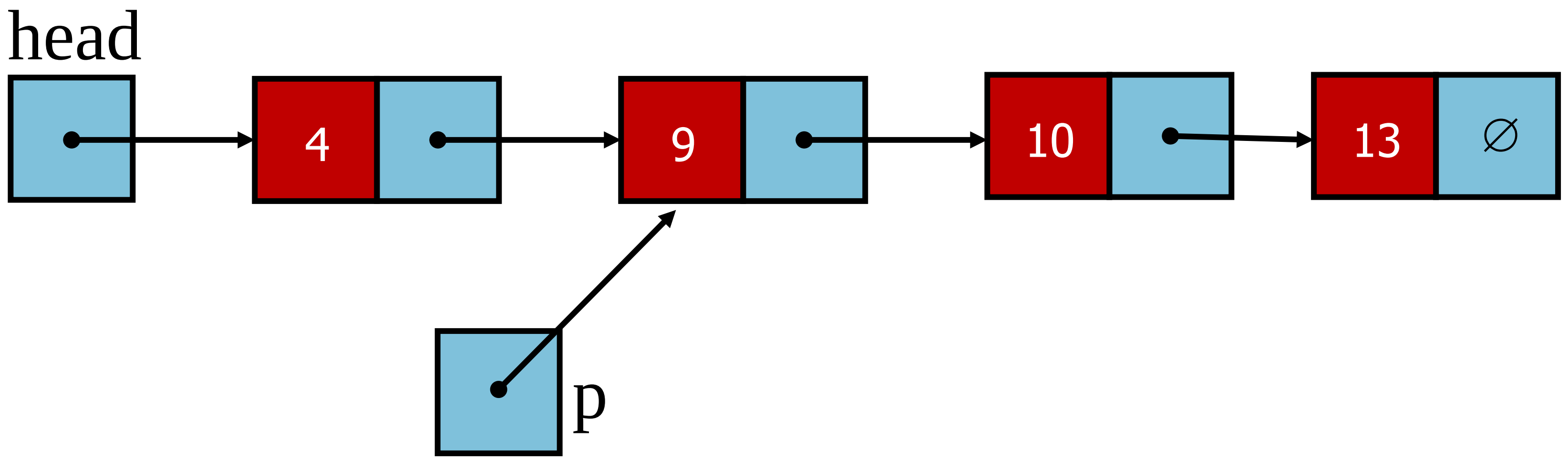
SL list Destructor

```
template<class type>
List<type>::~List() {
   Node<type> * p = head;
   while (!isEmpty()) {




   }
}
```



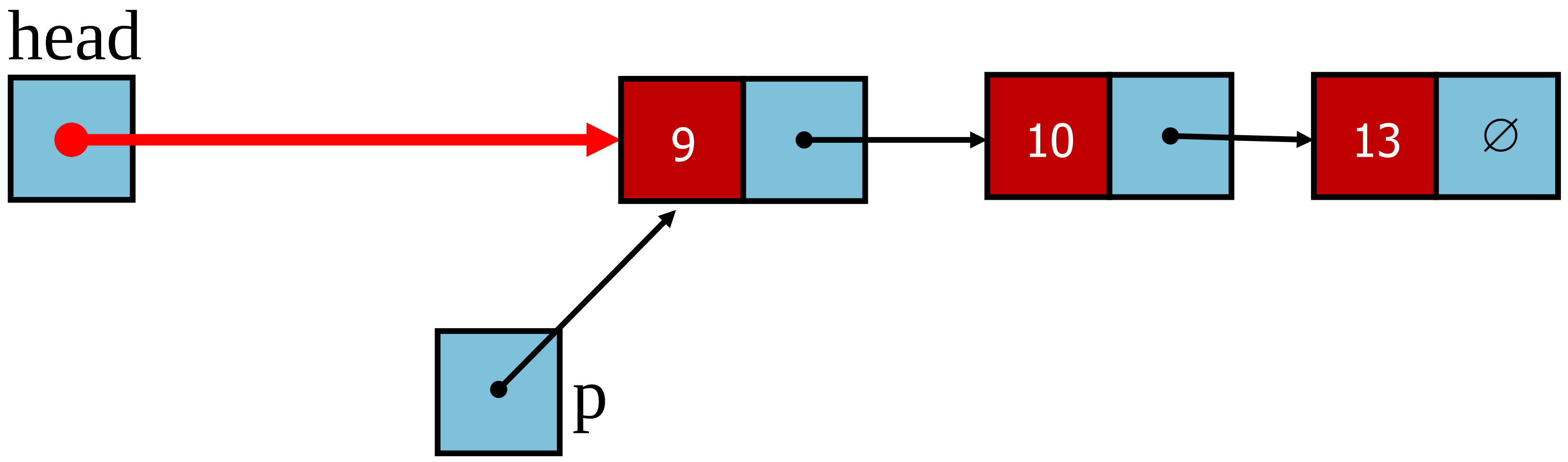
SL list Destructor

```
template<class type>
List<type>::~List() {
   Node * p = head;
   while (!isEmpty()) {
      p = head->next;
      delete head;


   }
}
```



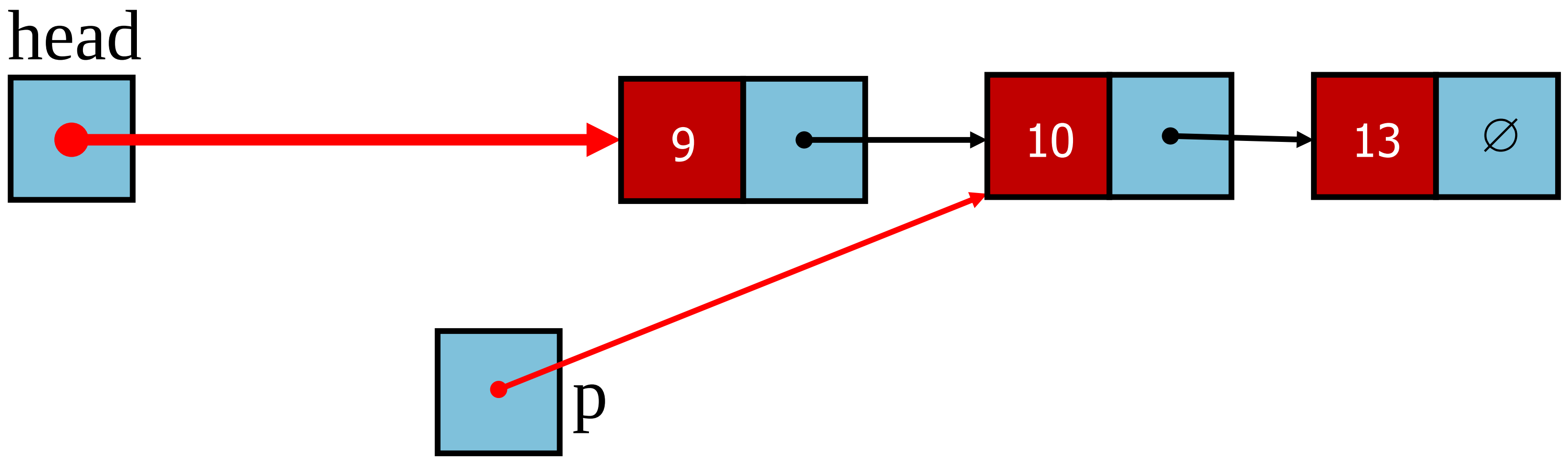
SL list Destructor

```
template<class type>
List<type>::~List() {
   Node * p = head;
   while (!isEmpty()) {
      p = head->next;
      delete head;
      head = p;
   }
}
```



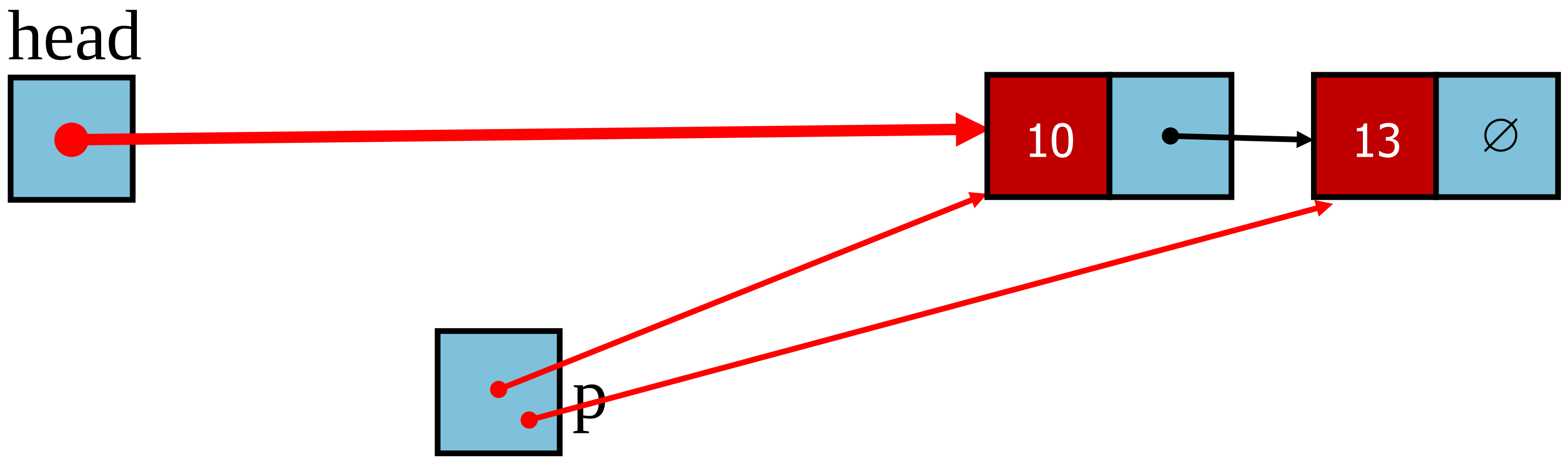
SL list Destructor

```
template<class type>
List<type>::~List() {
   Node * p = head;
   while (!isEmpty()) {
      p = head->next;
      delete head;
      head = p;
   }
}
```



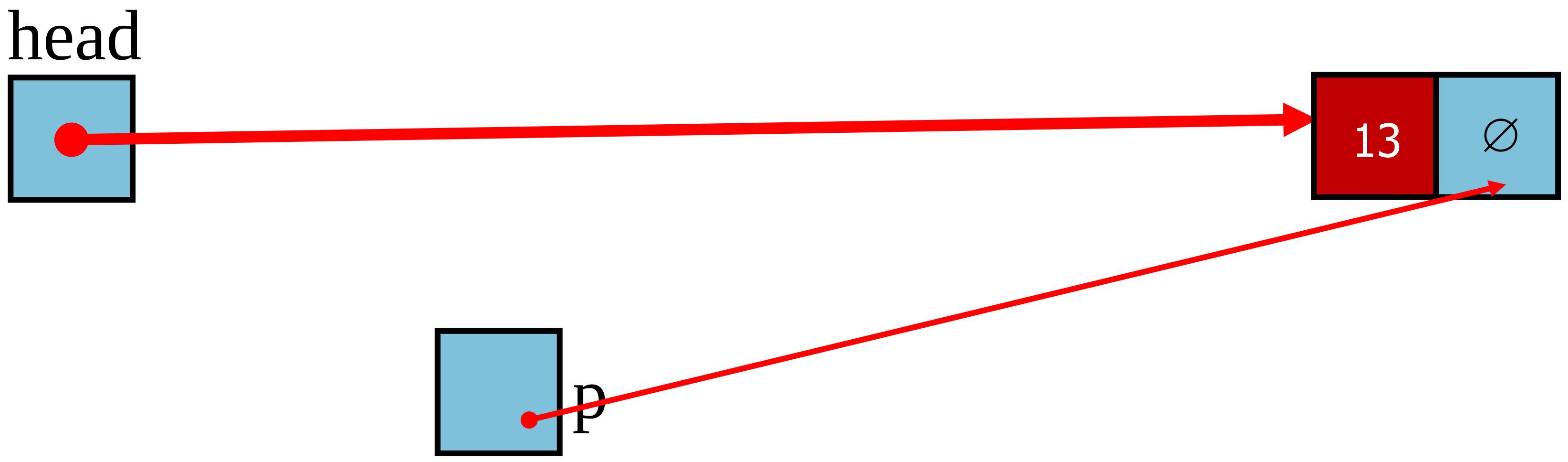
SL list Destructor

```
template<class type>
List<type>::~List() {
   Node * p = head;
   while (!isEmpty()) {
      p = head->next;
      delete head;
      head = p;
   }
}
```



SL list Destructor

```
template<class type>
List<type>::~List() {
   Node * p = head;
   while (!isEmpty()) {
      p = head->next;
      delete head;
      head = p;
   }
}
```



To Do SL list

- Implement the following functions
  - Sort the list
  - Merge two sorted lists
  - Remove the given element
  - Remove all occurrences of the given element
  - Find if two lists have same contents
  - Reverse the SL List in one pass
  - Reverse given k-nodes in a linked list
  - Reverse-Print, print the list in reverse without reversing it
  - Find median in one pass (use two pointer one move by one element and other faster …faster reach end and the first one will reach middle)

To Do SL list

- Implement the following functions
  - Remove duplicate from sorted list
  - Remove duplicate from unsorted list
  - Intersection of two sorted lists
  - Union of two sorted lists
  - Delete alternate node of linked list
  - Segregate even and odd nodes in a link list
  - Create a linked list from an array
  - Find sum of even and odd nodes in a list
  - Maximum sum of k-consecutive elements in a list
  - Check if list is palindrome

What's Wrong

```
template<class type>
List<type> List<type>::Input(){
    List<type> L;
    for (int i = 0; i < 5; i++)
        L.addToStart(i);

    return L;
}
```

```
void main() {

    List<int> L1, L2;
    L2=L1.Input();
    L2.print();
}
```

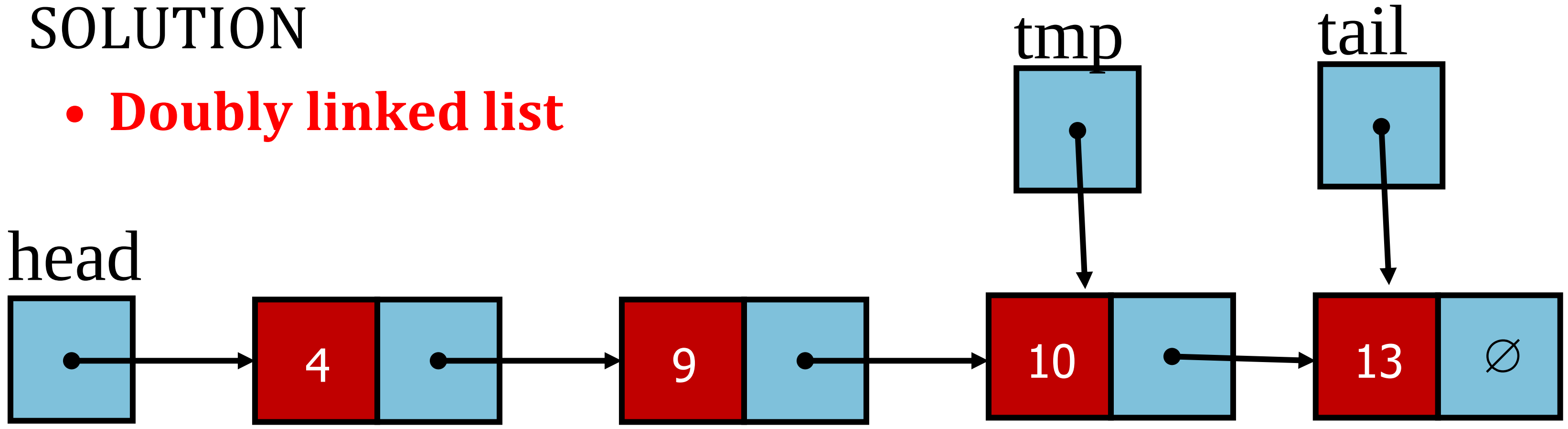
- Destructor is called at the end of the function for the object and delete the List L.
- The program will crash when u try to print it in main as L2 is NULL

Deep vs Shallow Copy

```
L2=L3
```

SL List issue

- deleteFromTail() indicates a problem inherent to singly linked lists.
  - The nodes in such lists contain only pointers to the successors; therefore, there is no immediate access to the predecessors
  - We have to scan the entire list to stop right in front of tail to delete it.
  - SOLUTION
    - **Doubly linked list**


tmp
tail
head
4
9
10
13
∅