ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 3

Linked Lists

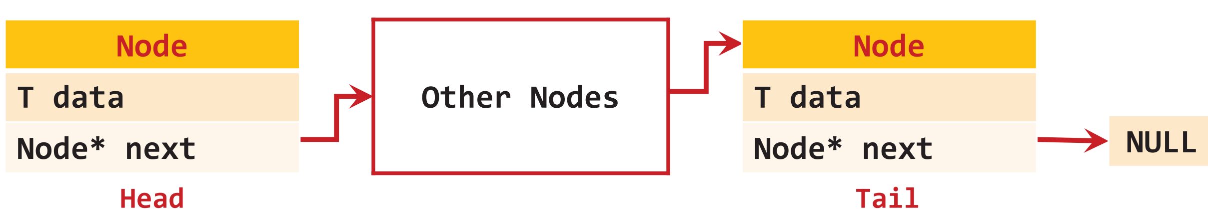
Dayu Wang

- **Node**

→ **Node is the basic unit of a linked list.**

```
/** A singly-linked list node */
template<class T>
struct Node {
    // Data fields
    T data;  // Stores some data in the node.
    Node<T>* next;  // Stores a pointer to the next node in the list.

    // Constructor
    Node(const T& data) : data(data), next(NULL) {}
};
```

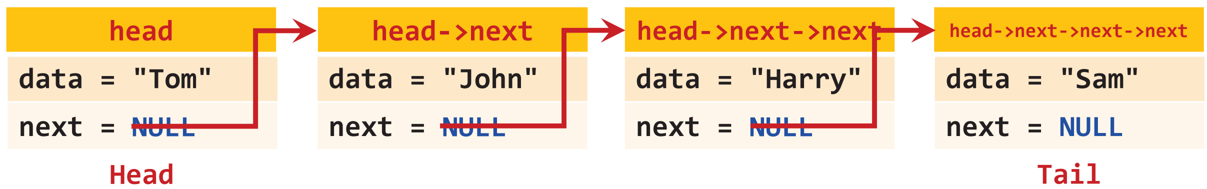


- **Items stored in a vector are physically connected in memory.**
- **Items stored in a linked list are logically connected in memory.**

→ **This kind of class definition is sometimes called self-referential, because it contains a field (pointer) of the same type as itself.**

- **Building a linked list**

```
Node<string>* tom = new Node<string>("Tom");
Node<string>* john = new Node<string>("John");
Node<string>* harry = new Node<string>("Harry");
Node<string>* sam = new Node<string>("Sam");
tom->next = john;
john->next = harry;
harry->next = sam;
```

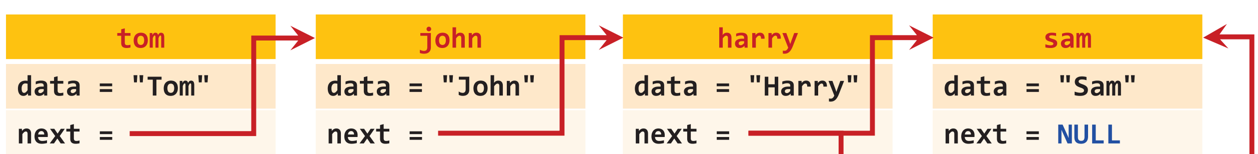


→ **In most cases, we only keep a pointer to the head of a linked list.**

```
Node<string>* head = new Node<string>("Tom");
head->next = new Node<string>("John");
head->next->next = new Node<string>("Harry");
head->next->next->next = new Node<string>("Sam");
```

Without specifically mention, you can only assume that you only have a pointer to the head of the a linked list.

- **Inserting a new node into linked list**



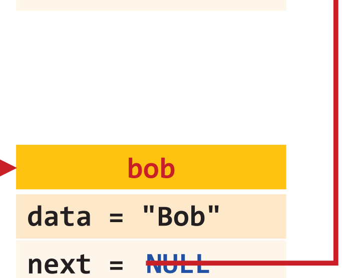
→ **Suppose we only have a pointer to node** harry**.**

→ **How to insert a new node, "Bob", after "Harry"?**

1) Create a new node, bob**.**

2) Set bob->next **equal to** harry->next**.**

3) Set harry->next **equal to** bob**.**




```
Node<string>* bob = new Node<string>("Bob");
bob->next = harry->next;
harry->next = bob;
```

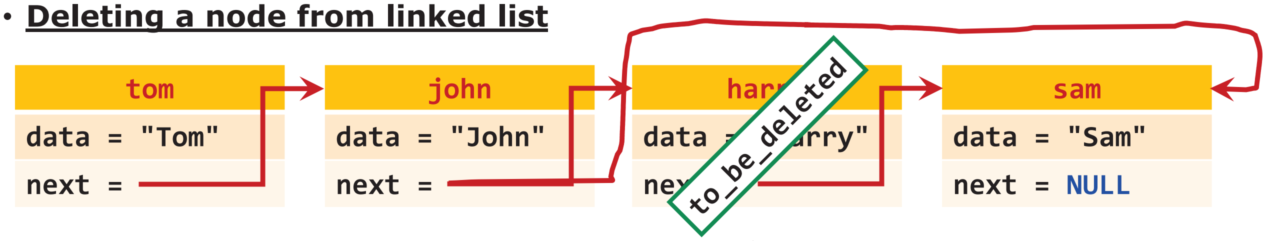
Time complexity is $O(1)$, no data shift needed.

→ **Can we swap steps 2) and 3), do 3) first then 2)?**

No. After we set harry->next **to** bob**, we lose the reference to** sam**.**

→ **[Good Habit] When you are working with linked lists, always have scratch paper, pencils (with erasers) with you.**

- **Deleting a node from linked list**



→ **Suppose we only have a pointer to node** `john`.

→ **How to delete the node after** `john`**?**

```
Node<string>* to_be_deleted = john->next;
john->next = john->next->next;
delete to_be_deleted;
```

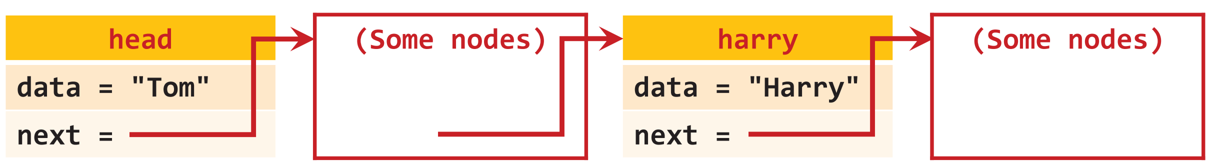
Time complexity is $O(1)$, no data shift needed.

- **Traversing a linked list**

```
Node<string>* current = head;
while (current) {
    cout << current->data << endl;
    current = current->next;  // Advances the current node.
}
```

For singly-linked lists, traversing backward is impossible.

- **How to insert a new node before a specific node?**



→ **You only have the pointer to the** `head` **("Tom") of the linked list.**

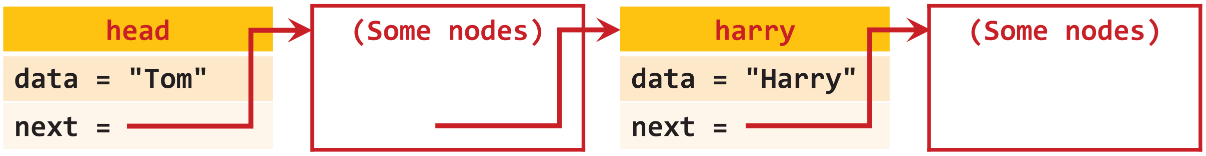
→ **How to add a new node, "Lisa", before "Harry"?**

- **Suppose there are at least 2 nodes in the linked list.**
- **Suppose** `harry` **is not the first node of the linked list.**

```
Node<string>* current = head;
while (current->next && current->next->data != "Harry") {
    current = current->next;
}
Node<string>* lisa = new Node<string>("Lisa");
lisa->next = current->next;
current->next = lisa;
```

- **You only know the next node of** `current`**; not the previous node.**
- **Searching from the head of the linked list gives time complexity of $O(n)$.**

- **How to delete a specific node?**



→ **You only have the pointer to the** head **("Tom") of the linked list.**

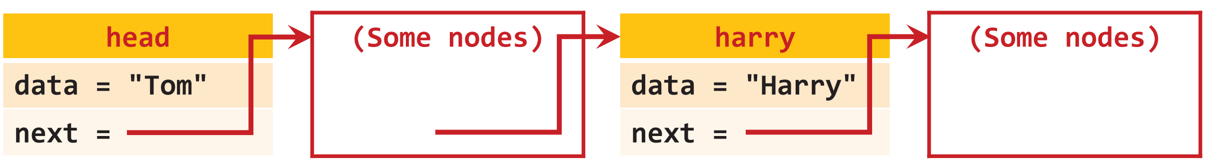
→ **How to delete node "Harry"?**

- **Suppose there are at least 2 nodes in the linked list.**
- **Suppose** harry **is not the first node of the linked list.**

```
Node<string>* current = head;
while (current->next && current->next->data != "Harry") {
    current = current->next;
}
Node<string>* to_be_deleted = current->next;
current->next = current->next->next;
delete to_be_deleted;
```

- **Time complexity:** $O(n)$

- **[Exercise]**



→ **You only have the pointer to the** head **("Tom") of the linked list.**

→ **Please write some code to delete the node before "Harry".**

- **Suppose there are at least 3 nodes in the linked list.**
- **Suppose** harry **is not the first or second node of the linked list.**

```
Node<string>* current = head;
while (current->next->next && current->next->next->data != "Harry") {
    current = current->next;
}
Node<string>* to_be_deleted = current->next;
current->next = current->next->next;
delete to_be_deleted;
```

- **Time complexity:** $O(n)$

- **Performance of Singly-Linked List**

→ **Inserting a node after a referenced node:** $O(1)$

→ **Inserting a node before a referenced node:** $O(n)$

→ **Deleting the node after a referenced node:** $O(1)$

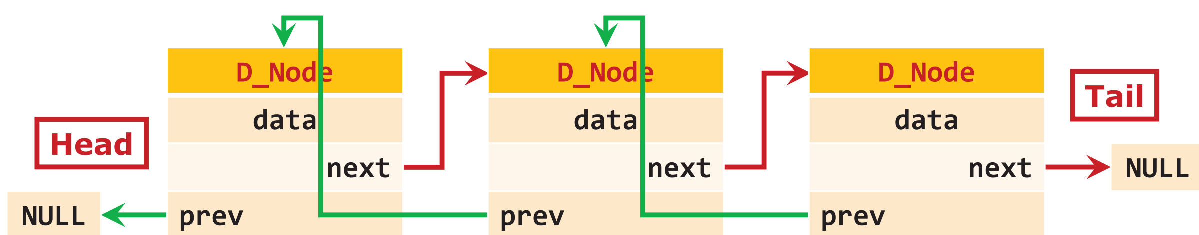
→ **Deleting the referenced node:** $O(n)$

→ **Deleting the node before a referenced node:** $O(n)$

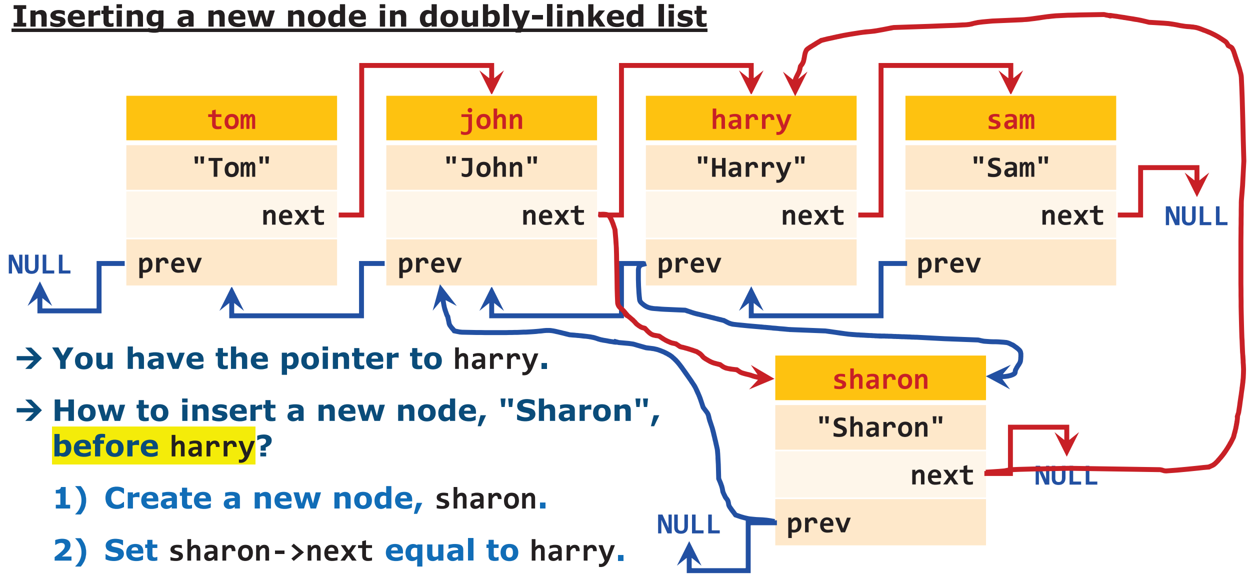
- **Doubly-Linked List**

```
/** A doubly-linked list node */
template<class T>
struct D_Node {
    // Data fields
    T data;  // Stores some data in the node.
    D_Node<T>* next;  // A reference to the next node in the list.
    D_Node<T>* prev;  // A reference to the previous node in the list.

    // Constructor
    D_Node(const T& data) : data(data), next(NULL), prev(NULL) {}
};
```



- **Inserting a new node in doubly-linked list**



→ **You have the pointer to** harry.

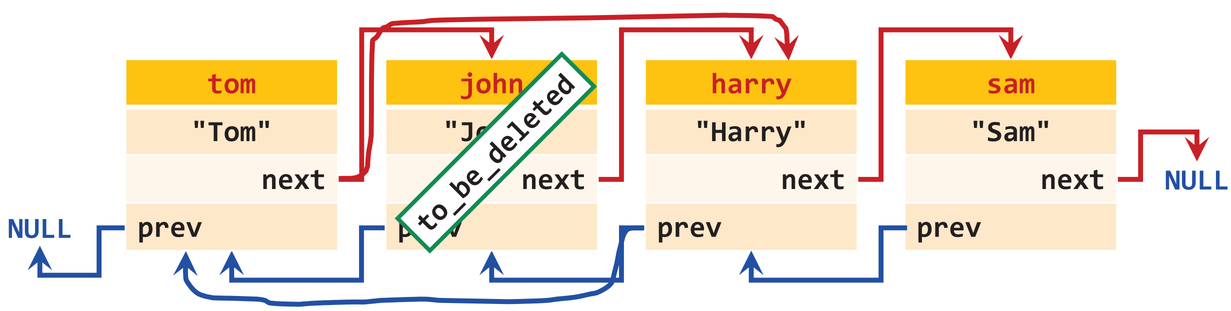
→ **How to insert a new node, "Sharon", before** harry**?**

1) **Create a new node,** sharon.
2) **Set** sharon->next **equal to** harry.
3) **Set** sharon->prev **equal to** harry->prev.
4) **Set** sharon->next->prev **equal to** sharon.
5) **Set** sharon->prev->next **equal to** sharon.

```
D_Node<string>* sharon = new D_Node<string>("Sharon");
sharon->next = harry;
sharon->prev = harry->prev;
sharon->next->prev = sharon;
sharon->prev->next = sharon;
```

Time complexity: $O(1)$

- **Deleting a node from doubly-linked list**



→ **You have the pointer to** harry.

→ **How to delete the node before** harry**?**

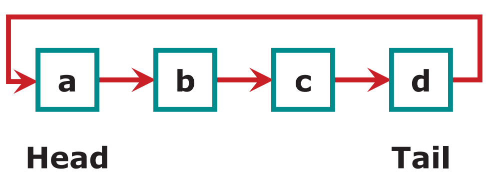
1) **Store** harry->prev **to** to_be_deleted.
2) **Set** harry->prev **equal to** harry->prev->prev.
3) **Set** harry->prev->next **equal to** harry.
4) **Delete** to_be_deleted.

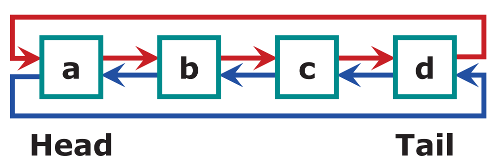
```
D_Node<string>* to_be_deleted = harry->prev;
harry->prev = harry->prev->prev;
harry->prev->next = harry;
delete to_be_deleted;
```

Time complexity: $O(1)$

- **Please practice the procedures below at home.**
  - → Inserting a new node, "Pete", after `harry`.
  - → Inserting a new node, "Sharon", before `harry` (included in slides).
  - → Deleting the node after `harry`.
  - → Deleting the node `harry`.
  - → Deleting the node before `harry` (included in slides).
- **Performance of Doubly-Linked List**
  - → Inserting a node after a referenced node: $O(1)$
  - → Inserting a node before a referenced node: $O(1)$
  - → Deleting the node after a referenced node: $O(1)$
  - → Deleting the referenced node: $O(1)$
  - → Deleting the node before a referenced node: $O(1)$

- **Circular List**
  - → A linked list is circular if the head and tail are linked together.





  - → For singly-linked lists, tail's `next` is head.
  - → For doubly-linked lists, tail's `next` is head and head's `prev` is tail.
  - → Since the list is circular, there will be no concept of head or tail.
    Any node can be treated as head or tail.

Pros	Cons
You can continue to traverse the list after the last item.	There is a risk of infinite loop.

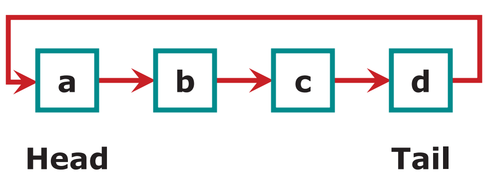
- **[Exercise]**

→ **Given the pointer to a node in a singly-linked list, write a function to test whether the linked list is circular.**

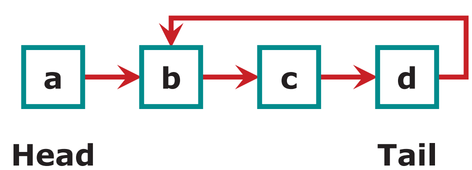
```cpp
/** Tests whether a singly-linked list is circular.
    @param node: pointer to a node in the linked list
    @return: {true} if the list is circular; {false} if it is linear
*/
template<class T>
bool is_circular(Node<T>* node) {
    Node<T>* current = node;
    while (current) {
        current = current->next;
        if (current == node) { return true; }
    }
    return false;
}
```

Time complexity: $O(n)$

→ **"Is Circular" and "Has Cycle" are different concepts!**



The list is circular.

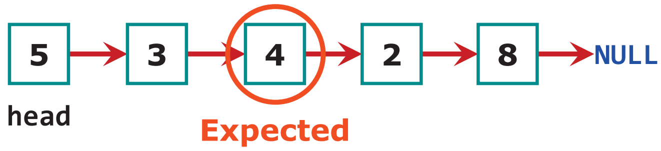


The list has cycle.

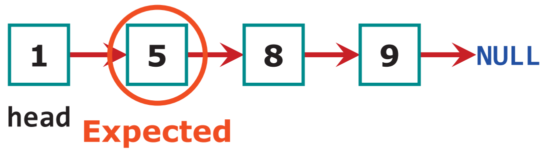
- **[Exercise]**

→ **Given the reference of the head of a non-empty singly-linked list, write a function that returns the item stored in the "middle node" of the list.**

Example 1:



Example 2:




```
/** Returns the element stored in the middle node of a linked list.
    @param head: a pointer to the head of the linked list
    @return: element stored in the middle node
*/
template<class T>
T middle_value(Node<T>* head) {
    unsigned int count = 0;
    Node<T>* p = head;
    while (p != NULL) { count++; p = p->next; }
    p = head;
    for (unsigned int i = 0; i < (count - 1) / 2; i++) { p = p->next; }
    return p->data;
}  // Time complexity: O(n).  Two loops (2 passes) used.
```


- **[Exercise] - This was a Google interview question.**

→ **Given the reference of the head of a non-empty singly-linked list, write a function that returns the item stored in the "middle node" of the list using only one pass.**

```
/** Returns the element stored in the middle node of a linked list.
    @param head: a pointer to the head of the linked list
    @return: element stored in the middle node
*/
template<class T>
T middle_value_one_pass(Node<T>* head) {
    bool is_even = true;
    Node<T>* fast = head;  // {fast} advances in every iteration.
    Node<T>* slow = head;  // {slow} advances in every even iteration.
    while (fast->next) {
        fast = fast->next;
        is_even = !is_even;
        if (is_even) { slow = slow->next; }
    }
    return slow->data;
}  // Time complexity: O(n)
```

- **The `List` Class**

→ **The `List` class implements a doubly-linked list.**

→ **The `D_Node` struct inside the `List` class**

```
/** A doubly-linked list node */
struct D_Node {
    // Data fields
    T data;  // Stores some data in the node.
    D_Node* next;  // A pointer to the next node in the list
    D_Node* prev;  // A pointer to the previous node in the list

    // Constructor
    D_Node(const &T);
};
```

`D_Node` is private since user will not use it outside the `List` class.

```
template<class T>
List<T>::D_Node::D_Node(const T& data) : data(data), next(NULL), prev(NULL) {}
```

→ **Data fields**

```
// Data fields
D_Node *head, *tail;  // Stores pointers to the first and last node.
size_t num_of_items;  // Stores the number of elements in the list.
```

We only need to keep track of the `head` and `tail`, not every node.

→ **Default constructor**

```
// Default constructor
template<class T>
List<T>::List() : head(NULL), tail(NULL), num_of_items(0) {}
```

→ **Copy constructor**

```
// Copy constructor
template<class T>
List<T>::List(const List<T>& other) {
    head = tail = NULL;
    *this = other;
}
```

→ **Destructor**

```
// Destructor
template<class T>
List<T>::~List() { clear(); }
```

→ Deep-copy assignment operator

```cpp
// Deep-copy assignment operator
template<class T>
const List<T>& List<T>::operator = (const List<T>& rhs) {
    if (this != &rhs) {
        clear();  // Will implement later.
        num_of_items = rhs.num_of_items;
        if (num_of_items) {
            head = tail = new D_Node<T>(rhs.head->data);
            D_Node<T>* q = rhs.head->next;
            while (q) {
                tail->next = new D_Node<T>(q->data);
                tail->next->prev = tail;
                tail = tail->next;
                q = q->next;
            }
        }
    }
    return *this;
}
```


- **Functions**

Function	Behavior
size_t size() const;	Returns the number of elements in the list.
bool empty() const;	Tests whether the list is empty.
T& front(); const T& front() const;	Returns the first element in the list.
T& back(); const T& back() const;	Returns the last element in the list.
void push_front(const T&);	Adds an element to the front end of the list.
void push_back(const T&);	Adds an element to the rear end of the list.
void pop_front();	Deletes the element at the front end of the list.
void pop_back();	Deletes the element at the rear end of the list.
void clear();	Deletes all the elements in the list.

→ The size() function

```
template<class T>
size_t Linked_List<T>::size() const {
    return num_of_items;
}  // Time complexity: O(1)
```

→ The empty() function

```
template<class T>
bool Linked_List<T>::empty() const {
    return !size();
}  // Time complexity: O(1)
```

→ The front() functions

```
template<class T>
T& Linked_List<T>::front() {
    if (empty()) { throw exception("Accessing empty listed list"); }
    return head->data;
}  // Time complexity: O(1)
```

```
template<class T>
const T& Linked_List<T>::front() const {
    if (empty()) { throw exception("Accessing empty listed list"); }
    return head->data;
}  // Time complexity: O(1)
```


→ The back() functions

```
template<class T>
T& Linked_List<T>::back() {
    if (empty()) { throw exception("Accessing empty listed list"); }
    return tail->data;
}  // Time complexity: O(1)
```

```
template<class T>
const T& Linked_List<T>::back() const {
    if (empty()) { throw exception("Accessing empty listed list"); }
    return tail->data;
}  // Time complexity: O(1)
```

→ The push_front() function

```
template<class T>
void Linked_List<T>::push_front(const T& item) {
    if (num_of_items++ == 0) { head = tail = new D_Node(item); }
    else {
        head->prev = new D_Node(item);
        head->prev->next = head;
        head = head->prev;
    }
}  // Time complexity: O(1)
```

→ The push_back() function

```
template<class T>
void Linked_List<T>::push_back(const T& item) {
    if (!(num_of_items++)) { head = tail = new D_Node(item); }
    else {
        tail->next = new D_Node(item);
        tail->next->prev = tail;
        tail = tail->next;
    }
}  // Time complexity: O(1)
```

→ The pop_front() function

```
template<class T>
void Linked_List<T>::pop_front() {
    if (empty()) { throw exception("Accessing empty listed list"); }
    D_Node* to_be_deleted = head;
    if (num_of_items-- == 1) { head = tail = NULL; }
    else {
        head = head->next;
        head->prev = NULL;
    }
    delete to_be_deleted;
}  // Time complexity: O(1)
```


→ The pop_back() function

```
template<class T>
void Linked_List<T>::pop_back() {
    if (empty()) { throw exception("Accessing empty listed list"); }
    D_Node* to_be_deleted = tail;
    if (num_of_items-- == 1) { head = tail = NULL; }
    else {
        tail = tail->prev;
        tail->next = NULL;
    }
    delete to_be_deleted;
}  // Time complexity: O(1)
```

→ The clear() function

```
template<class T>
void Linked_List<T>::clear() {
    while (head) {
        D_Node* p = head;
        head = head->next;
        delete p;
    }
    tail = NULL;
    num_of_items = 0;
}  // Time complexity: O(n)
```

- **Linked list does not support index.**

  → **In a vector, the `[i]` syntax (`at()` function) gives you $O(1)$ time complexity to access the item at index `i`.**

- **How to iterate through a `List`?**

  → **You cannot use `D_Node` for iteration.**

```
private:
    /** A doubly-linked list node */
    struct D_Node {
        // Data fields
        T data;  // Stores some data in the node.
        D_Node* next;  // A pointer to the next node in the list
        D_Node* prev;  // A pointer to the previous node in the list
        // Constructor
        D_Node(const T&);
    };
```

  - **There is no `D_Node` class outside the `List` class.**

  → **People use another data structure to iterate through linked lists.**

  The implemented data structure that be used to iterate through linked lists is called iterator.