Linked List

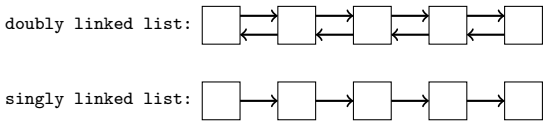
Introduction

A *linked list* is an alternative data storage structure that provides efficient insertion and removal operations with a constant worst-case runtime $O(1)$ in certain scenarios. Unlike arrays, which store elements in contiguous memory and provide random access, linked lists consist of independent memory nodes connected via pointers.

Because linked lists dynamically allocate memory, they only use the space needed without reserving extra memory for future elements. However, a drawback is that traversing a linked list requires sequential access from the head, as elements do not have direct indices.

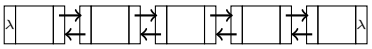
Nodes & Linked Lists

A *node* is the fundamental building block of a linked list. Each node consists of *data* (which holds the actual value) and at least one *link* (node pointers). Sequential linked lists are commonly constructed as a *singly-linked* or *doubly-linked* list.

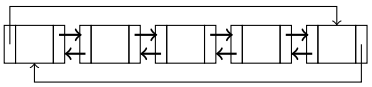


A singly linked list consists of nodes with a single link that points to the next node; traversal is one-directional. Meanwhile, a doubly linked list contains two links per node; one points to the next node, while the other points to the previous node. It allows traversal in both directions. Additionally, a doubly-linked list can be reduced to a singly-linked list by using only one of its links.

A linked list can be constructed into either a *null-terminated* or a *circular* linked list. A null-terminated doubly linked list makes the previous link of the first node and the next link of the last node both null.



Meanwhile, a circular doubly linked list makes the previous link of the first node its last node and the next link of the last node its first node; thus, forming a loop.



Node Creation

To create a new node, a constructor is used with the new operator as follows:

```
new Node<type>(value)
```

where *type* is the type of the node's data and *value* is the content of its data. Constructors, typically, initialize the links of the node to nullptr.

Linked List Traversal

Traversing a linked list requires using pointers instead of indices (like in arrays). To ensure the list always has a pointer pointing to it, always use a temporary node to traverse it. For null-terminated lists, traversals are of the form:

- **Forward Traversal (Singly or Doubly Linked List)**

```
for(Node<T>* t = node;t != nullptr;t = t->next) {//operation on t->data}
```

- **Backward Traversal (Doubly Linked List)**

```
for(Node<T>* t = node;t != nullptr;t = t->prev) {//operation on t->data}
```

And to get to either end of the list:

- **Finding the Last Node**: If the list is not empty (equal to null),

```
for(Node<T>* t = node;t->next != nullptr;t = t->next);
```

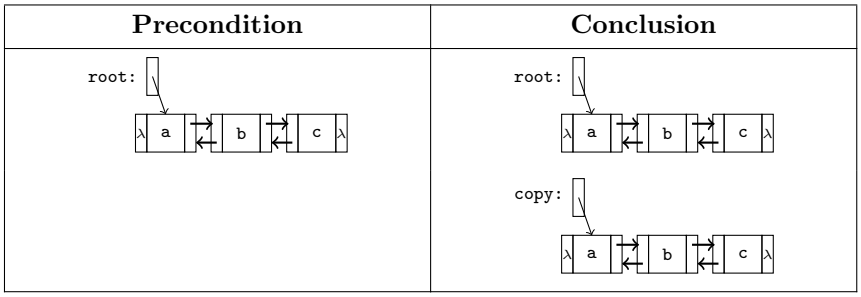
- **Finding the First Node (Doubly Linked List)**: If the list is not empty,

```
for(Node<T>* t = node;t->prev != nullptr;t = t->prev);
```

Since a circular linked list is a loop, a traversal ends when it returns to a particular node; it is only null when it is empty.

Copying a Linked List

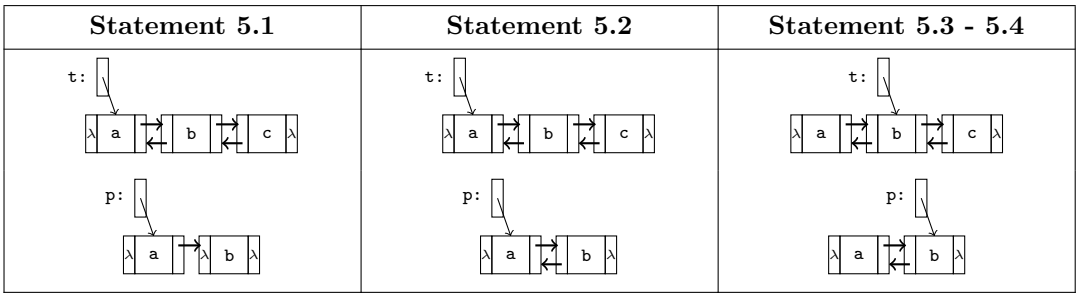
A deep copy is required to create an entirely new linked list with duplicated data to ensure the newly created list does not reference old memory locations and avoid irregular behaviors.



The algorithm is:

```
Copy(rt)
1. if rt is null, then
   1. return null.
2. cp ← new Node(rt.data).
3. t ← rt.
4. p ← cp.
5. while t.next ≠ null do
   1. p.next ← new Node(t.next.data).
   2. p.next.previous ← p.
   3. t ← t.next.
   4. p ← p.next.
6. return cp.
```

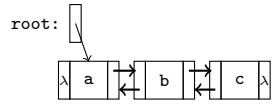
Once the first node of the copied list is created, and the original and copy are assigned to temporary node pointers, the copy process traverses through the original list as follows:



At the end of the copy process, it returns the copied list.

Deleting a Linked List

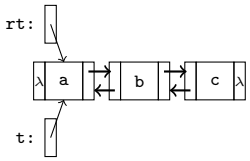
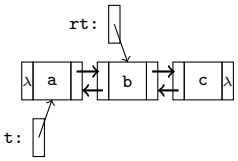
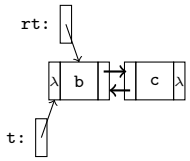
To delete a linked list, each node must be removed and deallocated one at a time to prevent memory leaks.

Precondition	Conclusion
	

For the null-terminated linked list, its algorithm is:

```
Delete(rt)
1. while rt is not null do
   1. t ← rt.
   2. rt ← rt.next.
   3. deallocate t.
   4. t ← null.
```

The delete process is as follows:

Statement 1.1	Statement 1.2	Statements 1.3 - 1.4
		

A doubly-linked list does not need to worry about its previous links since all their nodes are deallocated from the forward traversal.