CS 225 - Lecture 5

Scribe : Harsha Srimath Tirumala

1 Learning Goals

↪ Implement Insert, Random Access and Remove operations

↪ Pointers vs reference-to-pointers

↪ Linked List Insert/Delete runtimes

2 Insert(data, index)

↪ Requirement : In order to insert *data* at position *index* of the list, we need:

- Previous node (*index* - 1) to point at new node with *data*
- New node to point to the node originally at position *index*

↪ Since _index returns a **ListNode* &**, it is easy to meet both these requirements.

1. Get reference to previous node's next	ListNode* &*curr* = **_index(index)**
2. Create new ListNode	ListNode* *tmp* = new ListNode(**data**)
3. Update new ListNode's next	*tmp*→next = *curr*
4. Modify previous node to point to new ListNode	*tmp* = *curr*

↪ Runtime : Index ($O(n)$) + Create new ListNode ($O(1)$) + Update-next ($O(1)$) + Modify link ($O(1)$) = **O**(**n**)

3 List Random Access [ ]

↪ Random access helps support operations like querying and modifying data within the list. (It is one of the minimal set of operations)

↪ Design choice - Return type **T &** supports getValue() as well as setValue()

↪ Runtime : $O(n)$

Random Access (getValue)

```
template <typename T>
T & List<T>::operator [ ] (unsigned index) {
    ListNode *& tmp = _index(index);
        return tmp -> data;
}
```

4 Remove

Remove can have three different input parameters :

- Remove by position - Remove(unsigned index) Runtime - $O(n)$
- Remove a specific node - Remove(ListNode * &) Runtime - $O(1)$
- Remove by value - Remove(T & data) Runtime - $O(n)$

↪ Memory Leaks - To prevent memory leaks, make sure to **delete** the removed node from memory.

Table 1: Linked list runtimes

	@Front	@RefPointer	@Index
Insert	$O(1)$	$O(1)$	$O(n)$
Delete	$O(1)$	$O(1)$	$O(n)$