Linked Lists, Stacks, and Queues

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

In a nutshell, a data structure describes how data are stored in memory, in order to facilitate certain operations. In all the problems we have studied so far, the input set of elements is always stored in an array—this is the only data structure (albeit a simple one) we have so far.

In this lecture, we will learn another simple data structure—the linked list—for managing a set. Then, we will utilize a linked list to implement two other slightly more sophisticated structures: the stack and the queue.

As mentioned earlier, so far we have been using an array to store a set of elements. Recall that an array is a sequence of consecutive memory cells. This requires that its length ℓ be specified at the time it is created, which creates two issues:

- It is not generally possible to increase the length as we wish, because the memory cells immediately after the array may have been allocated for other purposes.
- Sometimes we may want to reduce the space consumption by shrinking an array, if the set has some elements removed. It is not easy to achieve the purpose.

Linked List

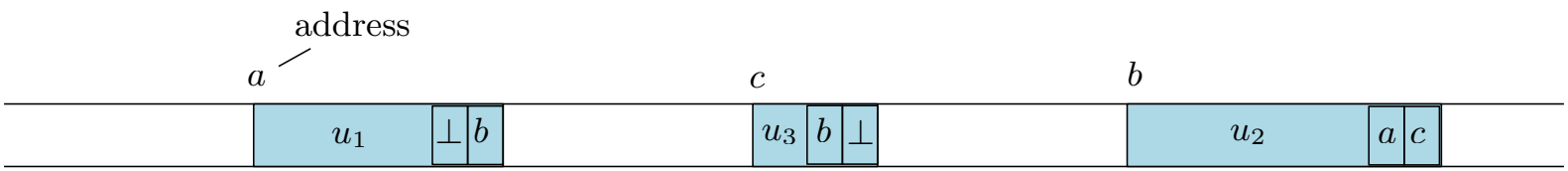
Next, we introduce the linked list, which is a sequence of nodes such that:

- The information of each node is stored in consecutive memory cells. The node's address is the address of the first cell.
- Every node stores pointers to its succeeding and preceding nodes (if they exist), where a pointer to a node u is simply the address of u.
  - We will refer to the pointer to the preceding node as the back-pointer, and the one to the succeeding node as the next-pointer.

The first node of a linked list is called the head, and the last node is called the tail.

Linked List

The figure below shows the memory contents of a linked list of three nodes u_1, u_2, u_3, whose addresses are a, b, and c, respectively.

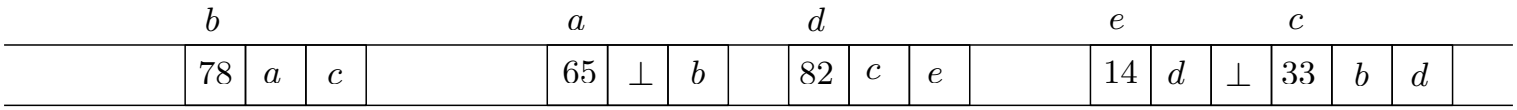


The back-pointer of node u_1 (the head) is `nil`, denoted by $\perp$. The next-pointer of u_3 (the tail) is also `nil`.

Each node can be stored anywhere in memory. Furthermore, the addresses of the nodes do **not** need to follow the ordering of the nodes in the linked list.

Example

Below is a possible linked for storing a set of 5 integers $\{14, 65, 78, 33, 82\}$.



The linked list stores the sequence $(65, 78, 33, 82, 14)$, as shown in the logical view below:

Two (Simple) Facts about a Linked List

Suppose that we use a linked list to store a set S of n integers (one node per integer).

Fact 1: The linked list uses $O(n)$ space, namely, it occupies $O(n)$ memory cells.

Fact 2: Starting from the head node, we can enumerate all the integers in S in $O(n)$ time.

A major advantage of using a linked list to manage a set S is that it supports updates to the set including:

- Insertion: Add a new element to S.
- Deletion: Remove an existing element from S.

Insertion into a Linked List

To insert a new element e into S, we simply append e to the linked list:

1. Identify the tail node u.
2. Create a new node u_{new}. Store e in u_{new}.
3. Set the next-pointer of u to the address of u_{new}.
4. Set the back-pointer of u_{new} to the address of u.

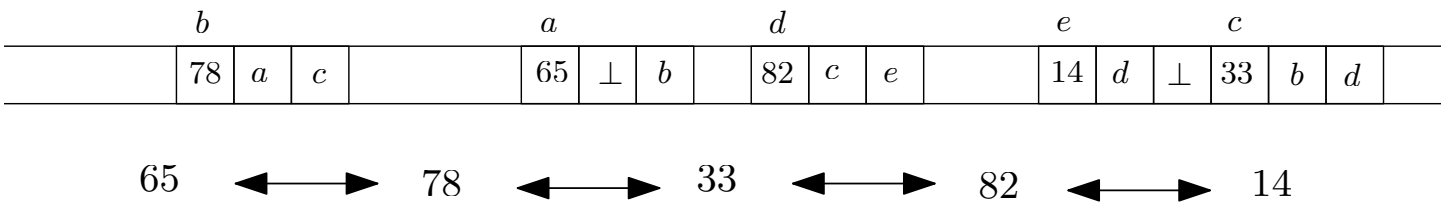
After the insertion, u_{new} becomes the new tail of the linked list.

Time of Insertion

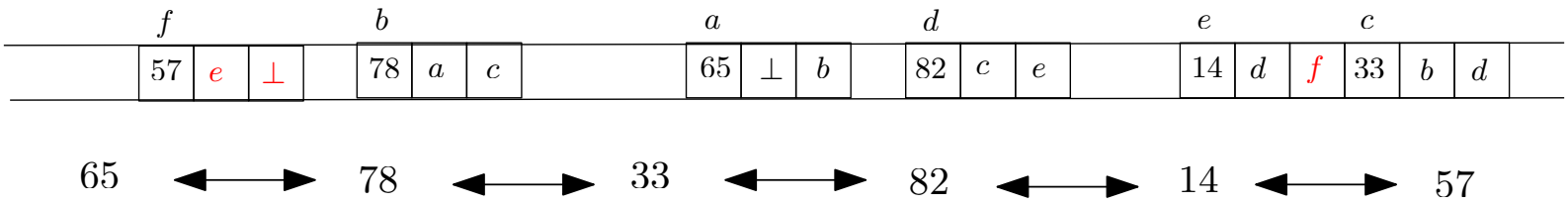
If we already know the address of the tail node, an insertion takes $O(1)$ time.

Example

Before the insertion:



After inserting 57:



Remember: the address of the new node can be anywhere in the memory.

Deletion from a Linked List

Given a pointer to (i.e., the address of) a node u in the linked list, we can delete it as follows:

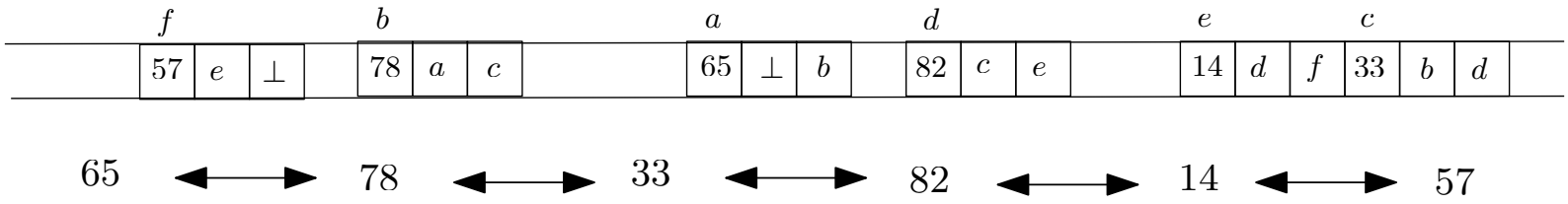
1. Identify the preceding node u_{prec} of u.
2. Identify the succeeding node u_{succ} of u.
3. Set the next-pointer of u_{prec} to the address of u_{succ}.
4. Set the back-pointer of u_{succ} to the address of u_{prec}.
5. Free up the memory of u.

Time of Deletion

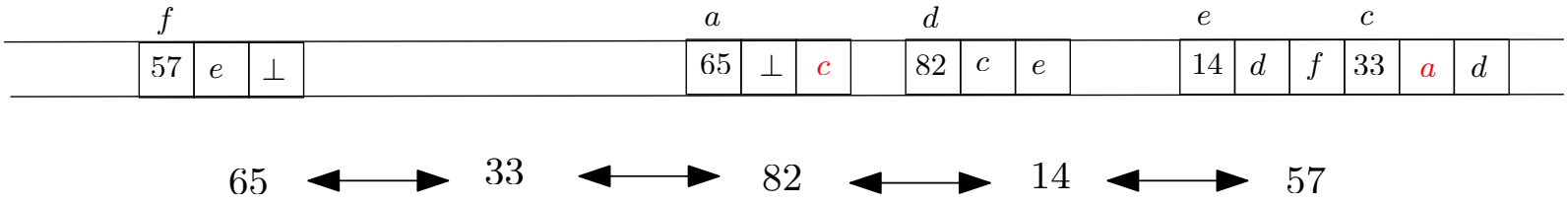
Obviously $O(1)$.

Example

Before the deletion:



After deleting 78:



Remember: the address of the new node can be anywhere in the memory.

Performance Guarantees of the Linked List

When used to manage a set of n integers:

- Space $O(n)$
- Insertion $O(1)$ time: if the address of the tail is known.
- Deletion $O(1)$ time: if the address of the node to be removed is known.
- $O(n)$ time to enumerate all the elements in the set: if the address of the head/tail is known.

Remark: We have explained how to perform an insertion by appending it to the end. Since the node ordering in a linked list can be arbitrary, there are many other ways of performing an insertion in $O(1)$ time. For example, you can do so by inserting the new element at the beginning of the linked list (think: how?), or by doing the insertion after an existing element (provided that the element's address is known; think: how?).

Next, we will deploy the linked list to implement two data structures: stack and queue.

Stack

The stack on a set S of n elements supports two operations:

- `Push`(e): Inserts a new element e into S.
- `Pop`: Removes the most recently inserted element from S, and returns it.

In other words, a stack obeys the rule of First-In-Last-Out (FILO).

Example

At the beginning, the stack is empty. Consider the following sequence of operations:

- Push(35): $S = \{35\}$.
- Push(23): $S = \{35, 23\}$.
- Push(79): $S = \{35, 23, 79\}$.
- Pop: Returns 79, and removes it from S. Now $S = \{35, 23\}$.
- Pop: Returns 23, and removes it from S. Now $S = \{35\}$.
- Push(47): $S = \{35, 47\}$.
- Pop: Returns 47, and removes it from S. Now $S = \{35\}$.

Implementing a Stack with a Linked List

At all times, we store the elements of the underlying set S in a linked list L.

`Push`(e): Insert e at the end of L.
`Pop`: Delete the tail node of L, and return the element stored in the tail.

At all times, we keep track of the address of the tail node.

Guarantees:

- $O(n)$ space consumption, where n is the size of S.
- `Push` in $O(1)$ time.
- `Pop` in $O(1)$ time.

Queue

The queue on a set S of n elements supports two operations:

- `En-queue`(e): Inserts a new element e into S.
- `De-queue`: Removes the least recently inserted element from S, and returns it.

Obeys First-In-First-Out (FIFO).

Example

At the beginning, the queue is empty. Consider the following sequence of operations:

- `En-queue`(35): $S = \{35\}$.
- `En-queue`(23): $S = \{35, 23\}$.
- `En-queue`(79): $S = \{35, 23, 79\}$.
- `De-queue`: Returns 35, and removes it from S. Now $S = \{23, 79\}$.
- `De-queue`: Returns 23, and removes it from S. Now $S = \{79\}$.
- `En-queue`(47): $S = \{79, 47\}$.
- `De-queue`: Returns 79, and removes it from S. Now $S = \{47\}$.

Implementing a Queue with a Linked List

At all times, we store the elements of the underlying set S in a linked list L.

`En-queue`(e): Insert e at the end of L.
`De-queue`: Delete the head node of L, and return the element stored in the head.

At all times, we keep track of the addresses of the head and tail nodes.

Guarantees:

- $O(n)$ space consumption, where n is the size of S.
- `En-queue` in $O(1)$ time.
- `De-queue` in $O(1)$ time.

At this moment, you should have developed a stronger sense of what is a data structure, and of what its theoretical guarantees may look like. In general, a data structure stores a set of elements so that some operations can be performed efficiently. In terms of guarantees, we typically care about:

- Its space consumption: how many memory cells does it occupy in the worst case.
- The worst-case running time of each of its operations.