Chapter 4 – Lists, Stacks, and Queues

In this chapter, we'll consider:

- Singly Linked Lists
- Doubly Linked Lists
- Stacks
- Queues
- Deques (Double-Ended Queues)

1

Introduction

- Arrays are useful in many applications but suffer from two significant limitations
  - The size of the array must be known at the time the code is compiled
  - The elements of the array are the same distance apart in memory, requiring potentially extensive shifting when inserting a new element
- This can be overcome by using ***linked lists***, collections of independent memory locations (***nodes***) that store data and links to other nodes
- Moving between the nodes is accomplished by following the links, which are the addresses of the nodes
- There are numerous ways to implement linked lists, but the most common utilizes pointers, providing great flexibility

2

Singly Linked Lists

- If a node contains a pointer to another node, we can string together any number of nodes, and need only a single variable to access the sequence
- In its simplest form, each node is composed of a datum and a link (the address) to the next node in the sequence
- This is called a ***singly linked list***

3

Singly Linked Lists

```
// class definition for a Singly-Linked List with
// an integer "payload" in the node
class IntSLLNode {
public:
    IntSLLNode() {
        next = NULL;
    }
    IntSLLNode(int i, IntSLLNode *in = 0){
        info = i; next = in;
    }
 private:
     int info;
     IntSLLNode *next;
}
```

- As can be seen here, a node consists of two data members, **`info`** and **`next`**

4

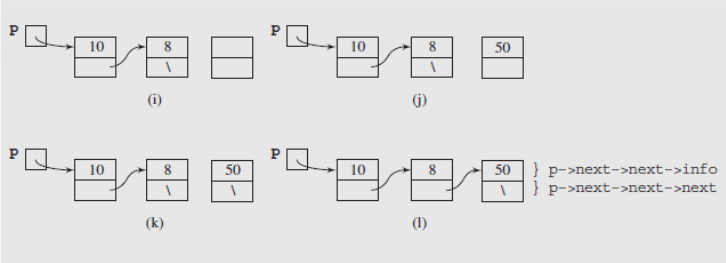
Singly Linked Lists

- The **`info`** member stores the node's "payload"; the **`next`** member points to the next node in the list
- Notice that the definition refers to the class itself, because the **`next`** pointer points to a node of the same type being defined
- Objects that contain this type of reference are called ***self-referential objects***
- The definition also contains two constructors:
  - One sets the next pointer to **`NULL`** and leaves its "payload" undefined
  - The other initializes both members in the node
- The remainder of the code shows how a simple linked list can be created using this definition

5

5

Singly Linked Lists



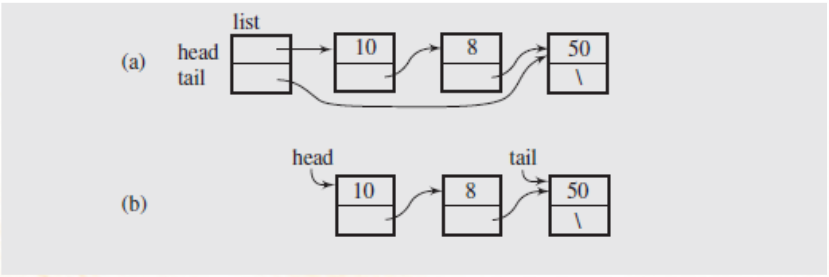
- This example illustrates a disadvantage of singly-linked lists: the longer the list, the longer the chain of **`next`** pointers that need to be followed to a given node
- This reduces flexibility and is prone to errors – and a single error could "break" the list!
- An alternative is to use an additional pointer to the end of the list, by building a class **`IntSSLList`** for a list, which allows us to define pointers to the **`head`** and the **`tail`** of the list

6

6

Singly Linked Lists

- The code uses two classes
  - **`IntSLLNode`**, which defines each nodes in the list
  - **`IntSLList`**, which defines two pointers, **`head`** and **`tail`**, as well as additional member functions to manipulate the list as a whole



7

7

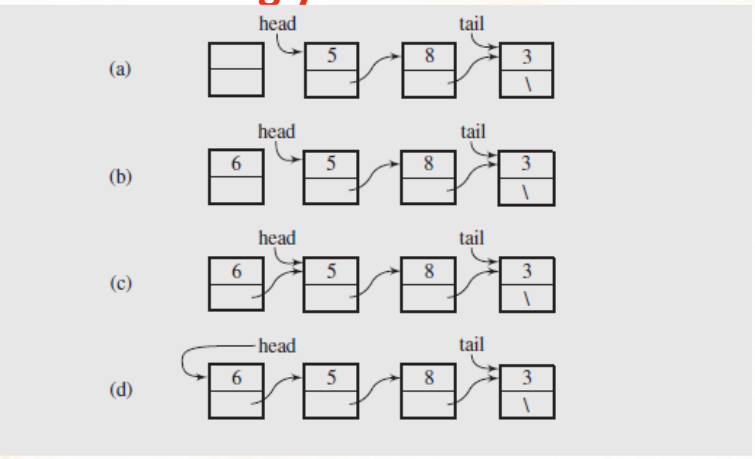
Singly Linked Lists - Insertion

- The structure of the list becomes more apparent now, and being able to get to the end of the list in O(1) time is more efficient than "walking" the entire list in O(n) time
- Let's consider some common operations on this type of list
- Inserting a node at the beginning of a list (called prepending) is straightforward:
  - First, the new node is created
  - The **`info`** member of the new node is set to its "payload" value
  - The **`next`** member is initialized to point to the previous first node in the list, which is stored as the current value of **`head`**
  - **`head`** is then updated to point to the new node

8

8

Singly Linked Lists



- Note that if the list was previously empty, inserting a new node would result in **head** and **tail** pointing to the new node
- **head** and **tail** in an empty list are both set to **0** or **NULL**

9

9

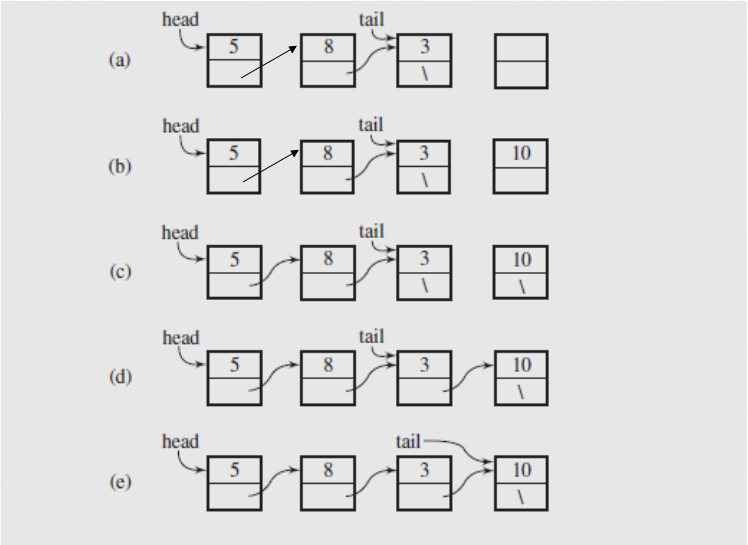
Singly Linked Lists

- Inserting a node at the end of a list (called appending) is likewise easy to accomplish:
  - The new node is created
  - The **info** member of the node is set to its "payload" value
  - The **next** member is initialized to **NULL**, since the new node will be at the end of the list (which always points to **NULL**)
  - The **next** member of the previous last node (which was set to **NULL**) is set to point to the new node
  - Since the new node is now the end of the list, the **tail** pointer of the list needs to be updated to point to the new node
  - As before, if the list was initially empty, both **head** and **tail** would be set to point to the new node

10

10

Singly Linked Lists



11

11

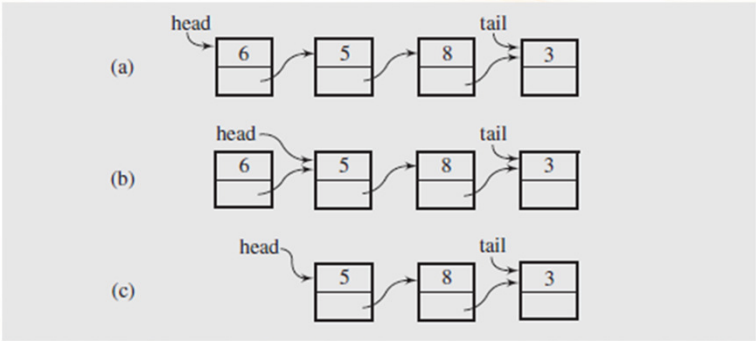
Singly Linked Lists - Deletion

- The operation of deleting a node consists "patching around" the list to remove the node, returning the "payload" stored in the node, then deallocating the memory of the node
- Again, we can consider operations at the beginning and end of the list.
- To delete at the beginning of the list:
  - We first retrieve the value stored in the first node (**`head → info`**)
  - Then we can use a temporary pointer to point to the node (so we don't lose track of it!), and set **`head`** to the value to **`head → next`**
  - Finally, the former first node can be deleted, releasing its memory

12

12

Singly Linked Lists



- Two special cases exist when carrying out this deletion
- The first arises when the list is empty -- the caller must be notified that nothing was found that could be deleted
- The second occurs when a single node is in the list, requiring that both **head** and **tail** be set to **NULL** because the list is now empty

13

13

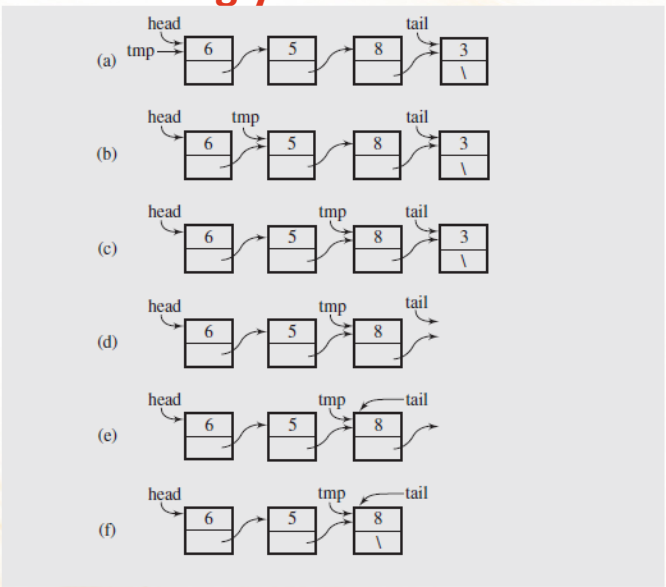
Singly Linked Lists

- Deletion at the end of a list also requires additional processing
- This is because the **tail** pointer must first be backed up to the previous node (what will be the new tail) in the list
- Since this can't be done directly, we need a temporary pointer **tmp** to traverse the entire list (!) until
  tmp → next == tail
- Once we have located the node, we can retrieve its "payload" contained in **tail → info**, delete the node, and set
  tail = tmp

14

14

Singly Linked Lists



15

15

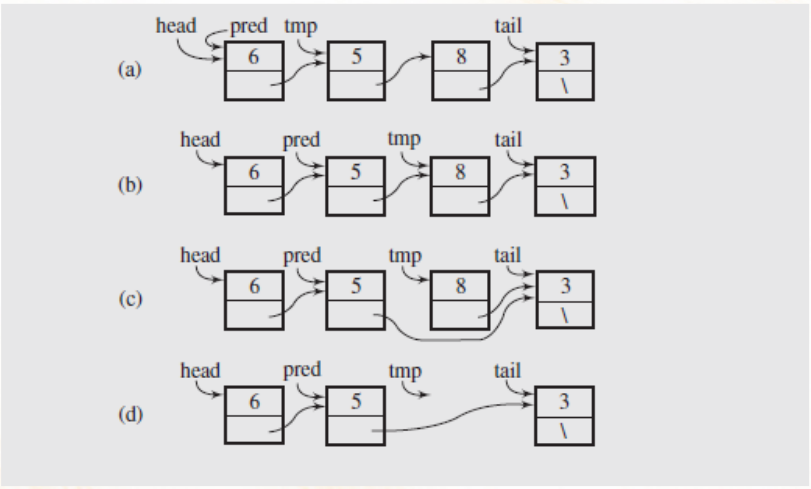
Singly Linked Lists

- So far, the deletion operations simply delete the physically first or last node in the list. But what if we want to delete a specific node based on the value of its **`info`**?
- In that case we have to first search for the specific node
- Then we "patch around" that node by linking the previous node to the following node
- To do this, we need to keep track of the previous node while we search, and we need to keep track of the node containing the target value
- This will require two pointers, which we'll call **`pred`** and **`tmp`**

16

16

Singly Linked Lists



- **pred** and **tmp** are initialized to the first and second nodes in the list
- They traverse the list until **tmp → info** matches the target value

17

17

Singly Linked Lists

- When **tmp → next** matches the value we wish to remove, we set **pred → next** to **tmp → next** which then "bypasses" the target node, allowing it to be deleted safely.
- There are several special cases to consider when this type of deletion is carried out:
  - Removing a node from an empty list
  - Trying to delete a value that isn't found in the list
  - Deleting the only node in the list (making it empty)
  - Removing the first or last node in a list that has at least two nodes

18

18

Singly Linked Lists - Searching

- The purpose of a search is to scan a linked list to find a particular value of the "payload" in **`info`**
- No modification is made to the list, so this can be done easily using just a single temporary pointer **`tmp`**
- We simply traverse the list until the **`info`** member of the node **`tmp`** points to matches the target, or until **`tmp → next`** is **`NULL`** (which means the value is not found)
- If the latter case occurs, we have reached the end of the list and the search fails
- And that pretty much sums up singly-linked lists!

19

19

Dummy Nodes in Lists

- To avoid the special cases of removing an item from a list with only one item (which makes if empty), or adding an item to an empty list, one strategy is to have the list never become "empty," even if it contains no items!
- This is done by using a "dummy node" that's always in the chain of nodes but doesn't count as an item in the list.
- The dummy node is always at the head of the list and the Head pointer points to it. The condition for an empty list can then be "Head == Tail", instead of Head == NULL.
- This means that the Head and Tail pointers are never set to NULL and having special cases that either set one (or both) to NULL, or test whether one (or both) are NULL, can be avoided

20

20

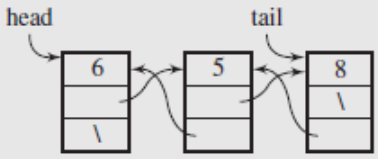
Doubly Linked Lists

- The difficulty in deleting the last node of a singly linked list points out one major limitation of that structure: We have to traverse the entire list just to get to the node before the end so it can become the new end!
- If the processing requires frequent deletions of this type, this significantly slows down operation and hurts efficiency
- To address this problem, we can redefine the node structure by adding a second pointer that points to the previous node
- Lists constructed from these nodes are called ***doubly linked lists***

21

21

Doubly Linked Lists



- The methods that manipulate doubly linked lists are more complicated than their singly linked counterparts
- However, the process is still straightforward as long as we carefully keep track of the pointers!
- We'll look at just two methods: inserting and removing a node at the end of the list

22

22

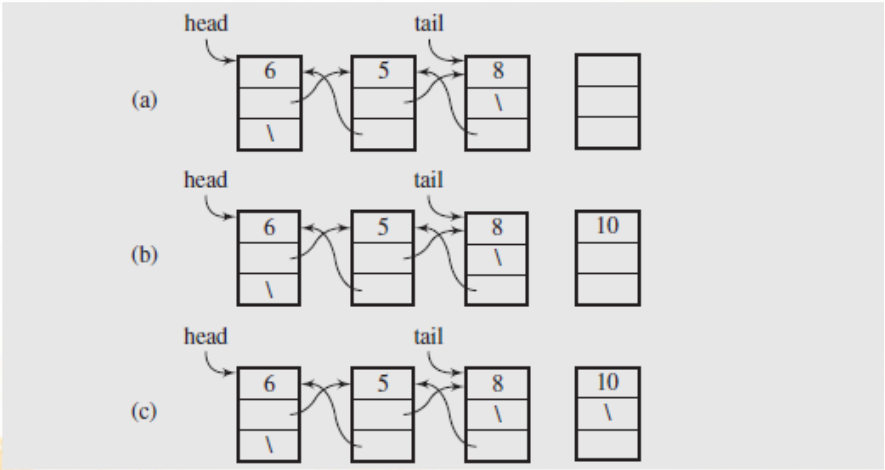
Doubly Linked Lists - Insertion

- To insert of a new node at the end of the list:
- The new node is created and the data member is initialized
- Since the node is being inserted at the end of the list, its **`next`** member is set to **`NULL`**
- The **`prev`** member of the new node is set to the current **`tail`** value of the list, to link it to the former end of the list
- The **`tail`** pointer of the list is now set to point to this new node's address
- To complete the link, the **`next`** member of the previous node is set to point to the new node

23

23

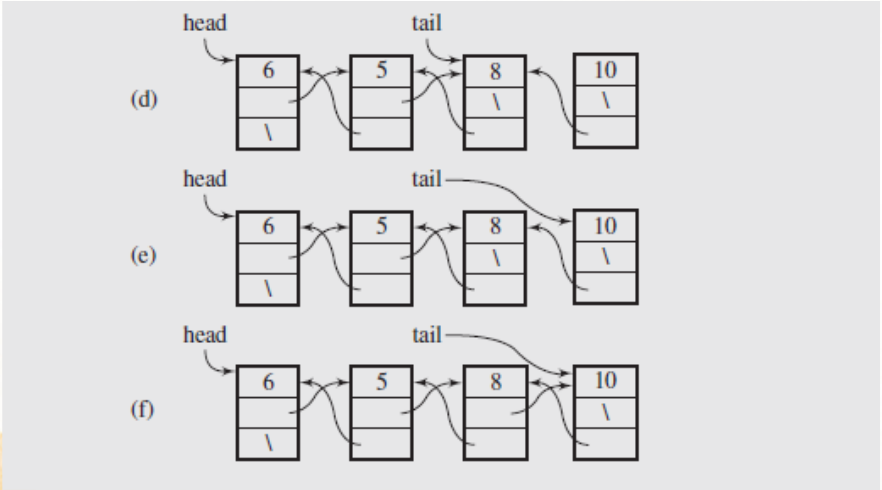
Doubly Linked Lists
Adding a node at end



24

24

Doubly Linked Lists
Adding a node at end



25

25

Doubly Linked Lists – Insertion, Deletion

- Insertion of a node into an empty list is a special case:
  - Because there is no previous node, both **head** and **tail** are set to the new node
- Deletion also has some special cases:
  - Deleting a node from the end of a doubly linked list is relatively easy, because there is a direct link to the previous node in the list. This eliminates the need to traverse the list to find the previous node
  - To do this, we simply retrieve the data member from the last, set **tail** to the node's predecessor, then delete the node and set the **next** pointer of the new last node to **NULL**

26

26

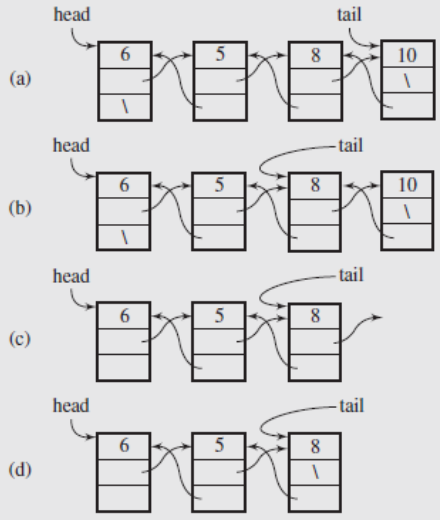
Doubly Linked Lists

- Deletion also has a couple of special cases:
  - If the node being deleted is the only node in the list, both **head** and **`tail`** need to be set to **`NULL`**
  - If the list was already empty, any attempt to delete a node should be reported to the user as having failed
  - We need to make sure we check whether any pointer is NULL before we try to dereference it (which can generate an error)
  - Alternately, if the programming language is "smart" enough to "catch" such errors, we can write code that takes advantage of this feature

27

27

Doubly Linked Lists - Deletion



28

28

Headers for Doubly Linked List Node Class

```
typedef int T;  // Just to set the type of payload

Class DLLNode {
    public:
        DLLNode() {
            next = prev = NULL;  // Generic node
        }

        DLLNode(const T& el, DLLNode *n = 0, DLLNode *p = 0) {
            info = el; next = n; prev = p;
        }
    protected:
        T info;
        DLLNode *next, *prev;
}
```

29

29

Headers for Doubly Linked List Class

```
Class DLList {
    public:
        DLList() {
            head = tail = NULL;  // Create empty list
        }
        ~DLList();

        bool isEmpty() {
            return (head == NULL);
        }

        void addToDLLHead(const T& info);
        void addToDLLTail(const T& info);
        T deleteFromDLLHead();
        T deleteFromDLLTail();
        bool removeFromDLL(const T& info);

    protected:
       DLLNode *head, *tail;
}
```

30

30

Dummy Node(s) in Doubly-Linked Lists

- For a doubly-linked list with dummy nodes, there are two possible implementations:
  - The list can have ONE dummy node, always pointed to by the Head pointer, and the list will be empty when Head == Tail
  - The list can have TWO dummy nodes -- one that Head points to, one that Tail points to, and the list will be empty when the Head dummy node is pointing to the Tail dummy node (and vice versa)

31

31

Recursion in Lists

- Remember the Racket language in CS 111, and the way it was used to process lists?

```
(define (process-list a-list)
    (cond
        [(empty? a-list) empty]
        [else (cons (process (first a-list))
                    (process-list (rest a-list)))]))
```

- The idea was to process the *first* item in the list, then to recurseively call the function on the *rest* of the list
- How many recursive calls does it require? Same as the length of the list
- Remember: sometimes it's necessary to write a recursive (private) "helper" function to do the recursion if the original function does not pass all the arguments necessary to support recursion

32

32