Linked List

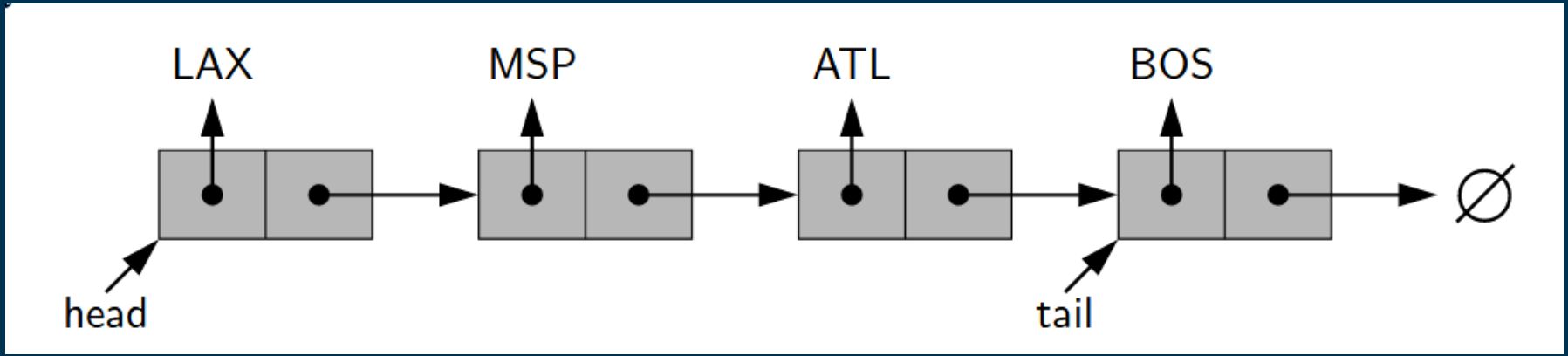
Algoritma dan Struktur Data

Dani Hidayat
1122094000014
4A

Daftar Isi

Linked list adalah salah satu struktur data dasar dalam pemrograman. Ini adalah kumpulan elemen-elemen data yang terhubung satu sama lain melalui penggunaan referensi atau pointer. Setiap elemen dalam linked list, yang disebut node, terdiri dari dua bagian utama: data dan referensi ke node berikutnya dalam urutan.

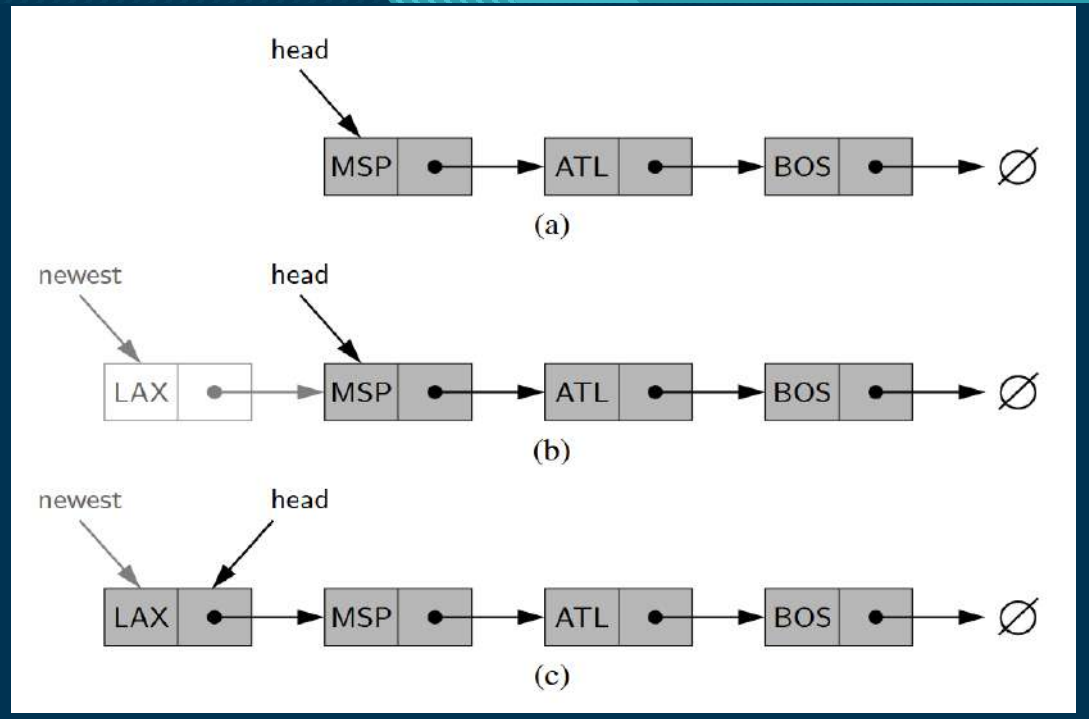
7.1 Singly Linked List



Singly linked list adalah bentuk sederhana dari *linked list* yang merupakan koleksi linear dari data, yang disebut sebagai *nodes*, dimana setiap *node* akan merujuk pada *node* lain melalui sebuah *pointer* ini juga dapat didefinisikan sebagai kumpulan *nodes* yang merepresentasikan sebuah *sequence*

7.1 Singly Linked List

Memasukkan Elemen di Head Pada Singly Linked List



Algorithm add first(L,e):

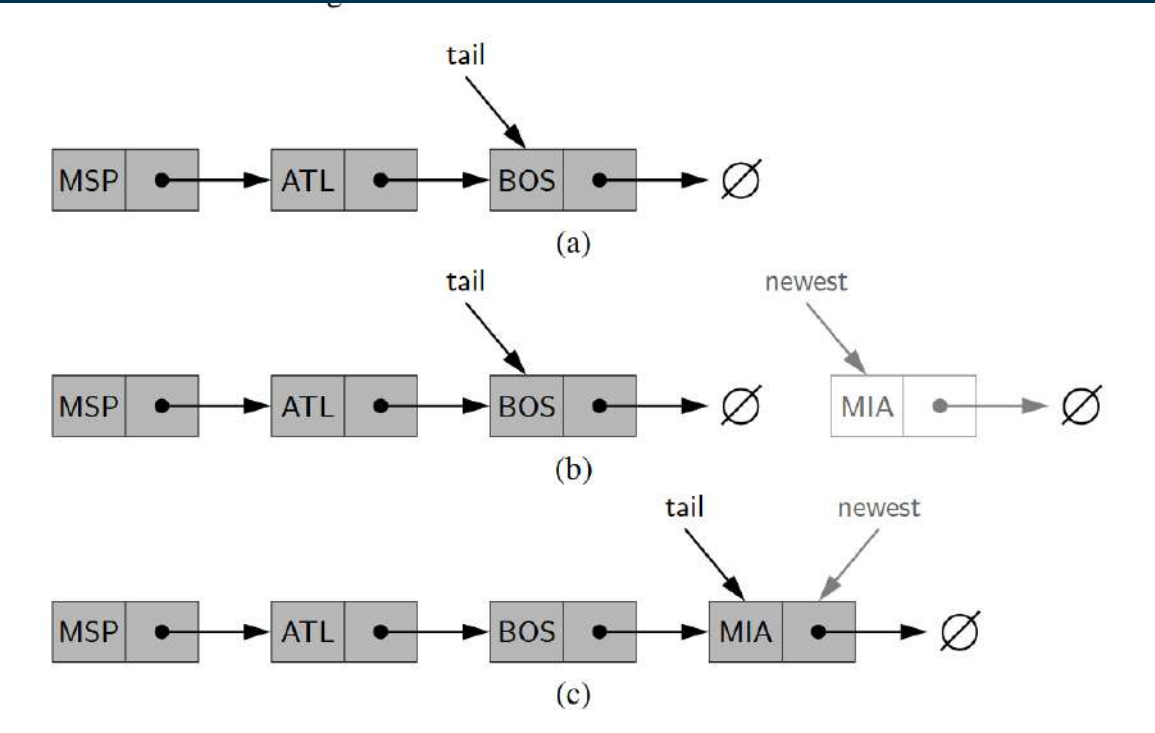
newest = Node(e) {membuat node baru yang menyimpan referensi ke elemen e}

newest.next = L.head {atur node baru yang referensi selanjutnya adalah head node yang lama}

L.head = newest {atur variabel head untuk mereferensikan node baru}

L.size = L.size+1 {menambah jumlah node}

7.1 Singly Linked List



Memasukkan Elemen di Tail Pada Singly Linked List

Algorithm add last(L,e):

newest = Node(e) {membuat node baru yang menyimpan referensi ke elemen e}

newest.next = None {atur node baru yang referensi selanjutnya adalah object}

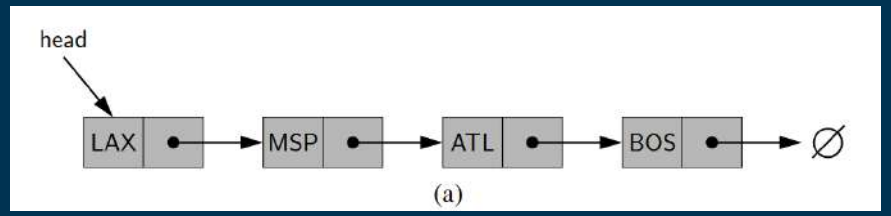
L.tail.next = newest {buat tail node yang lama menunjuk ke node baru}

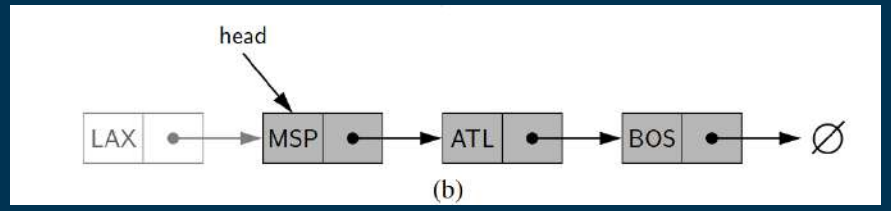
L.tail = newest {atur variabel tail untuk mereferensikan node baru}

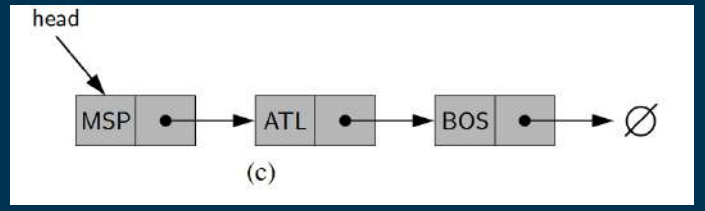
L.size = L.size+1 {menambah jumlah node}

7.1 Singly Linked List

Menghapus Elemen Pada Singly Linked List







Algorithm remove first(L):

```
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next {buat head pint ke node selanjutnya (atau None)}
    L.size = L.size-1 {kurangi jumlah node}
```

7.1.1 Implementasi Stack dengan Singly Linked List

```python
class LinkedStack:
    """LIFO Stack implementation using a singly linked list for storage."""

    #-------------------------- nested Node class --------------------------
    class Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element' , '_next' # streamline memory usage

        def __init__(self, element, next): # initialize node's fields
            self._element = element # reference to user's element
            self._next = next # reference to next node

    #------------------------------ stack methods ------------------------------
    def __init__(self):
        """Create an empty stack."""
        self._head = None # reference to the head node
        self._size = 0 # number of stack elements

    def __len__(self):
        """Return the number of elements in the stack."""
        return self._size

    def is_empty(self):
        """Return True if the stack is empty."""
        return self._size == 0

    def push(self, e):
        """Add element e to the top of the stack."""
        self._head = self._Node(e, self._head) # create and link a new node
        self._size += 1

    def top(self):
        """Return (but do not remove) the element at the top of the stack.
        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element # top of stack is at head of list

    def pop(self):
        """Remove and return the element from the top of the stack (i.e., LIFO).
        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        answer = self._head._element
        self._head = self._head._next # bypass the former top node
        self._size -= 1
        return answer
```

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

Performa dari implementasi LinkedStack kami. Semua batasan adalah kasus terburuk dan space usage adalah $O(n)$, di mana n adalah jumlah elemen saat ini di dalam stack.

7.1.2 Implementasi Queue dengan Singly Linked List

LinkedQueue harus selalu mempertahankan referensi tail (ekor), sedangkan LinkedStack tidak membutuhkannya. Menghapus elemen head pada queue dengan 1 elemen berarti menghapus tail sekaligus. Oleh karena itu, perlu diatur self.tail menjadi None untuk menjaga konsistensi. Menambahkan elemen baru (enqueue) pada queue juga sedikit berbeda. Node baru menjadi tail baru, tetapi jika node tersebut menjadi elemen tunggal di list, maka node tersebut juga menjadi head baru. Sebaliknya, jika ada node lain, node baru harus ditautkan setelah node tail yang ada. Kinerja LinkedQueue mirip dengan LinkedStack

```python
class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage."""

    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element' , '_next' # streamline memory usage

        def __init__(self, element, next): # initialize node's fields
            self._element = element # reference to user's element
            self._next = next # reference to next node

    def __init__(self):
        """Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 0 # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue."""
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._head._element # front aligned with head of list

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        if self.is_empty(): # special case as queue is empty
            self._tail = None # removed head had been the tail
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = self._Node(e, None) # node will be new tail node
        if self.is_empty():
            self._head = newest # special case: previously empty
        else:
            self._tail._next = newest
        self._tail = newest # update reference to tail node
        self._size += 1
```

7.2 Circularly Linked List

Berbeda dengan array, pada linked list, terdapat konsep yang lebih nyata tentang circularly linked list. Pada circularly linked list, bagian akhir (tail) dari list memiliki referensi "next" yang menunjuk kembali ke bagian awal (head) list. Linked list sirkular menyediakan model yang lebih umum daripada linked list standar untuk kumpulan data yang bersifat siklis, yaitu yang tidak memiliki awal dan akhir tertentu.

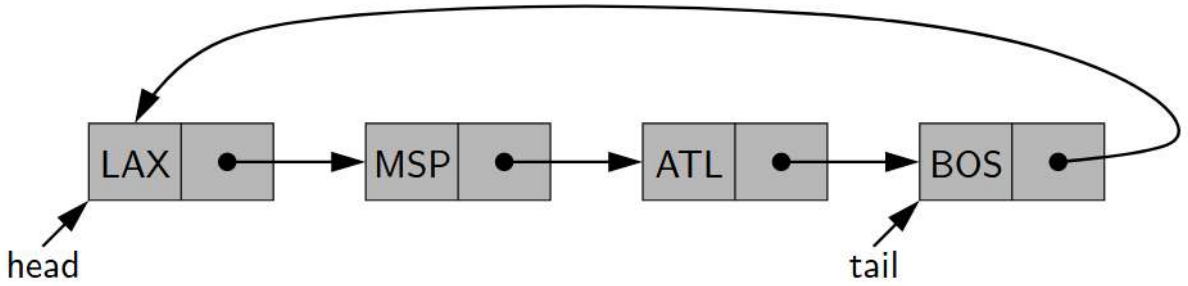


Figure 7.7: Example of a singly linked list with circular structure.

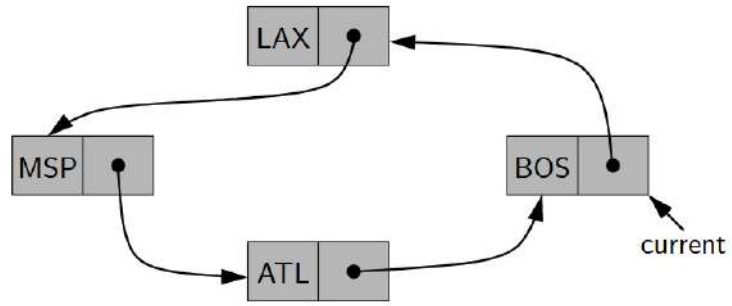


Figure 7.8: Example of a circular linked list, with current denoting a reference to a select node.

7.2.1 Round-Robin Schedulers

Circularly linked lists berguna dalam skenario seperti round-robin scheduling di mana elemen perlu diiterasi secara circular. Dalam round-robin schedulers setiap elemen "diservis" dengan melakukan tindakan tertentu pada elemen tersebut.

A round-robin scheduler bisa diimplementasi dengan general queue ADT, Dengan berulangkali melakukan steps queue Q (see Figure 7.9):

1. e = Q.dequeue()
2. Service element e
3. Q.enqueue(e)

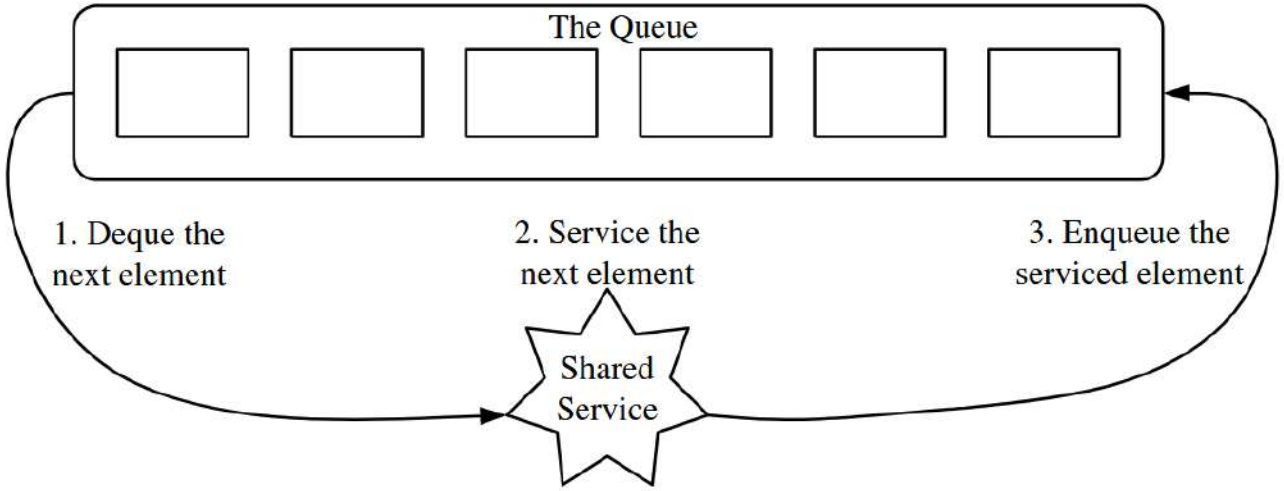


Figure 7.9: The three iterative steps for round-robin scheduling using a queue.

7.2.2 Implementasi Queue dengan Circularly Linked List

```python
class CircularQueue:
    """Queue implementation using circularly linked list for storage."""

    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        # (omitted here; identical to that of LinkedStack._Node)

    def __init__(self):
        """Create an empty queue."""
        self._tail = None # will represent tail of queue
        self._size = 0 # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty("Queue is empty")
        head = self._tail._next
        return head._element
```

```python
def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).
    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
        raise Empty('Queue is empty')
    oldhead = self._tail._next
    if self._size == 1: # removing only element
        self._tail = None # queue becomes empty
    else:
        self._tail._next = oldhead._next # bypass the old head
    self._size -= 1
    return oldhead._element

def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None) # node will be new tail node
    if self.is_empty():
        newest._next = newest # initialize circularly
    else:
        newest._next = self._tail._next # new node points to head
        self._tail._next = newest # old tail points to new node
    self._tail = newest # new node becomes the tail
    self._size += 1

def rotate(self):
    """Rotate front element to the back of the queue."""
    if self._size > 0:
        self._tail = self._tail._next # old head becomes new tail
```

Class ini hanya memiliki dua variabel instan, yaitu tail, yang merupakan referensi ke node tail (atau None jika kosong), dan size, yang menunjukkan jumlah elemen saat ini di dalam antrian. Ketika operasi melibatkan bagian depan antrian, kita menggunakan self._tail._next untuk mengidentifikasi node head antrian. Ketika method enqueue dipanggil, node baru ditempatkan tepat setelah tail tetapi sebelum head saat ini, kemudian node baru tersebut menjadi tail yang baru.

7.3 Doubly Linked List

Untuk memberikan simetri yang lebih baik, kita mendefinisikan sebuah linked list di mana setiap node memiliki referensi eksplisit ke node sebelumnya dan referensi ke node setelahnya. Struktur seperti ini dikenal sebagai **doubly linked list** (daftar tertaut ganda). List ini memungkinkan variasi yang lebih besar dari operasi update waktu-*O(1)*, termasuk penyisipan dan penghapusan pada posisi sembarang di dalam list. Kita tetap menggunakan istilah *"next"* untuk referensi ke node yang mengikuti node lainnya, dan kita memperkenalkan istilah *"prev"* untuk referensi ke node yang mendahuluinya.

7.3 Doubly Linked List

Head and Trailer Sentinels

Untuk menghindari kasus-kasus khusus saat beroperasi di dekat batas-batas doubly linked list, menambahkan node khusus di kedua ujung list akan sangat membantu. Node tersebut berupa Node header di awal list dan Node trailer di akhir list. Node *"dummy"* ini dikenal sebagai *sentinel* (atau guard), dan tidak menyimpan elemen dari sequence utama.

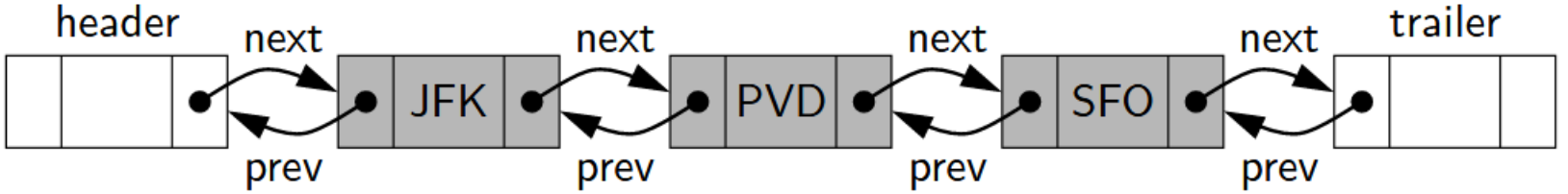


Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

7.3 Doubly Linked List

Keuntungan dari Penggunaan Sentinels

Dengan menggunakan Sentinel dalam Doubly Linked List sangat menyederhanakan logika operasi dengan Header dan trailer tidak pernah berubah, hanya node di antara keduanya yang berubah. Semua penyisipan menjadi lebih mudah karena node baru selalu ditempatkan di antara sepasang node yang ada. Selain itu, semua penghapusan dijamin terjadi pada node yang memiliki tetangga di kedua sisinya. Dengan ini, tidak ada lagi kasus khusus seperti pada LinkedQueue.

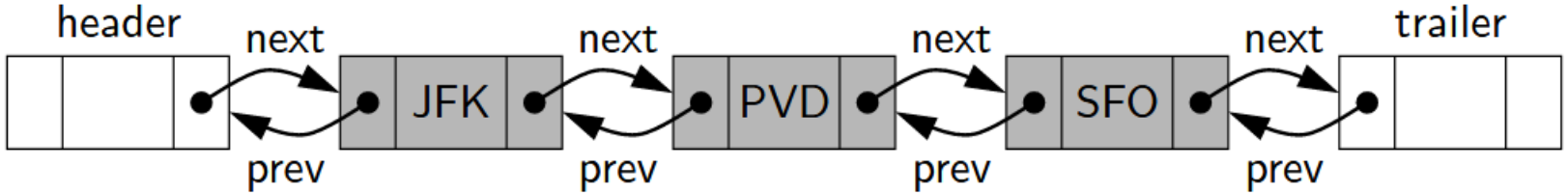


Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

7.3 Doubly Linked List

Inserting and Deleting with Doubly Linked List

Setiap penyisipan elemen pada Doubly Linked List akan disisipkan di antara dua nodes yang ada. Ketika elemen baru disisipkan di depan sequence, kita akan menambahkan node baru di antara header dan node setelah header.

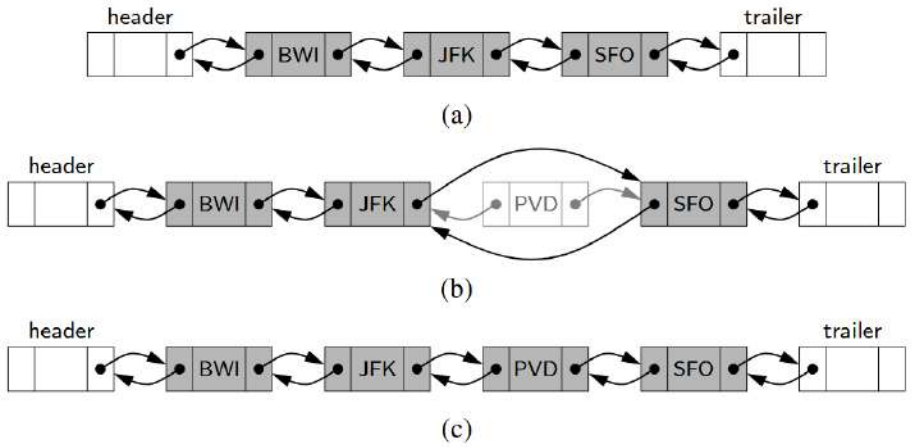


Figure 7.11: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

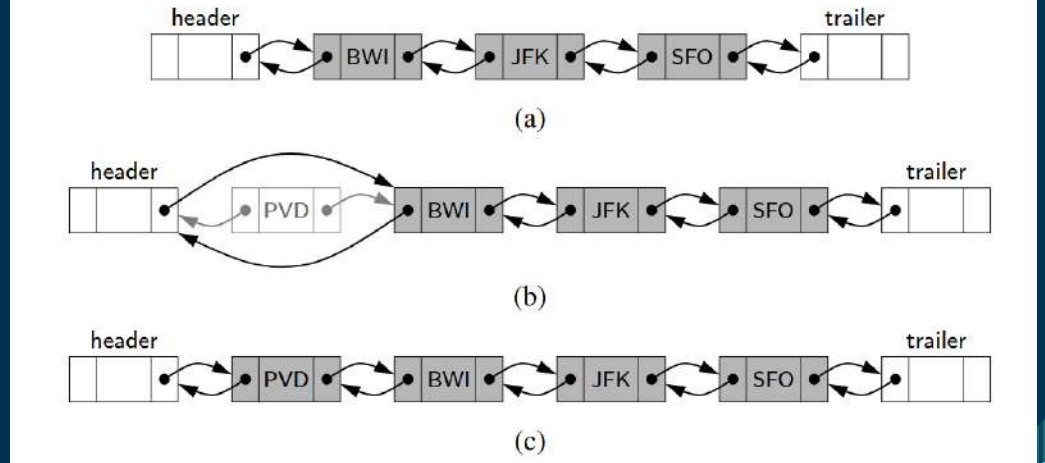


Figure 7.12: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

7.3 Doubly Linked List

Inserting and Deleting with Doubly Linked List

Kedua node tetangga dari node yang akan dihapus dihubungkan secara langsung, sehingga melewati node asli. Akibatnya, node tersebut tidak lagi dianggap sebagai bagian dari list dan dapat diambil kembali oleh sistem. Keuntungan sentinel: cara penghapusan yang sama bisa digunakan untuk elemen pertama atau terakhir sekalipun, karena elemen tersebut tetap berada di antara node lainnya.

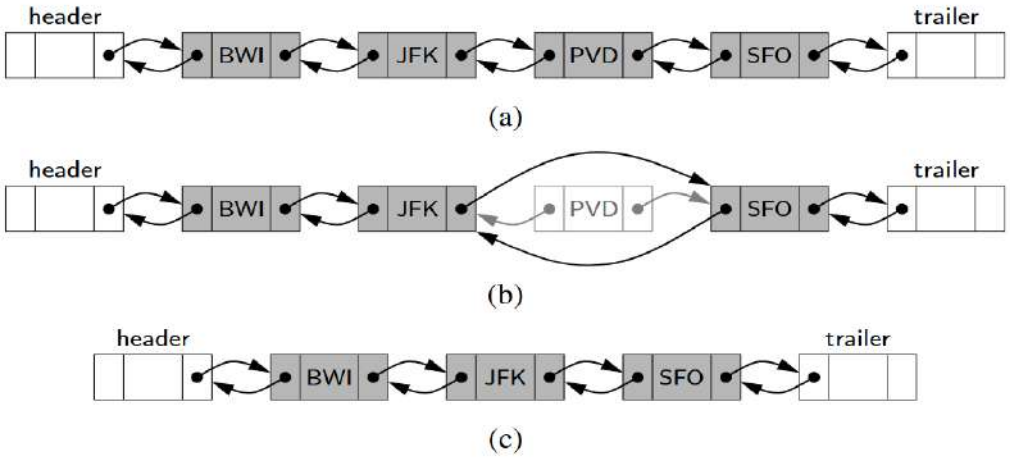


Figure 7.13: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

Implementasi Doubly Linked List

Implementasi Code Doubly Linked List Pada Python

```python
class _DoublyLinkedBase:
    """A base class providing a doubly linked list representation."""

    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        # (omitted here; see previous code fragment)

    def __init__(self):
        """Create an empty list."""
        self._header = self._Node(None, None, None)
        self._trailer = self._Node(None, None, None)
        self._header._next = self._trailer #_trailer is after _header
        self._trailer._prev = self._header #_header is before _trailer
        self._size = 0 # number of elements

    def __len__(self):
        """Return the number of elements in the list."""
        return self._size

    def is_empty(self):
        """Return True if list is empty."""
        return self._size == 0

    def _insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return new node."""
        newest = self._Node(e, predecessor, successor) # linked to neighbors
        predecessor._next = newest
        successor._prev = newest
        self._size += 1
        return newest

    def _delete_node(self, node):
        """Delete nonsentinel node from the list and return its element."""
        predecessor = node._prev
        successor = node._next
        predecessor._next = successor
        successor._prev = predecessor
        self._size -= 1
        element = node._element # record deleted element
        node._prev = node._next = node._element = None # deprecate node
        return element # return deleted element
```

Implementasi Deque dengan Doubly Linked List

```python
class LinkedDeque(_DoublyLinkedBase): # note the use of inheritance
    """Double-ended queue implementation based on a doubly linked list."""

    def first(self):
        """Return (but do not remove) the element at the front of the deque."""
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._header._next._element # real item just after header

    def last(self):
        """Return (but do not remove) the_element at the back of the deque."""
        if self.is_empty( ):
            raise Empty("Deque is empty")
        return self._trailer._prev._element # real item just before trailer

    def insert_first(self, e):
        """Add an element to the front of the deque."""
        self._insert_between(e, self._header, self._header._next) # after header

    def insert_last(self, e):
        """Add an element to the back of the deque."""
        self._insert_between(e, self._trailer._prev, self._trailer) # before trailer

    def delete_first(self):
        """Remove and return the_element from the front of the deque.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._delete_node(self._header._next) # use inherited method

    def delete_last(self):
        """Remove and return the_element from the back of the deque.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._delete_node(self._trailer._prev) # use inherited method
```

Thank You