Lab Manual

Data Structures & Application Lab

S.Y. B. Tech

MIT | Academy of Engineering

Department of Information Technology

MIT ACADEMY OF ENGINEERING, Alandi (D). PUNE

Savitribai Phule Pune University

2017-2018

Academic Year : 2017-18	MIT Academy of Engineering Alandi (D), Pune – 412 105.	**Format No.:** MAE/IT/F/35
Semester : IV		**Page No. : 1** of **1**
Class : **S. Y. B. Tech**	Department of Information Technology	**Rev. No. :** 01
	LAB MANUAL	**Rev. Date :** 03/12/2012

Subject : Data Structures & Application Lab

Work Load		**Marking Scheme**		
Theory	**Practical**			
--	4 hrs / week	Term Work	Practical	Oral
		--	50	--

Sr. No.	**Name of Assignment**	**Time Span (No. of Hrs)**
1	**Project 01: Create a Word Dictionary using suitable data structure** A. Create word dictionary using Primitive data structure like list, dictionary. B. Create word dictionary using data structure using file. C. Create word dictionary using data structure using Linked List.	12
2	**Project 02:** A. Create a queue which will maintain the list of customer at restaurant/ Bank. B. Create a Priority Queue for Patients in Hospital.	12
3	**Project 03: Simulate an air traffic controller using suitable data structure**	12
4	**Project 04: Implement a Ticket Checker application suitable data structure**	14

Prepared by & Subject Incharge :	**Reviewed By :** **DRC Members**	**Approved By:**
1. Mrs. Seema Mandlik	1. Mrs. J.A. Patil	Prof. S.M. Bhagat
2. Mrs. Sunil Mhamane		HOD

Project- 01

Phase 01 : **Create a Word Dictionary using suitable data structure**

A. Create word dictionary using Primitive data structure like list, Dictionary.

B. Create word dictionary using data structure using file.

C. Create word dictionary using data structure using Linked List.

The required functions for given projects are:

1. Add a word into dictionary
2. Search a word from Dictionary.
3. Show the Dictionary.
4. Calculate time complexity of search module.
5. Exit.

After completion of this assignment students will able to:

Indentify suitable data structure for any given application and use of primitive data structures and non primitive data structures like List, Dictionary, Linked List etc,

Relevant Theory / Literature Survey:

Definition: A **data structure** is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently.
OR
Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.

Basic types of Data Structures:

1. Primitive Data Structures:
Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char all are data structures. They are known as **Primitive Data Structures.**

2. Abstract Data Structures (Non Primitive):
Data Structures, which are used to store large and connected data, are called as Abstract Data Structures. All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. Non-primitive data structures are more complicated data structures and are derived from primitive data structures.

- Linked List
- Tree
- Graph
- Stack, Queue etc.

Basic Operations:

Following are the basic operations on Data structures.

1. Traversing- It is used to access each data item exactly once so that it can be processed.

2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

3. Inserting- It is used to add a new data item in the given collection of data items.

4. Deleting- It is used to delete an existing data item from the given collection of data items.

5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

There are quite a few data structures available. The built-in data structures are: lists, tuples, dictionaries, strings, sets.

A. List:

The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

For example −

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove () method if you do not know. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

B. Dictionary:

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

```
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

1. Access elements from a dictionary:

```
my_dict = {'name':'Jack', 'age': 26}

# Output: Jack

print(my_dict['name'])

# Output: 26

print(my_dict.get('age'))
```

2. Change or add elements in a dictionary:

```
y_dict = {'name':'Jack', 'age': 26}

# update value

my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}

print(my_dict)

# add item

my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

print( my_dict)
```

3. Delete or remove elements from a dictionary:

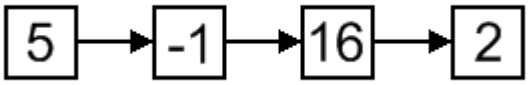
```
# create a dictionary

squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item

# Output: 16

print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}

print(squares)

# remove an arbitrary item

# Output: (1, 1)

print(squares.popitem())

# Output: {2: 4, 3: 9, 5: 25}
```

C. Linked List:

Linked list is a very important dynamic data structure. Basically, there are two types of linked list, singly-linked list and doubly-linked list. In a singly-linked list every element contains some data and a link to the next element, which allows to keep the structure. On the other hand, every node in a doubly-linked list also contains a link to the previous node. Linked list can be an underlying data structure to implement stack, queue or sorted list.

Example

Sketchy, singly-linked list can be shown like this:



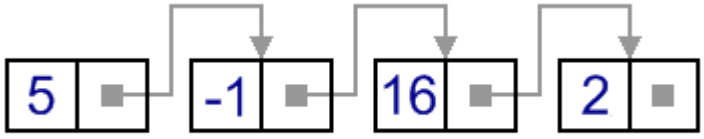
Each cell is called a **node** of a singly-linked list. First node is called **head** and it's a dedicated node. By knowing it, we can access every other node in the list. Sometimes, last node, called **tail**, is also stored in order to speed up add operation.

Singly-linked list. Internal representation.

Every node of a singly-linked list contains following information:

- a value (user's data);
- a link to the next element (auxiliary data).

Sketchy, it can be shown like this:

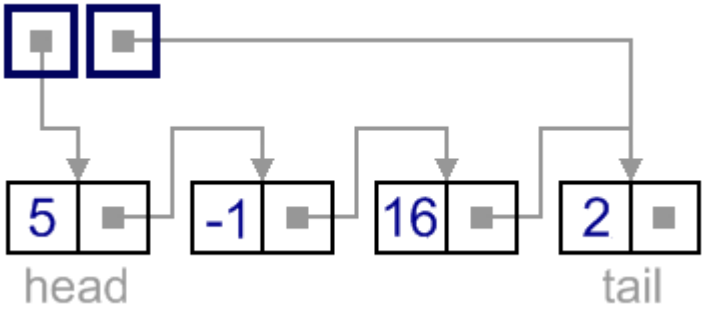


First node called **head** and no other node points to it. Link to the head is usually stored it the class, which provides an interface to the resulting data structure. For empty list, head is set to

NULL.

Also, it makes sense to store a link to the last node, called **tail**.

Though no node in the list can be accessed from the tail (because we can move forward only), it can accelerate an add operation, when adding to the end of the list. When list is big, it reduces add operation complexity essentially, while memory overhead is insignificant. Below you can see another picture, which shows the whole singly-linked list internal representation:

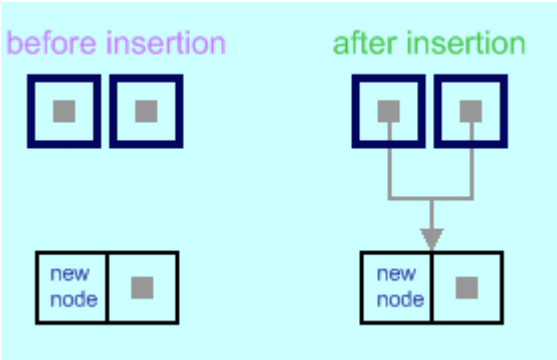


Singly-linked list. Insertion operation.

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

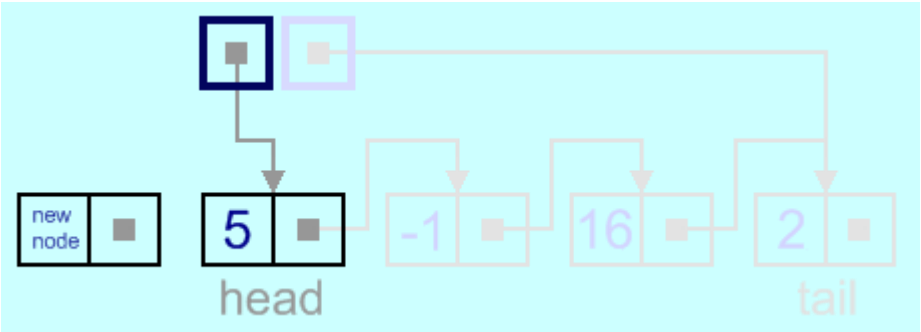
Empty list case

When list is empty, which is indicated by (head == NULL) condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.



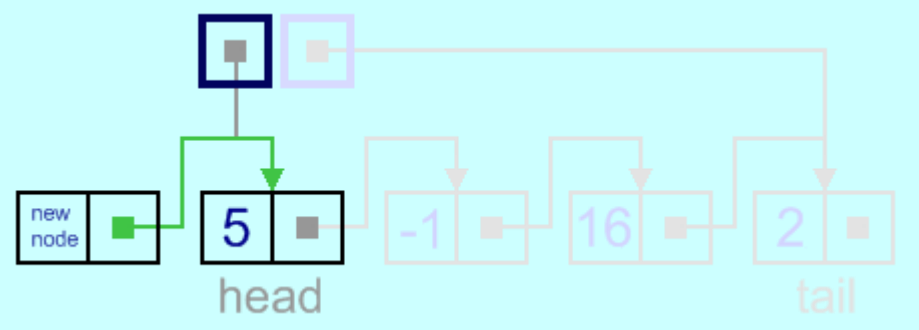
Add first

In this case, new node is inserted right before the current head node.

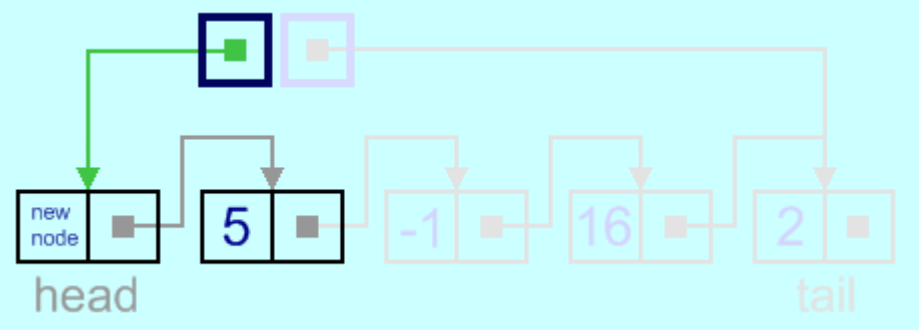


It can be done in two steps:

1. Update the next link of a new node, to point to the current head node.

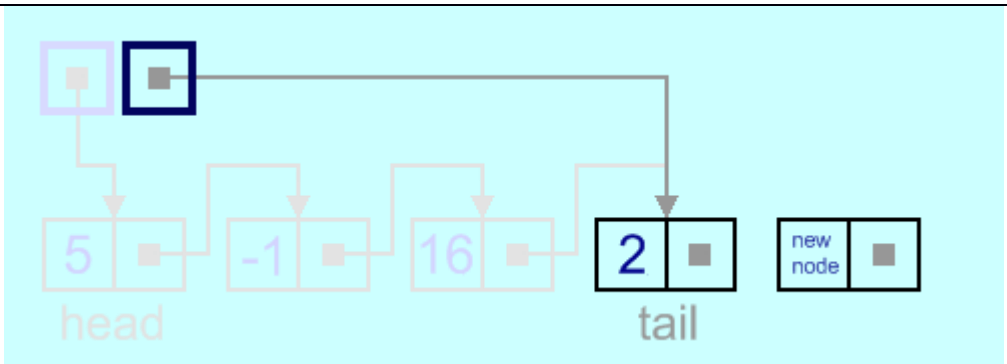


2. Update head link to point to the new node.



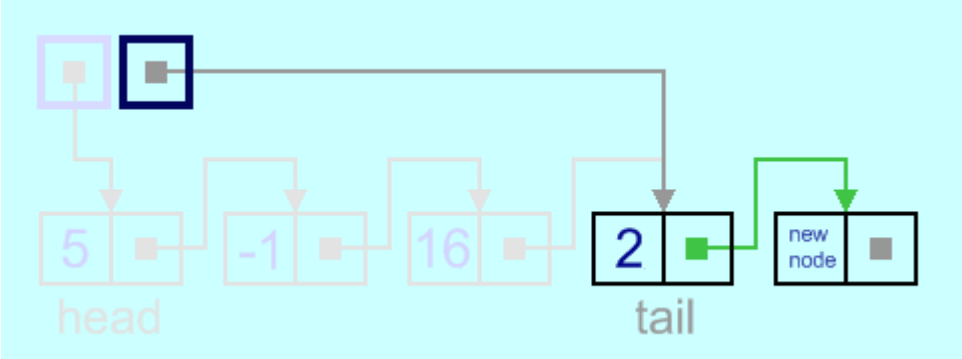
Add last

In this case, new node is inserted right after the current tail node.

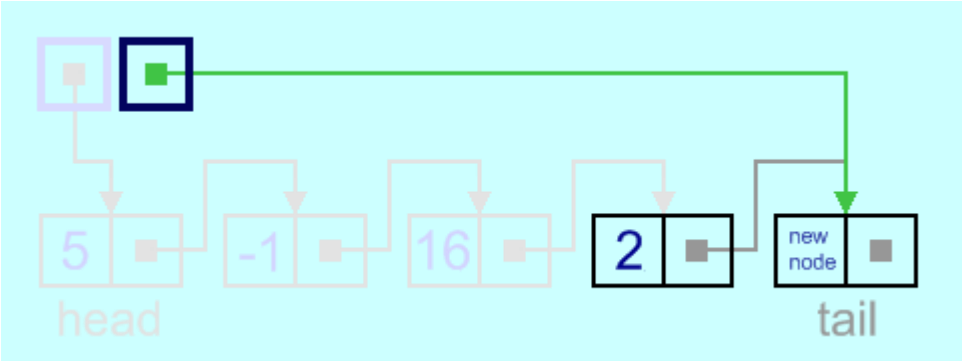


It can be done in two steps:

1. Update the next link of the current tail node, to point to the new node.

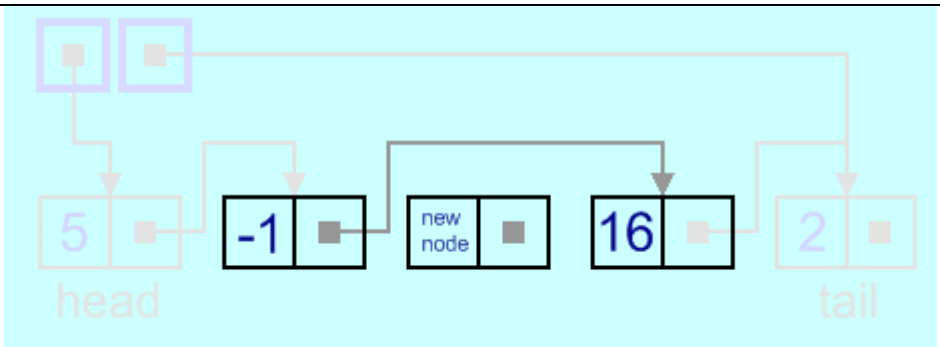


2. Update tail link to point to the new node.



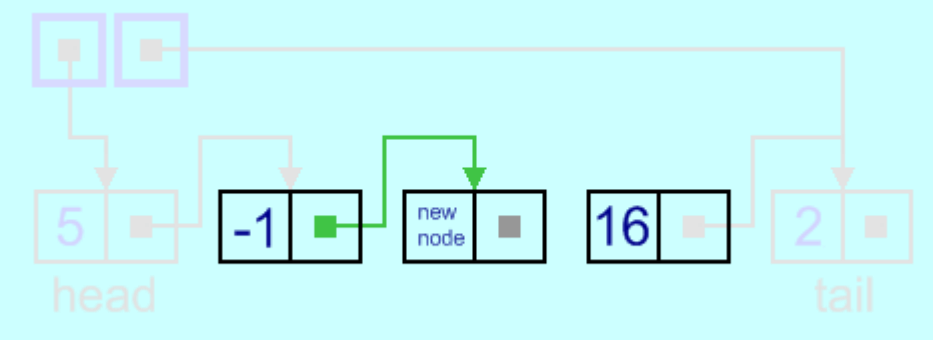
General case

In general case, new node is **always inserted between** two nodes, which are already in the list. Head and tail links are not updated in this case.

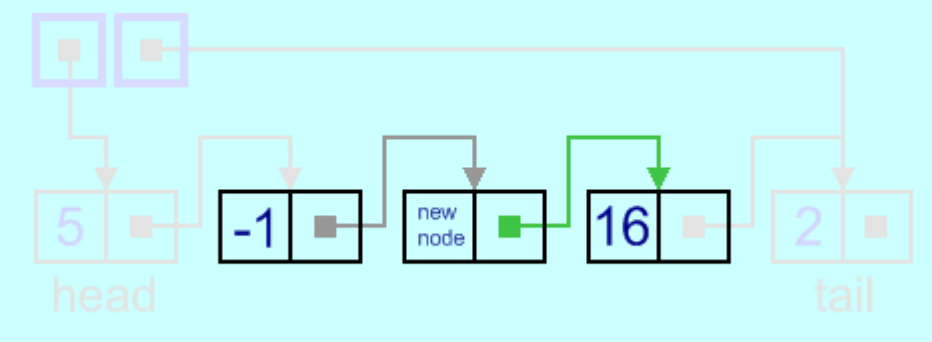


Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the new node.



2. Update link of the new node, to point to the "next" node.

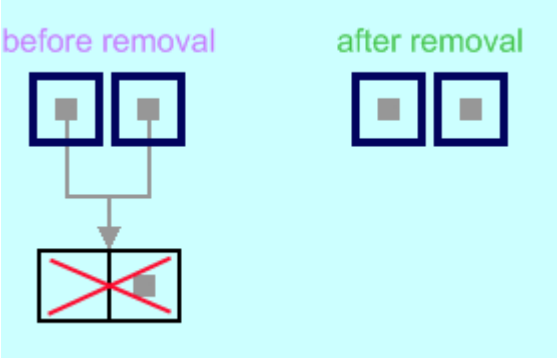


Singly-linked list Deletion operation.

There are four cases, which can occur while removing the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite. Notice, that removal algorithm includes the disposal of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

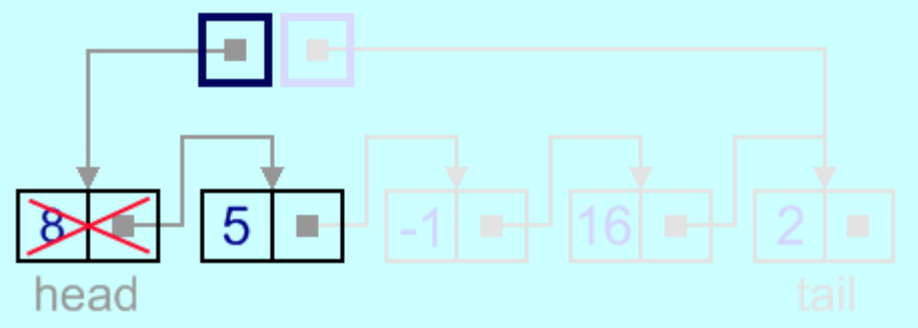
List has only one node

When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to *NULL*.



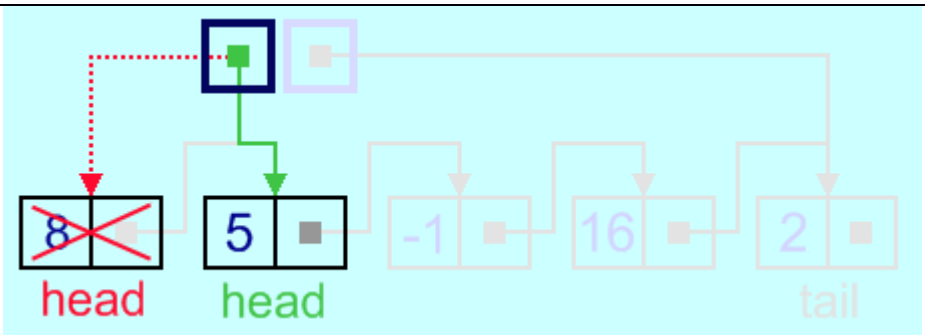
Remove first

In this case, first node (current head node) is removed from the list.

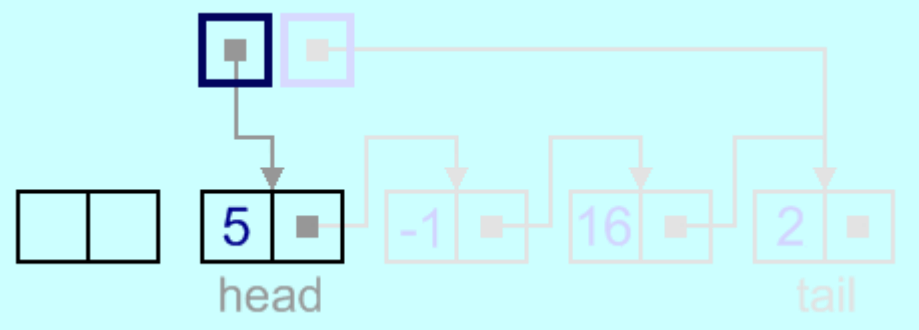


It can be done in two steps:

1. Update head link to point to the node, next to the head.

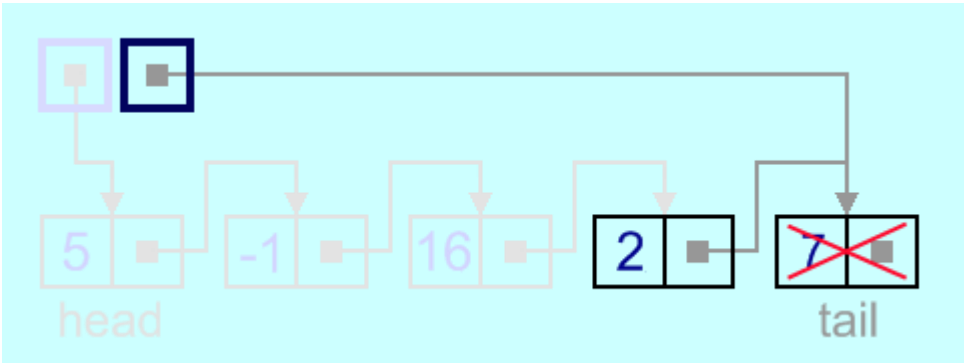


2. Dispose removed node.



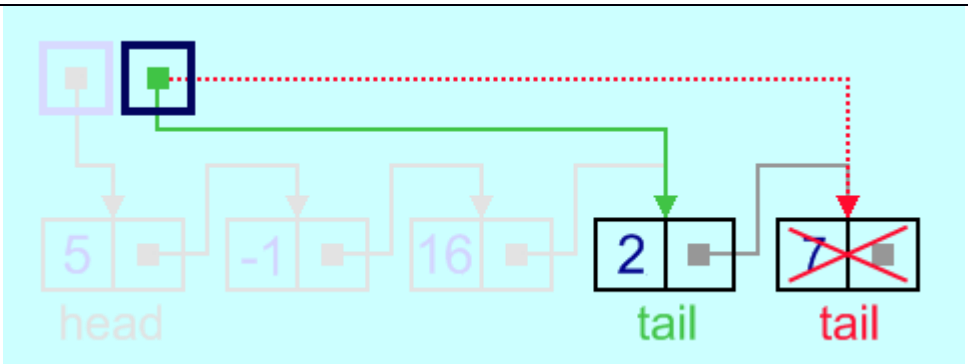
Remove last

In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.

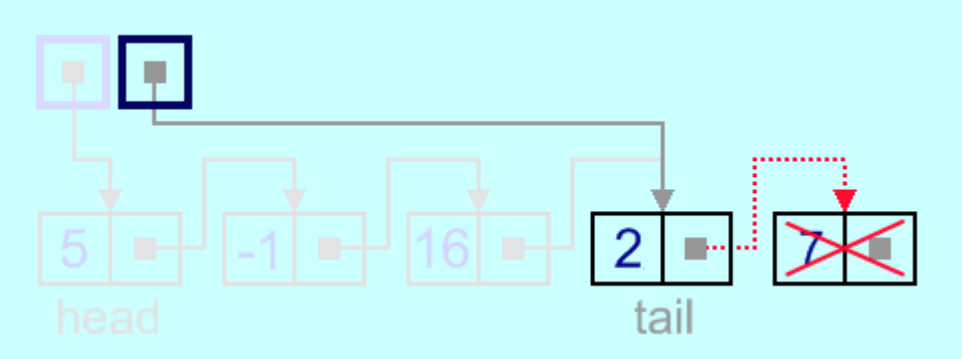


It can be done in three steps:

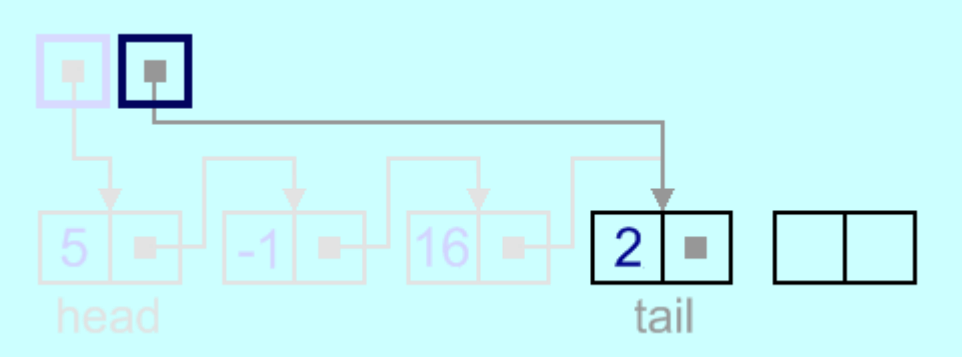
1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



2. Set next link of the new tail to NULL.

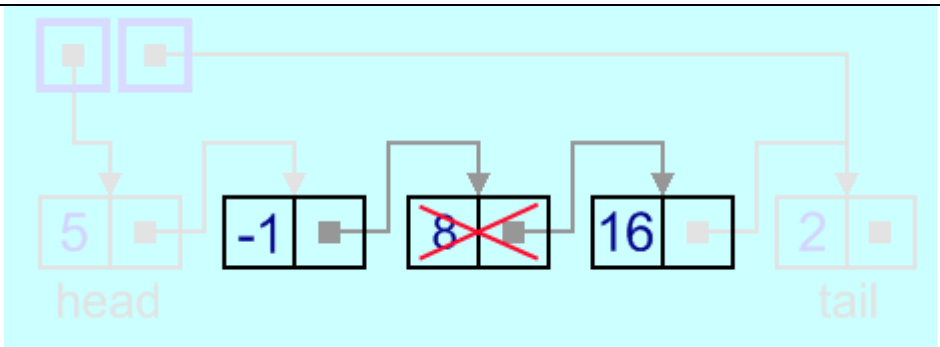


3. Dispose removed node.



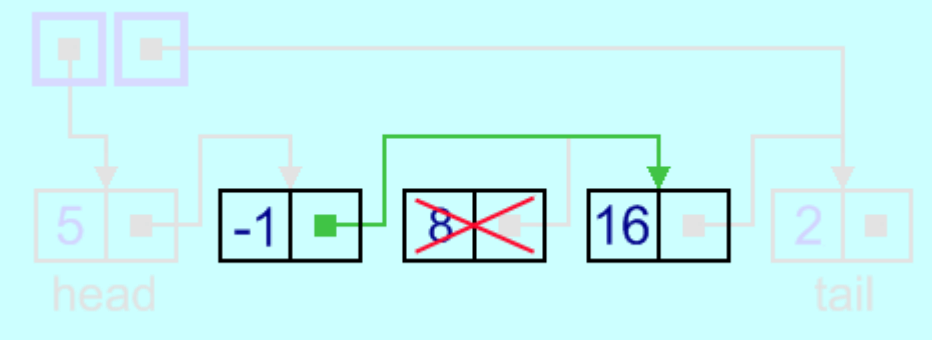
General case

In general case, node to be removed is **always located between** two list nodes. Head and tail links are not updated in this case.

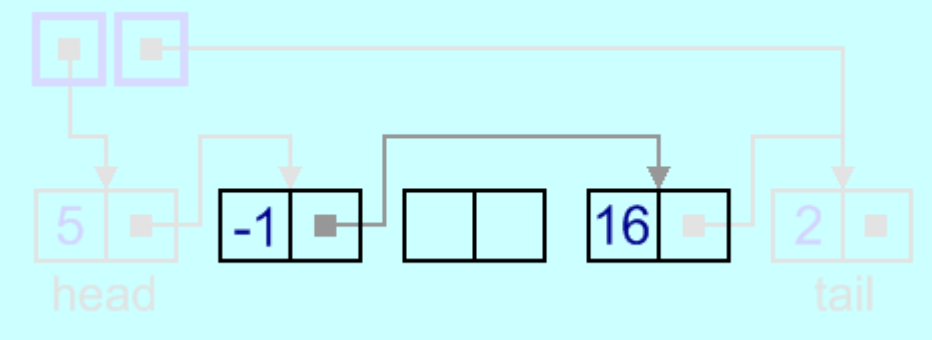


Such a removal can be done in two steps:

1. Update next link of the previous node, to point to the next node, relative to the removed node.



2. Dispose removed node.



Design Analysis/ Implementation Logic:

Single Linked Implementation:

 # class for node

```
class  Node(object):
   def __init__(self,data):
       self.data=data
       self.nextNode=None
```

```
# class for linkedlist:


class Linklist(object):
    def __init__(self):
        self.head=None
        self.size=0

     #function for inserting node at start
    def insert_start(self,data):
        self.size+=1
        newnode=Node(data)
        if not self.head:
            self.head=newnode

        else:
            newnode.nextNode=self.head
            self.head=newnode



#function for inserting node at end

    def insert_end(self,data):
        self.size+=1
        newnode=Node(data)
        temp=self.head
        while temp.nextNode  is not None:
            temp=temp.nextNode
        temp.nextNode=newnode




#function for inserting node at middle

    def insert_middle(self,data):
        self.size+=1
        newnode=Node(data)
        print("Enter after which node u want insert your data")
        x=int(input())
        temp=self.head
        while temp.data !=x :
            temp=temp.nextNode
        newnode.nextNode=temp.nextNode
        temp.nextNode=newnode

 #function for display Linked List


    def display_list(self):
        newnode=self.head
```

```
        if self.head is None:
           print("List is empty")
        else:
              print(" Linked List is :")
              while newnode is not None:
                print(newnode.data)
                newnode=newnode.nextNode
```

#function for delete node at start:

```
    def delete_start(self):
        self.size -=1
        if not self.head:
          print("List is empty")
        else:
           temp=self.head
           self.head=temp.nextNode
           print("Data is deleted",temp.data)
```

#function for delete node at end:

```
    def delete_end(self):
        self.size -=1
        if not self.head:
          print("List is empty")
        else:
           newnode=self.head
           if newnode.nextNode==None:
              self.head=None
           else:

              while newnode.nextNode is not None:
                    temp1=newnode
                    newnode=newnode.nextNode

              temp1.nextNode=None
           print("Data is deleted",newnode.data)
```

#function for delete a specific node.

```
    def delete_data(self,data):
        flag=False
        newnode=self.head
        if self.head is None:
           print("List is empty")
        elif newnode.data==data:
           self.head=newnode.nextNode
           print("Data is deleted",newnode.data)

        else:
```

```
        newnode=self.head
        while newnode.data!=data and newnode!=None:
            temp1=newnode
            newnode=newnode.nextNode
            if newnode is None:
                flag=True
                break
        if flag is True:
           print("data is not present")
        else:
           temp1.nextNode=newnode.nextNode
           print("Data is deleted",newnode.data)
```

Conclusion: Students performed analysis on given real time application and identified objects, classes and their properties and methods.

Project- 02

Project- 02

A. Create a queue which will maintain the list of customer at restaurant/ Bank.

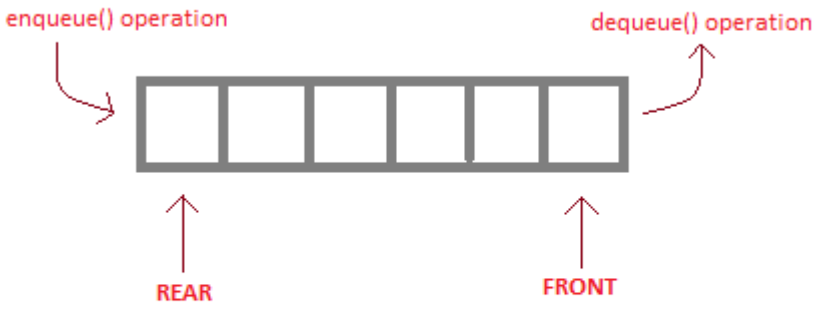
B. Create a Priority Queue for Patients in Hospital.

After completion of this assignment students will able to:

Use queue for creating application and perform different primitive operations like enqueue and dequeue**.**

Relevant Theory / Literature Survey:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is used to return the value of first element without dequeuing

A priority queue is an abstract data type (ADT) supporting the following three operations:

1. Add an element to the queue with an associated priority.
2. Remove the element from the queue that has the highest priority, and
   Return it.

Design Analysis/ Implementation Logic:

Algorithm for Priority Queue:

1. Set the appointments of the patients before the Doctor's visit with different priorities assigned to them depending on the illness and remove his entry from the queue after the service.

2. Also allow on the spot service to the patients depending on the severity of the illness.

```
Class Node():
   def _init_(self, id, name, priority):
       self.id=id
       self.name=name
       self.priority=priority
       self. nextNode=None

class Queue():
    def __init__(self):
       self.front=None
       self.rear=None
```

Algorithm Enqueue operation :

```
def  enque (self,id,name,priority):

1. Create a new node newnode=Node(id,name,priority)

2. if rear is None then
          rear=newnode;
          front=newnode;
 3. else if newnode.priority > (front. priority)  #highest priority element
          newnode.nextNode=self.front
          self.front=newnode
 4.else  temp=self.front
          While temp.nextnode==None &&   newnode.priority<=temp.nextnode. priority
                  Temp=temp.nextNode
            newnode. nextNode= temp.nextNode;
           temp. nextnode=newnode;
```

Algorithm Dequeue operation:

qdelete(self)://function to serve the patient

1. if self.front is None:
   Queue is empty

2. temp=self.front
   front=temp.nextNode

Conclusion: Students implements two different application Queue and priority queue.

Project- 03

Project- 03: Simulate an air traffic controller using suitable data structure

After completion of this assignment students will able to:

Indentify suitable data structure for any given application and using it implement application as per given requirement.

Relevant Theory / Literature Survey:

The project involves the design, implementation, testing, and analysis of a simplified air traffic monitoring and control system (ATC). A standard computer system, such as a PC or workstation, with a graphics display and accessible clock is assumed. The specifications are deliberately vague in some places, for example, in the formatting details of display output and command input. To a large extent, these depend on the particular computer and software employed; we also wish to leave some scope for design creativity.

The purpose of the ATC is to assure the safety of aircraft traversing a given airspace and to assure their efficient traversal. To achieve these goals, the ATC must track or monitor each aircraft in the space and must be able to command an aircraft to change its path or its speed. Safety means avoiding collisions. The safety constraint is defined in terms of minimum separation - at any time, no two aircraft may be closer than a given distance from each other, say 1000 feet in elevation and 3 miles in the horizontal plane. These numbers and units, and the others that are given below, should be treated as examples that could be changed easily. We will not be too concerned with efficiency, for example, trying to maximize aircraft throughput; however, the ATC should not degrade efficient operation except when the safety constraint may be otherwise violated. For the project, this means that unless two aircraft are, or are projected to violate the minimum separation constraint, they will not be commanded to change their direction or speed.

Conclusion: Students simulate an air traffic controller using suitable data structure.

Project- 04

Project 04: **Implement Ticket Checker Application using suitable Data Structure.**

After completion of this assignment students will able to:

Indentify suitable data structure for any given application and implement as per given requirement.

Relevant Theory / Literature Survey:

The ticket checker system is an application, where user can get railway ticket in the form of QR (Quick Response) code. Once the user buys the ticket depending upon the ticket type it will get validated using his mobile. For single ticket type the ticket will be valid for the respective journey time and we will provide extra thirty minutes in case if the train is late. For return ticket type the ticket will be valid till midnight. User's ticket information is stored in suing appropriate data structure. In this application one function is created to scan users QR Code for getting the user ticket id and he will search the ticket id in admin database to get the ticket information.

Application consist of three main module.

1. Admin application
2. User application
3. Ticket checker application

1. Admin Application

This system provides desktop application for an admin. Using this application the admin can make changes in the system. These changes may include:

1. Add new location
2. Add new route
3. Manage route

2. User Application

2.1. Signup and Login

This is the first procedure to know the users information. User need to register before using

this application. During the registration user must fill his personal information like name, address, set a username, password, phone number and email-id for his account. The next time whenever user wishes to buy a ticket, user can simply login into his account using username and password which he has registered. All these information are stored in data structure and can be accessed at any time.

2.2 Buy ticket:

In order to buy the ticket the user should select the source and destination locations. These source and destination will be checked from the database. The user will check schedule of trains according to source and destination. Once the user has selected source and destination he/she can book the ticket. Before buying the ticket the user has to ensure if there is enough balance in his/her account.

2.3 Password validation:

After selecting source and destination by the user it will hit the buy button. The server validates the password of the user, if it is successful the journey details and information regarding user in Database will be saved. Ticket Number and time of buying is generated and balance amount value is displayed.

2.4. Generating QR Code :

Once the ticket number and time of buying the ticket is saved in server database, the transaction id is send to server database to generate QR code. Then generated QR code is send to the user mobile and saved within the device memory. This QR code contains the transaction ID, status of ticket, source and destination and amount.

2.5. Ticket Validation:

Once the user buys the ticket depending upon the journey time it will get validated. Once the journey time gets finished the ticket will get deleted automatically.

2.6. View History:

The user can view all the previous transaction. User can also delete them if user wishes to. This information will be deleted from his phone only and not from the server database. The user can also edit his personal information.

3. Ticket Checker Application

Using the ticket checker application, the checker can do the following tasks

3.1. Checking QR Code:

The ticket checker will have QR code scanner which will scan the QR code with the checker application to obtain user transaction ID, status of ticket, source and destination and amount.

3.2. Checking with Database:

This is a backup arrangement just in case if the ticket checker is not able to scan QR Code if the user's mobile display is being damage, battery failure etc. In this case the ticket checker will directly verify with admin database by making use of the username to get detail information about the ticket for validation purpose. Checker will enter the ticket id in server database to get information about the ticket, in order to verify the journey details especially time and date of the ticket.

Conclusion: Students performed analysis on given real time application and implement using suitable data structure.