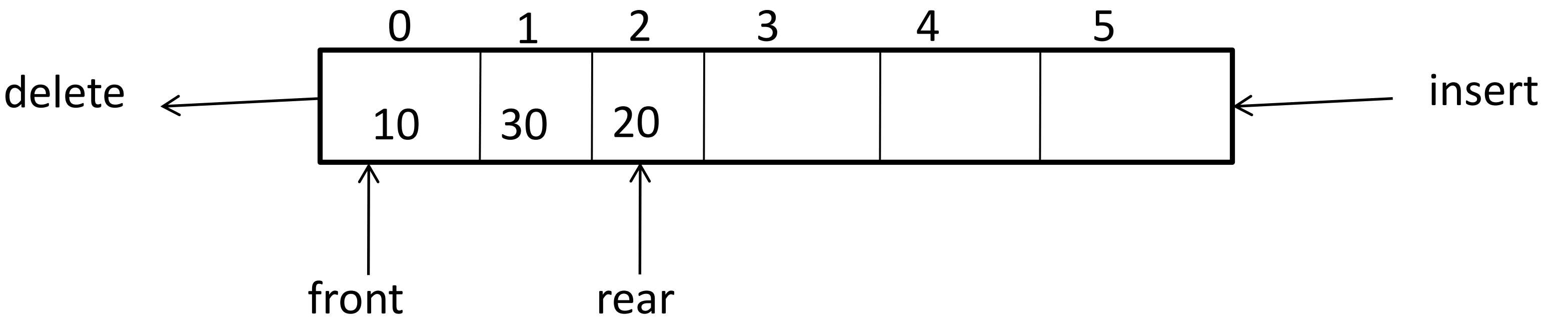
Queues :

- Queue is a first in first out data structure.
- Queue concept is used in printers, Operating systems for process scheduling.
- Queue concept is used extensively in shared systems.
- Queuing a phone call when the line is busy.
- Message processing in client server applications in computer networks.

8. Queues and lists

What is a queue?

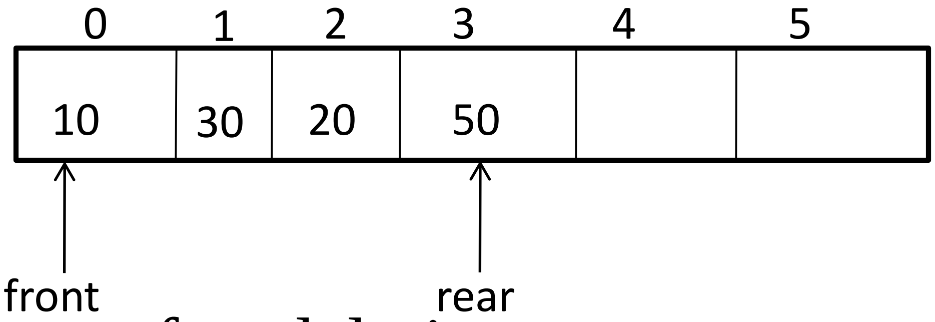
- Is a special type of data structure where elements are inserted from one end and deleted from the other end.
- The end from elements are inserted is called rear end.
- The end from where elements are deleted is called front end.



This is a queue of size 6 with currently 3 elements.

Queue is called First In First Out(FIFO) data structure.

Queue after inserting 50

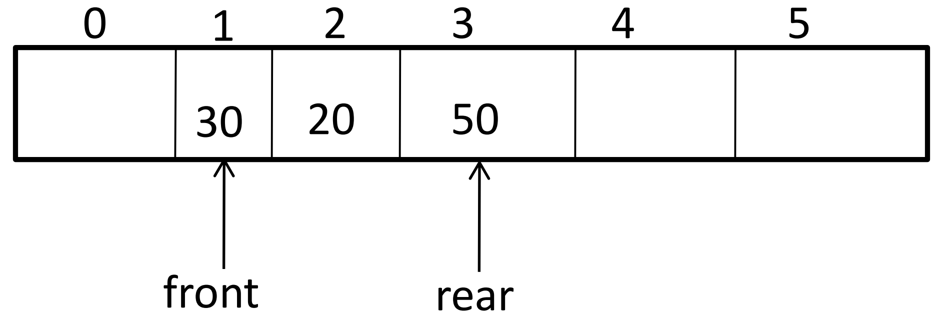


0	1	2	3	4	5
10	30	20	50		

front

rear

Queue after deletion



0	1	2	3	4	5
	30	20	50		

front

rear

After every insertion, rear moves 1 position ahead and after every deletion, front moves 1 position ahead.

- Hence array can be used to hold the elements of queue, a rear variable to indicate rear position and a front variable to indicate the front position.

```
struct queue
{
        int front, rear;
        int items[MAX_SIZE];
};
```

Operations performed on queues:

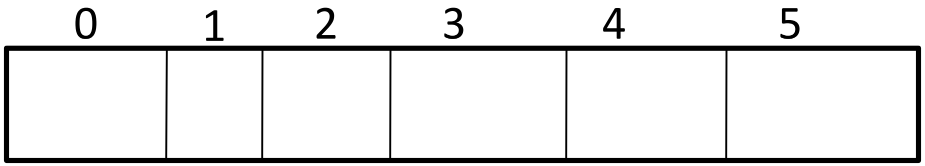
1. Insert an item
2. Delete an item
3. Display the queue

Different types of queues:

1. Ordinary queue(queue)
2. Circular queue
3. Double ended queue
4. Priority queue

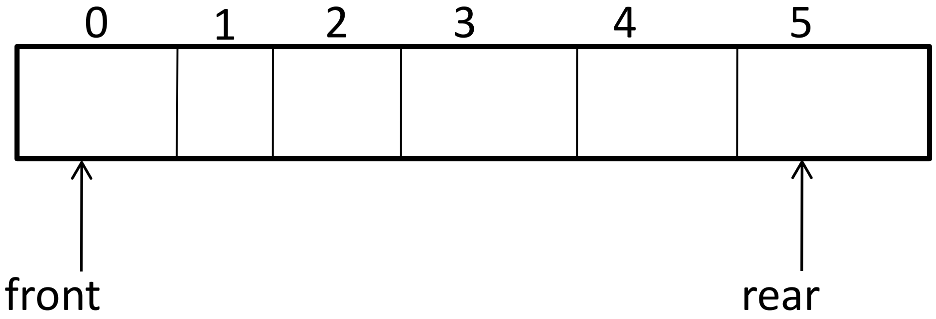
Queue(ordinary queue):

- Queue already discussed.
- To implement the queue, rear is set to -1 and front to zero initially.
- At any point, queue is empty if rear is -1 or front>rear.
- At any point, queue is full if rear is MAX_SIZE-1.



front=0 ,rear=-1

- when element is inserted, rear is incremented and element is placed.(rear=rear+1)
- when element is deleted, front is incremented.(front=front+1)

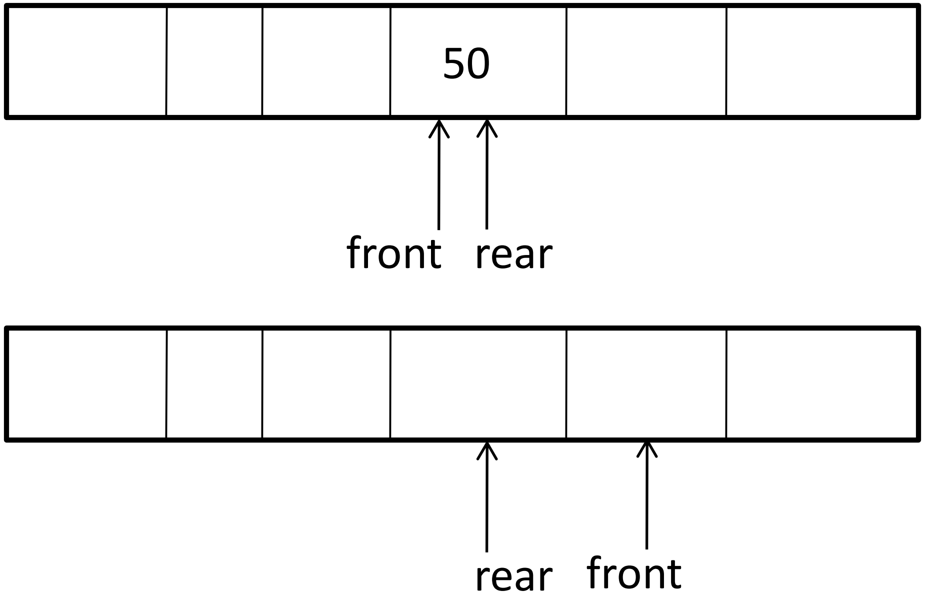


- Here rear is MAX_SIZE-1. Now if we try to insert an element, it is not possible because queue is full.

```
/*c program to implement ordinary queue*/
#include<stdio.h>
#define MAX_SIZE 6
struct queue
{
    int front, rear;
    int items[MAX_SIZE];
};
typedef  struct queue QUEUE;
Void main( )
{
    QUEUE q;
    q.front=0;
    q.rear=-1;
    int choice;
    int item;
for(; ;)
{
    printf(“1.insert 2.delete 3.display 4.exit”);
    scanf(“%d”,&choice);
    switch(choice)
    {
        case 1:printf(“enter the element to be inserted”);
               scanf(“%d”,&item);
               insert(item, &q);
               break;
        case 2:delete(&q);
               break;
        case 3:display(&q);
               break;
        case 4:exit(0);
    }
}/*end for*/
}/*end main*/
```

```
Void insert(int element, QUEUE *qptr)
{
    if(qptr→rear==MAX_SIZE-1)
    {
            printf(“queue full”);
            return;
    }
    qptr→rear=qptr→rear+1;
    qptr→items[qptr→rear]=element;
}
Void display(QUEUE *qptr)
{
    int i,j,k;
    i=q→front;          j=q→rear;
    printf(“contents are”);
    for(k=i;k<=j;k++)
    printf(“%d”, qptr→items[i]);
}
```

```
Void delete(QUEUE *qptr)
{
   if(qptr→front>qptr→rear)
   {
        printf(“queue empty”);
        return;
   }
   printf(“element deleted is %d”, qptr→items[qptr→front]);
   qptr→front=qptr→front+1;
   if(qptr→front> qptr→rear)     /* reset front and rear if queue
   {                                gets empty after deletion*/

        qptr→front=0;
        qptr→rear=-1;
   }
}
```



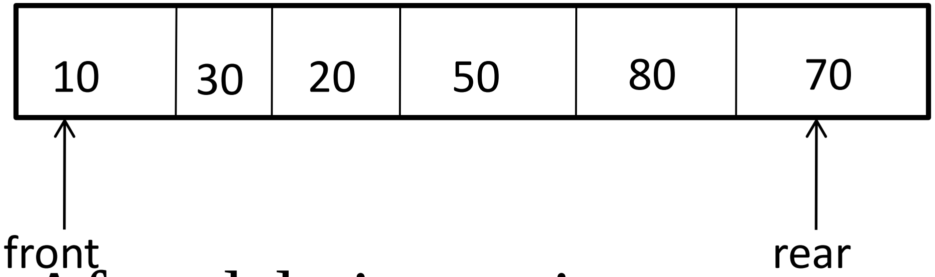
50

front rear

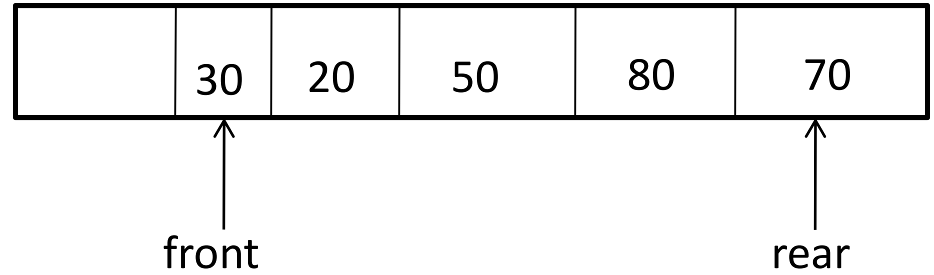
rear front

- Here after deleting the last item, front goes beyond the rear as shown above.
- This is nothing but queue is empty and hence front and rear are set to their initial values.

- Consider a full queue



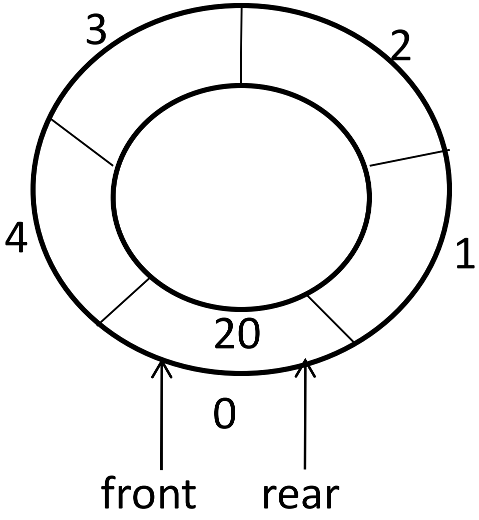
- After deleting an item

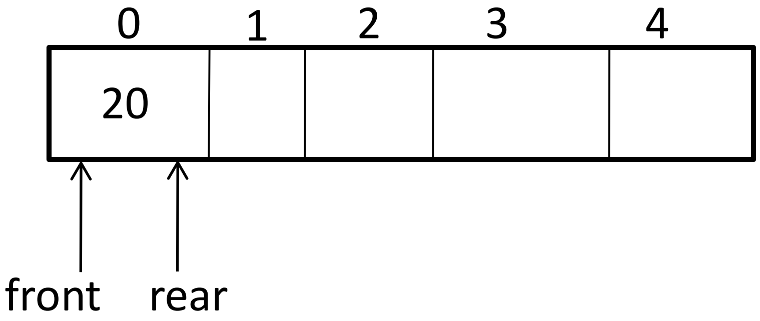


- Now if we try to insert an item, it gives queue full because rear is MAX_SIZE-1.
- Actually queue is not full.
- Hence we cannot insert elements even though there are vacant positions.

Circular queue:

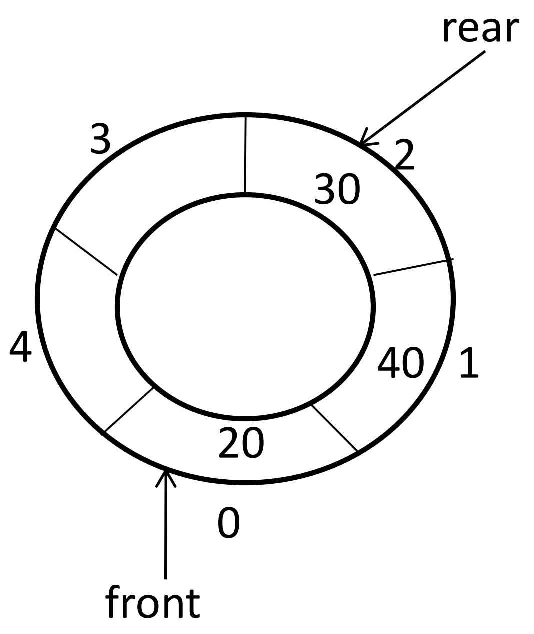
- Problem associated with the ordinary queue can be overcome using circular queue.
- The elements can be stored efficiently in an array so as to wrap around so that the end of queue is followed by front of queue.

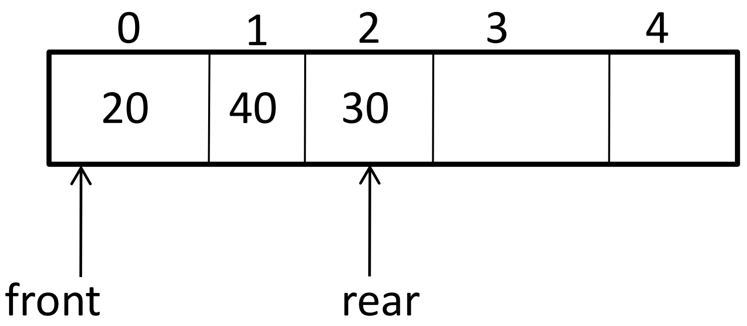




Ordinary queue

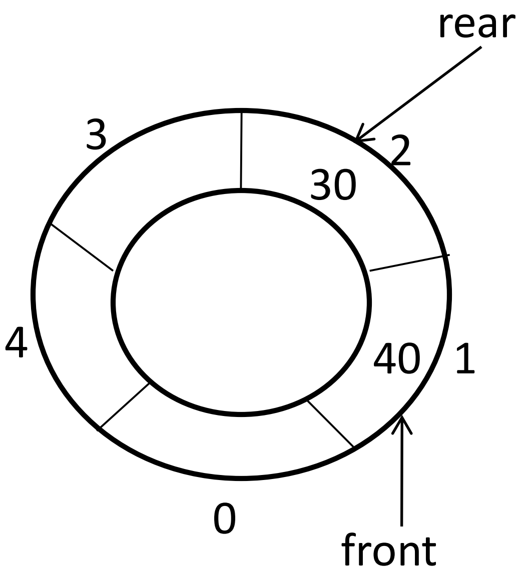
After inserting two more items:

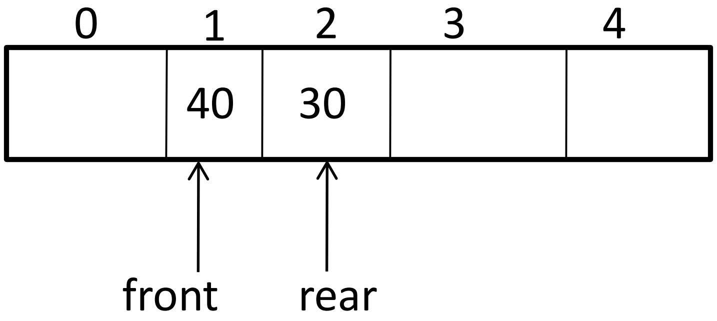




Ordinary queue

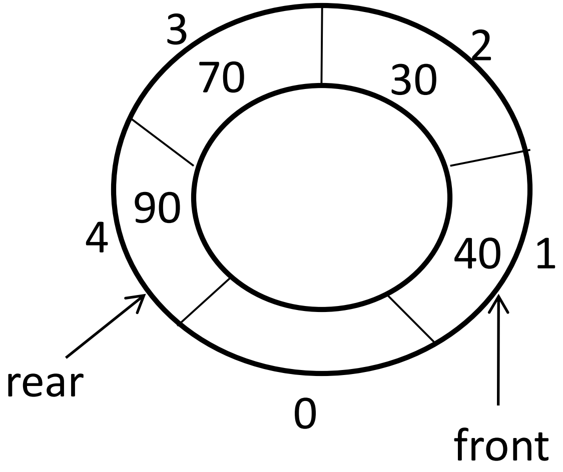
After deleting an item:

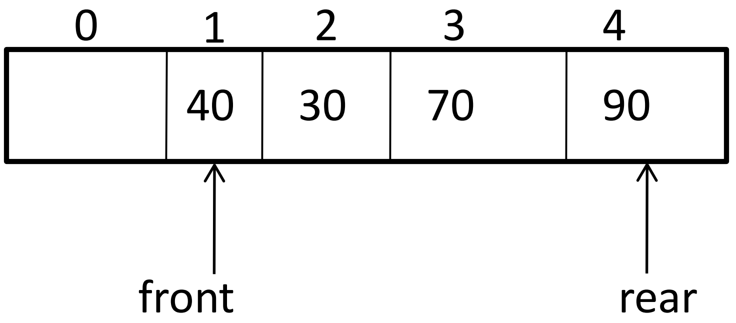




Ordinary queue

After inserting two more items:

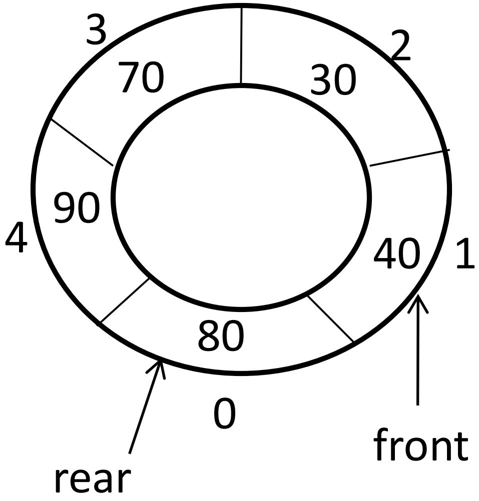


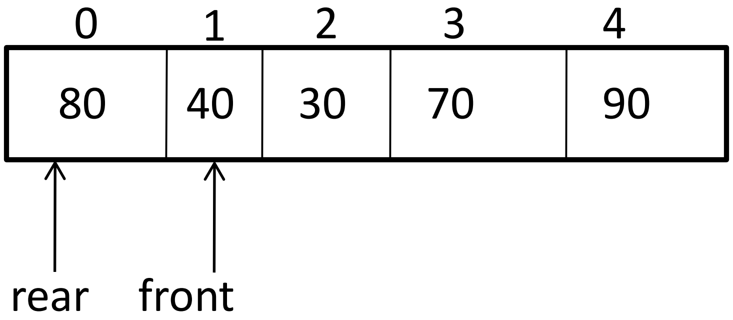


Ordinary queue

At this point ordinary queue gives queue full but in circular queue there is one position vacant.

After inserting one more item:





0	1	2	3	4
80	40	30	70	90

Ordinary queue

- Element is inserted in zero position since it was vacant.

Implementing circular queue:

- A count variable is used. It is incremented after every insertion and decremented after every deletion.
- Value of count gives the total number elements in circular queue.
- Hence at any point, if count==0, queue is empty and if count==MAX_SIZE-1, queue is full.
- rear is incremented using rear=(rear+1)%MAX_SIZE;

For ex: if rear==4, MAX_SIZE is 5 and count!=MAX_SIZE

rear+1%MAXSIZE→(4+1)%5→0. hence element is stored in 0th location.

Now if rear==2, (2+1)%MAX_SIZE→3. hence element is inserted in 3rd position.

- front is incremented in the same way front=(front+1)+MAX_SIZE;
- Circular queue implementation is similar to ordinary queue. The only difference is using a counter variable and incrementing front and rear as shown above.
- To check for queue full, check if count==MAX_SIZE;
- To check for queue empty, check if count==0;

```
/*c program to implement circular queue*/
#include<stdio.h>
#define MAX_SIZE 5
struct queue
{
    int front, rear;
    int items[MAX_SIZE];
    int count;
};
typedef  struct queue QUEUE;
Void main( )
{
    QUEUE q;
    q.front=0;
    q.rear=-1;
    q.count=0;
    int choice;
    int item;
for(; ;)
{
    printf(“1.insert 2.delete 3.display 4.exit”);
    scanf(“%d”,&choice);
    switch(choice)
    {
        case 1:printf(“enter the element to be inserted”);
               scanf(“%d”,&item);
               insert(item, &q);
               break;
        case 2:delete(&q);
               break;
        case 3:display(&q);
               break;
        case 4:exit(0);
    }
}/*end for*/
}/*end main*/
```

```
Void insert(int element, QUEUE *qptr)
{
    if(qptr→count==MAX_SIZE)
    {
          printf(“queue full”);
          return;
    }
    qptr→rear=(qptr→rear+1)%MAX_SIZE;
    qptr→items[qptr→rear]=element;
    (qptr→count)++;
}
```

```
Void display(QUEUE *qptr)
{
    int i,k;
    if(q→count==0)
    {
            printf(“queue is empty”);
            return;
    }
    i=q→front;
    printf(“contents are”);
    for(k=1;k<=qptr→count;k++)
    {
            printf(“%d”, qptr→items[i]);
            i=(i+1)%MAX_SIZE;
    }
}
```

```
Void delete(QUEUE *qptr)
{
    if(qptr→count==0)
    {
        printf(“queue empty”);
        return;
    }
    printf(“element deleted is %d”, qptr→items[qptr→front);
    qptr→front=(qptr→front+1)%MAX_SIZE;
    (qptr→count)- -;
}
```

Double ended queue:(deque)

- Special type of data structure in which insertions and deletions will be done either at front end or at the rear end.

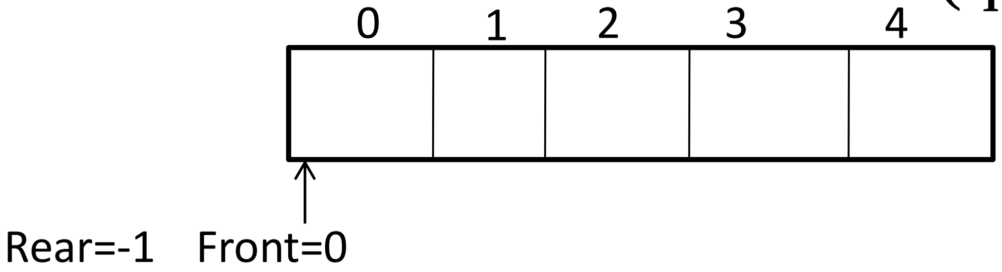
The operations performed on deque

1. Insert from front
2. Insert from rear
3. Delete from front
4. Delete from rear
5. Display

Insert at front end

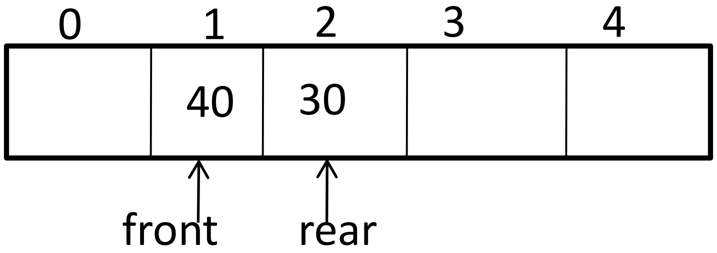
It can be done in the following 2 cases

1. When front==0 and rear==-1(queue is empty)



Here rear is incremented and element is inserted.

2. When front!=0



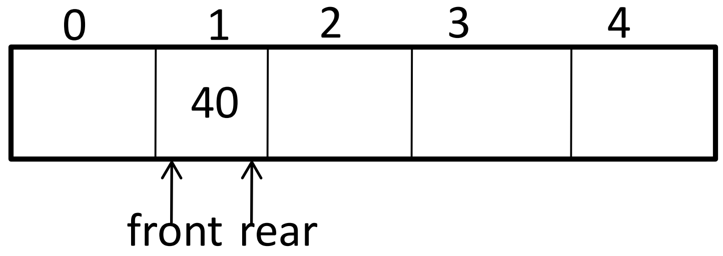
Here front is decremented and element is inserted.

- Insertion is not possible at front end if above 2 cases does not exist.

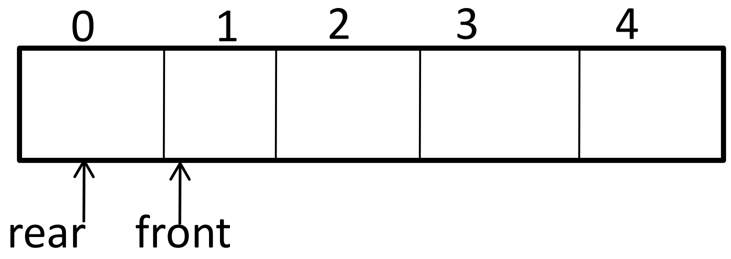
Deletion from rear end:

- Decrement the rear after every deletion.
- If front goes beyond rear, then reset front and rear to 0 and -1 respectively.
- At any point, queue is empty if front > rear.

Before last item is deleted



After last item is deleted



```
/*c program to implement dequeue*/
#include<stdio.h>
#define MAX_SIZE 5
struct queue
{
    int front, rear;
    int items[MAX_SIZE];
};
typedef  struct queue QUEUE;
Void main( )
{
    QUEUE q;
    q.front=0;
    q.rear=-1;
    int choice;
    int item;
```

```
for(; ;)
{
   printf(“1.insert front 2.insert rear 3.delete front 4.delete
   rear 5.display 6.exit”);
   scanf(“%d”,&choice);
   switch(choice)
   {
        case 1:printf(“enter the element to be inserted”);
               scanf(“%d”,&item);
               insert_front(item, &q);
               break;
        case 2:printf(“enter the element to be inserted”);
               scanf(“%d”,&item);
               insert_rear(item, &q);
               break;
        case 3:delete_front(&q);
               break;
case 4:delete_rear(&q);
          break;
case 5:display(&q);
          break;
case 6:exit(0);
}
}/*end for*/
}/*end main*/

Void insert_front(int element, QUEUE *qptr)
{
   if(qptr→front==0 && qptr→rear==-1)
         qptr→items[++(qptr→rear)]=element;
   else if(qptr→front!=0)
         qptr→items[--(qptr→front)]=element;
   else
         printf("insertion not possible");
}
```

```
Void insert_rear(int item, QUEUE *qptr)
{
    if(qptr→rear==MAX_SIZE-1)
    {
          printf(“queue is full”);
          return;
    }
    qptr→items[++(qptr→rear)]=item;
}
```

```
Void delete_front(QUEUE *qptr)
{
    if(qptr→front>qptr→rear)
    {
        printf(“queue empty”);
        return;
    }
    printf(“element deleted is %d”, qptr→items[qptr→front);
    qptr→front=qptr→front+1;
    if(qptr→front  > qptr→rear)  /* reset front and rear if queue
    {                               gets empty after deletion*/

        qptr→front=0;
        qptr→rear=-1;
    }
}
```

```
Void delete_rear(QUEUE *qptr)
{
   if(qptr→front>qptr→rear)
   {
        printf(“queue empty”);
        return;
   }
   printf(“element deleted is %d”, qptr→items[qptr→rear];
   qptr→rear=qptr→rear-1;
   if(qptr→front  > qptr→rear)  /* reset front and rear if
                                    queue gets empty after
                                    deletion*/
   {
        qptr→front=0;
        qptr→rear=-1;
   }
}
```

```
Void display(QUEUE *qptr)
{
    int i;
    if(qptr→front > qptr→rear)
    {
        printf(“queue empty”);
        return;
    }
    printf(“contents are”);
    for(i=qptr→front;i<=qptr→rear;i++)
        printf(“%d”, qptr→items[k]);
}
```

Priority queue:

- Special type of DS in which items can be inserted and deleted based on priority.

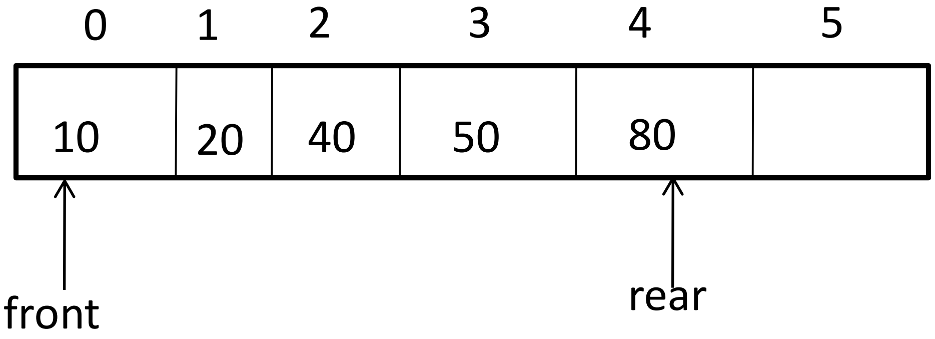
Various types of priority Queues

1.Ascending priority queue

2.Decsending priority queue

- In both the queues insertion can be done in any order but while deleting from ascending queue, the smallest element is removed and from descending queue largest is removed first.

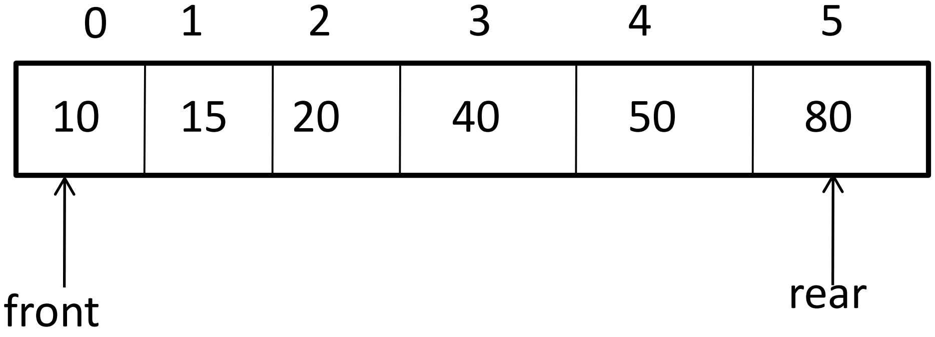
- Hence while inserting elements if the elements are placed such that they are in ascending order and remove the front element, it will be ascending queue.
- Hence few changes have to be made in the insert function to place the elements in ascending order but display and delete remain same.
- In insert function
  - item to be inserted is checked with every item in queue starting from rear until it is lesser than the items in queue.
  - Every time the item is less than the item being checked, the item in queue is moved 1 position right.
  - Finally item is inserted in its position and rear incremented by 1.



0	1	2	3	4	5
10	20	40	50	80	

Now if 15 is to be inserted it has to be placed in position 1.

- 15 is compared with 80 and 80 moved to 5th position.
- 15 is compared with 50 and 50 moved to 4th position.
- 15 is compared with 40 and 40 moved to 3rd position.
- 15 is compared with 20 and 20 moved to 2nd position.

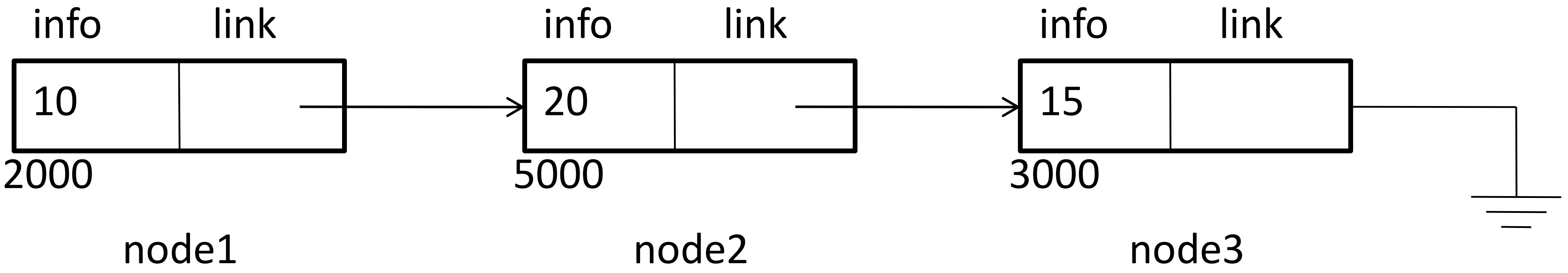


0	1	2	3	4	5
10	15	20	40	50	80

```
Void insert(int element, QUEUE *qptr)
{
    int j;
    if(qptr→rear==MAX_SIZE-1)
    {
        printf(“queue full”);
        return;
    }
    j=qptr→rear;
    while(j>=0 && item < qptr→items[j])
    {
        qptr→items[j+1]=qptr→items[j];
        j--;
    }
    qptr→items[j+1]=item;
    qptr→rear++;                    /*increment rear*/
}
```

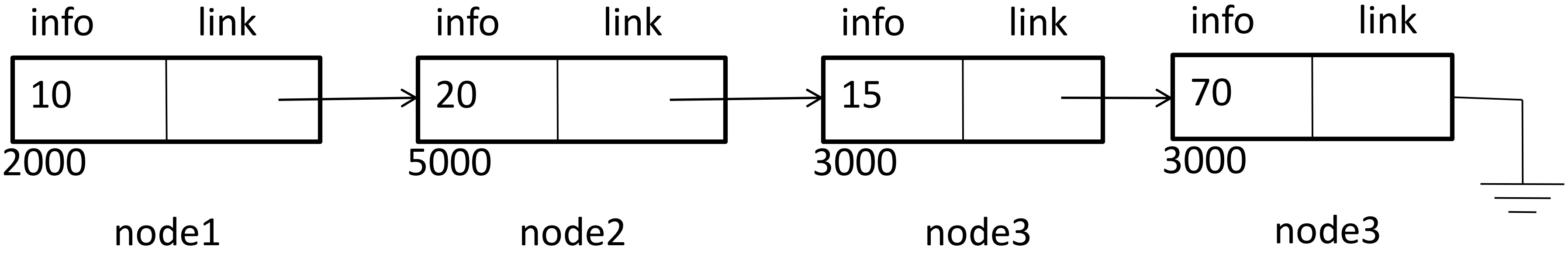
Linked lists:

- In previous sections, stacks and queues are implemented using arrays and the size of array was fixed.
- In linked list concept, memory is allocated dynamically using malloc().
- A linked list is a data structure which is a collection of zero or more nodes, where each node has some information and between two nodes in the list there is a link.
- In short, linked list consists of many nodes, where each node has a pointer to next node.



- The above diagram is a linked list having 4 nodes.
- Each node has 2 fields
  - Info field to hold the item.
  - A link field which points to the next node. This link field is pointer.
- These nodes are stored in non-contiguous memory locations.
- Link field of a node is pointer which contains address of the next node.
- Link field of last node points to NULL.
- A node can be added to and deleted from the front or rear of the list.

After adding node with info 70 at the end of list:



- In linked list a node is added only when required unlike in the array where memory is allocated in the beginning.

How nodes are represented?

- Nodes can be represented using a structure having 2 fields.
  1. Item of the node
  2. Link to next node.

```
struct node
{
        int item;
        struct node *link;
};
```

- Here item is the info of the node and link is the pointer which is required to point to the next node.
- Since link points to next node which is of struct node type, it is declared as struct node *link;

Advantages of linked lists:

1. Size is not fixed.
2. Data can be stored in non contiguous blocks of memory.
3. Insertion and deletion of nodes is easier.

Disadvantages of linked lists:

1. Needs extra space because each node has an extra link field.
2. Accessing a particular node in the list may take more time.

Different types of linked lists are:

1. Singly linked list.
2. Circular singly linked list.
3. Doubly linked list.
4. Circular doubly linked list.

Singly linked lists:

- Is a linked list, where each node has an item and a link field which points to the next node.

Operations performed on singly linked list:

1. Inserting a node into the list.
2. Deleting a node from the list.
3. Display the contents of the list.

Implementing the linked list:

```
struct node
{
        int item;
        struct node *link;
};
typedef struct node *NODEPTR;
```

- Initially a node say, first is declared and assigned a NULL value, which indicates the list is empty.

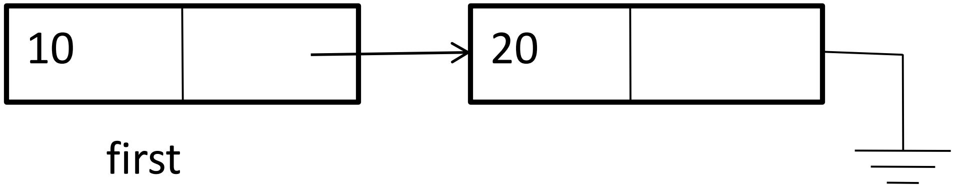
  i.e NODEPTR first=NULL;

- Whenever a new node say temp is created, first is checked. If it is NULL, then temp is made as the first node

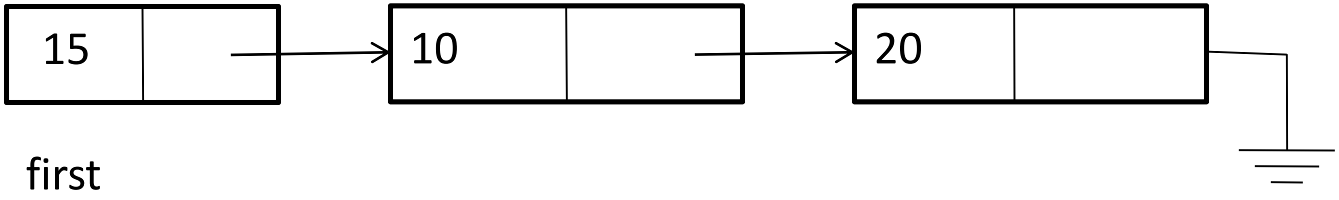
  i.e first=temp;

- If first is not equal to NULL and if the new node is to be inserted at the front of the list, make the link of new node point to the first node as shown

List before inserting



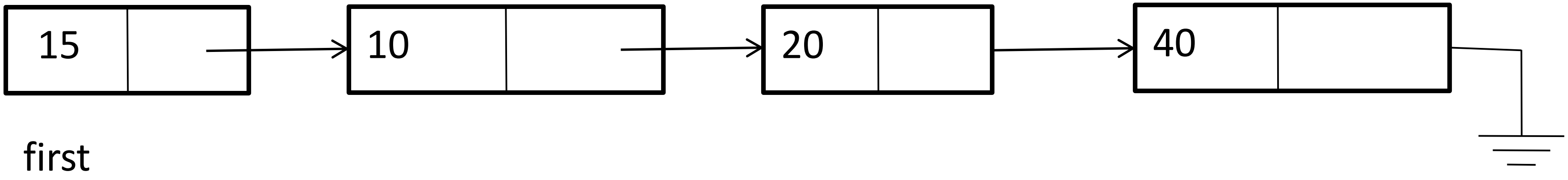
List after inserting at front



- After inserting new node is made the first node.

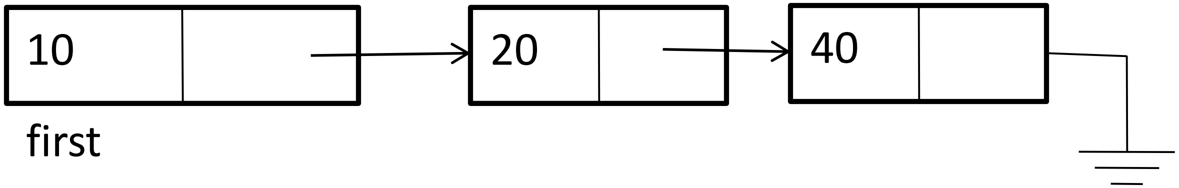
- If the node is inserted at the rear, then link field of the last node is made to point to the new node and link field of new node is pointed to NULL;

List after inserting node at the end:



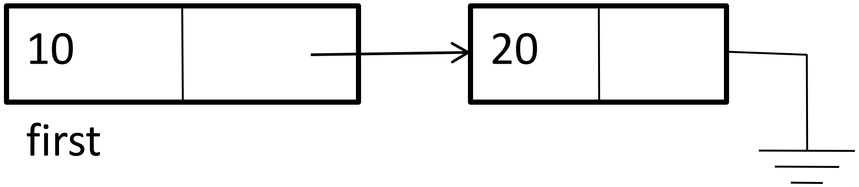
- To delete a node from front of the list, we have to make the second node as the first node and free first node.

List after deleting node at front:



- To delete a node at rear, the last but one node is made to point to NULL and last node is freed.

List after deleting node at rear:



Creating a node

- To create a node, a separate function getnode( ) is written. This function allocates memory for the node using malloc( ) and returns the node.

```
NODEPTR getnode( )
{
    NODEPTR temp;
    temp=(NODEPTR) malloc(sizeof(struct node));
    return temp;
};
```

- malloc( ) allocates the memory of size ‘node’.
- malloc returns void pointer but our pointer is of type NODEPTR and hence it is typecasted.

Freeing the memory:

- Memory allocated must be freed at the end of all the operations.
- It is done by using the function free( ).

free(temp);-- will free the memory allocated for temp.

```
/* C program to implement singly linked list*/
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
    int info;
    struct node *link;
};
typedef struct node * NODEPTR;
Void main( )
{
    NODEPTR first=NULL;
    int choice,item;
    for(; ;)
    {
          printf(“1.insert 2. insert rear 3. delete front 4. delete
          rear 5.display 6. exit”);
          printf(“enter the choice”);

          scanf(“%d”,&choice);
switch(choice)
{
    case 1:printf(“enter the item to be inserted”);
           scanf(“%d”,&item);
           first=insert_front(item,first);
           break;
    case 2:printf(“enter the item to be inserted”);
           scanf(“%d”,&item);
           first=insert_rear(item,first);
           break;
    case 3:first=delete_front(first);
           break;
    case 4:first=delete_rear(first);
           break;
    case 5:display(first);
           break;
    default: exit(0);
}}}
```

```
/*function to obtain new node*/
NODEPTR getnode( )
{
     NODEPTR temp;
    temp=(NODEPTR) malloc (sizeof(struct node));
    if(temp==NULL)
    {
        printf(“out of memory”);
        exit(0);
    }
return remp;
}
/*function to free a node*/
Void freenode(NODEPTR x)
{
    free(x);
}
```

```
/*function to insert an item at front */
NODEPTR insert_front(int item, NODEPTR first)
{
    NODEPTR temp;
    temp=getnode( ) ;          /* obtain a new node*/
    temp→info =item;
    /*attach new node to the first of existing list*/
    temp→link=first;
    return temp;
}
```

```
/*function to insert an item at rear end*/
NODEPTR insert_rear(int item, NODEPTR first)
{
    NODEPTR temp,cur;
    temp=getnode( ) ;/* obtain a new node*/
    temp→info =item;
    temp→link=NULL;
    /* if list is empty, return new node as first node*/
    if(first==NULL)
          return temp;
     /* if list is not empty, traverse till last node to add new node
    at end*/
    cur=first;
    while(cur→link !=NULL)
          cur=cur→link;
    cur→link=temp;             /*point last node to new node*/
    return(first);
}
```

```
/* function to delete the front node*/
NODEPTR delete_front(NODEPTR first)
{
    NODEPTR temp;
    if(first==NULL)
    {
        printf(“list is empty, cannot delete”);
        return first;
    }
     /*retain the address of node to be deleted*/
     temp= first;
     /*make second node as first node*/
     first=first→link;
    /*free the previous front node*/
    freenode(temp);
    return(first)                    /* return the new first node*/
}
```

```
/* function to delete rear node*/
NODEPTR delete_rear(NODEPTR first)
{
    NODEPTR cur,prev;
    if(first==NULL)
    {
        printf(“list is empty, cannot delete”);
        return first;
    }
/*if there is only one node, delete it and set first==NULL*/
    if(first→link == NULL)
    {
        freenode(first);
        first=NULL;
        return first;
    }
/*obtain addresses of last node and its previous node*/
prev=NULL;
cur=first;
While(cur→link!=NULL)
{
    prev=cur;
    cur=cur→link;
}
freenode(cur);
/*set the link of present last node to NULL*/
prev→link=NULL;
return first;
}
```

```
/*function to display*/
Void display(NODEPTR first)
{
    NODEPTR temp;
    if(first==NULL)
    {
        printf(“list is empty”);
        return;
    }
    printf(“contents is”);
    temp=first;
    while(temp!=NULL)
    {
        printf(“%d”,temp→info);
        temp=temp→link;
    }
}
```

Linked list as a stack:

- stack can be implemented using linked list by providing only insert_front and delete_front functions.
- Insertion and deletion is done on front end.

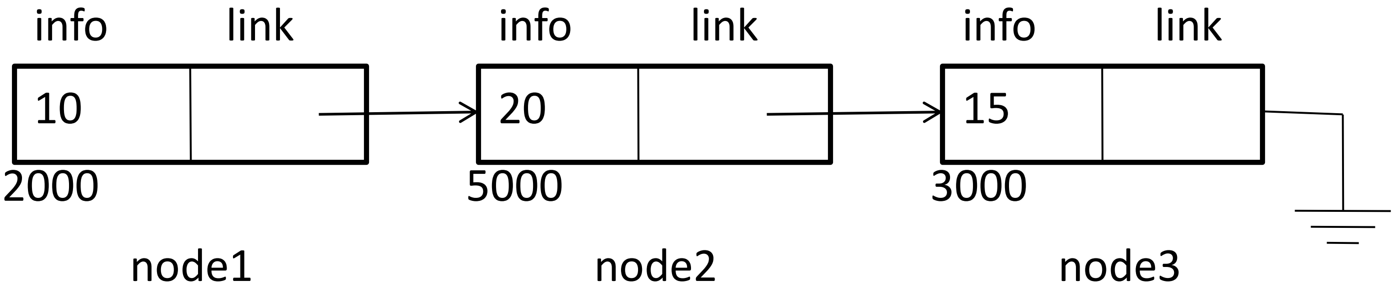
Linked list as a queue:

- queue can be implemented using linked list by providing only insert_rear and delete_front functions.
- Insertion is done on rear and deletion is done on front end.

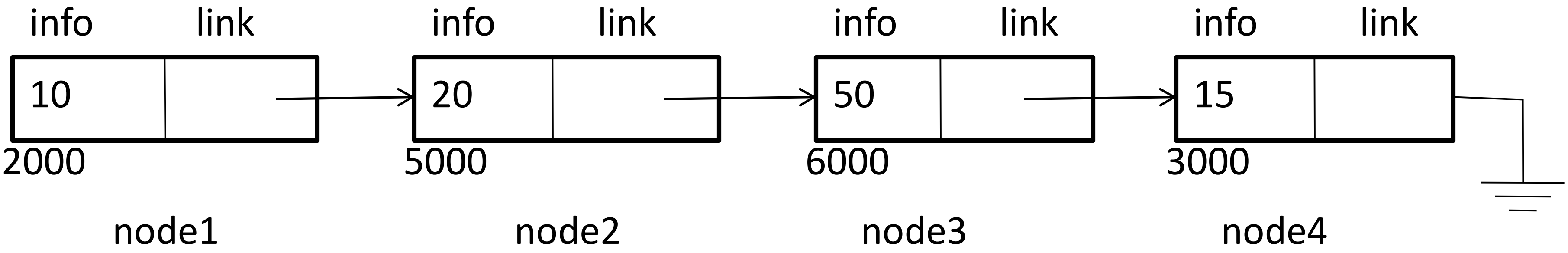
Linked list as a double ended queue:

- Double ended queue can be implemented using linked list by providing insert_rear, insert_front, delete_rear and delete_front functions.

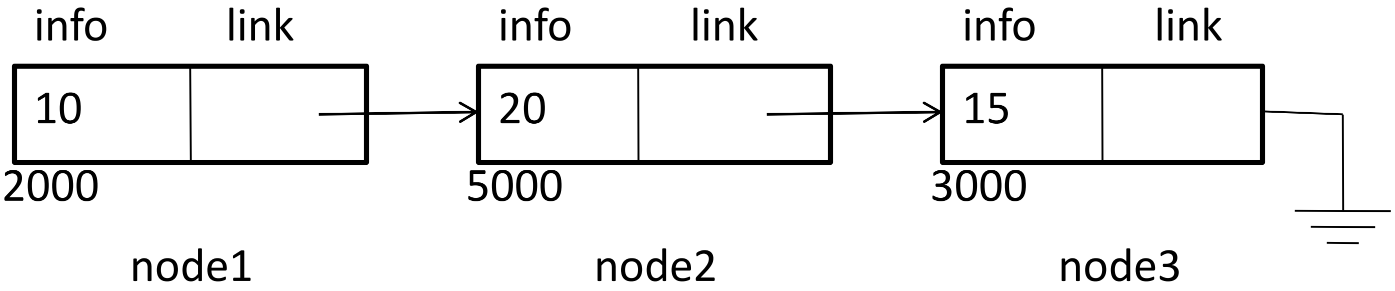
Inserting a node in the middle of the list



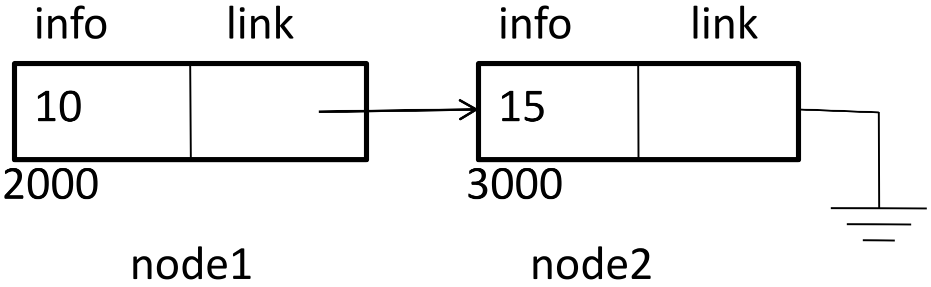
- In the above list, to add node with item 50 after the node with item 20
  - Traverse the list till the node with item 20.
  - set the link of node2 to the new node and link of new node to node3(with item 15).



Deleting a node in the middle of the list

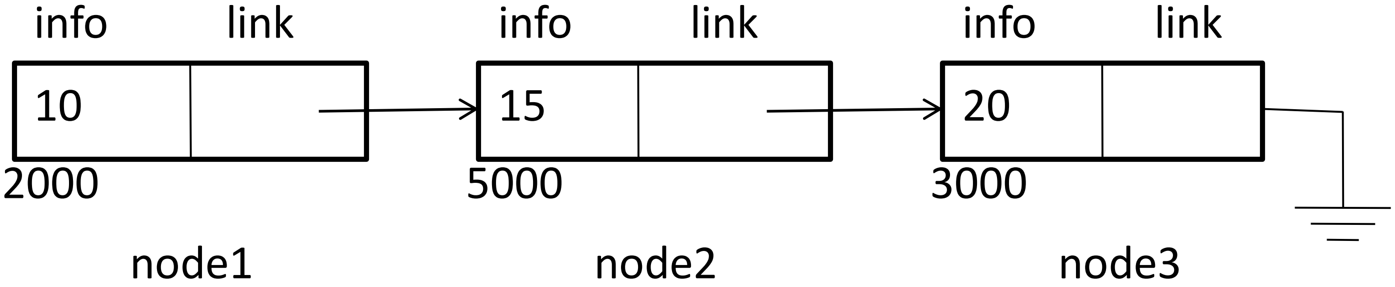


- In the above list, to delete node with item 20
  - Traverse the list till the node with item 20.
  - set the link of node1 to node3(with item 15).
  - Delete node2.

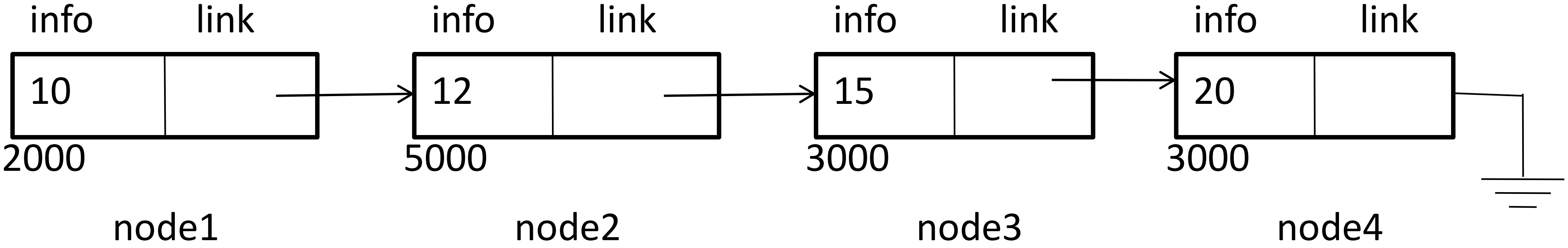


Creating an ordered linked list:

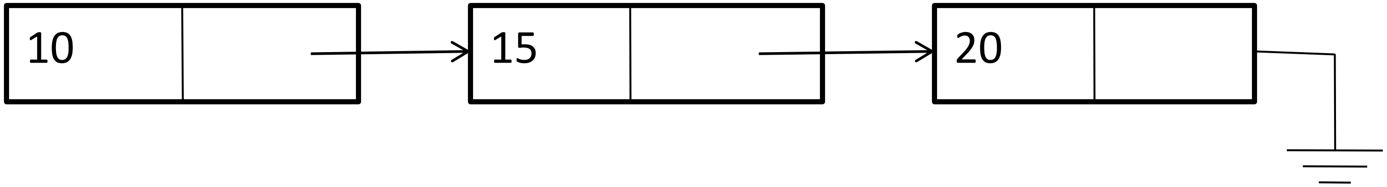
- Is a list in which items are stored in either ascending or descending order.
- Hence in insert function care has to be taken while inserting such that the order is maintained.



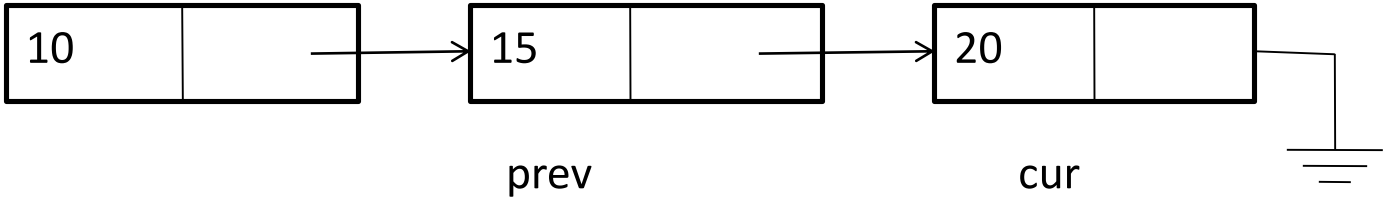
- In the above ordered list, if 12 is to be inserted, it has to be inserted between node1 and node2.



```
/*function to insert an item in ordered linked list*/
NODEPTR insert(int item, NODEPTR first)
{
    NODEPTR temp,cur,prev;
    temp=getnode( ) ;   /* obtain a new node*/
    temp→info =item;
    temp→link=NULL;

    if(first==NULL)
        return temp;

/*if  item is less than item of first node make new node as first node*/
    if(item<first→info)
    {
        temp→link=first;
        return temp;
    }
/*else traverse until the item is greater than the node items*/
prev=NULL;
cur=first;
while(cur !=NULL && item >cur→info)
{
    prev=cur;
    cur=cur→link;
}
/*set link of previous node to new node and link of new node to the
    current node*/
prev→link=temp;
temp→link=cur;
return first;
}
```



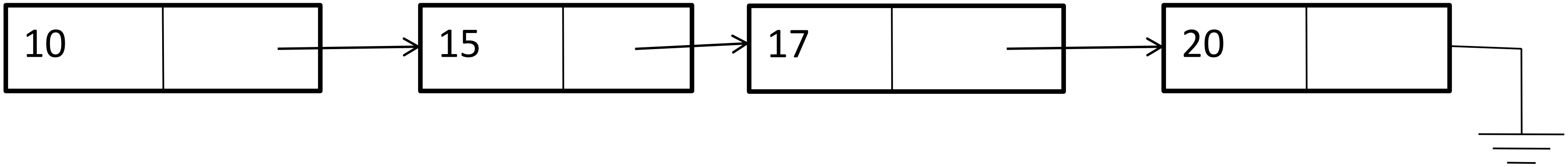
While inserting 17 to the above list, when the traversal stops, the list looks like this



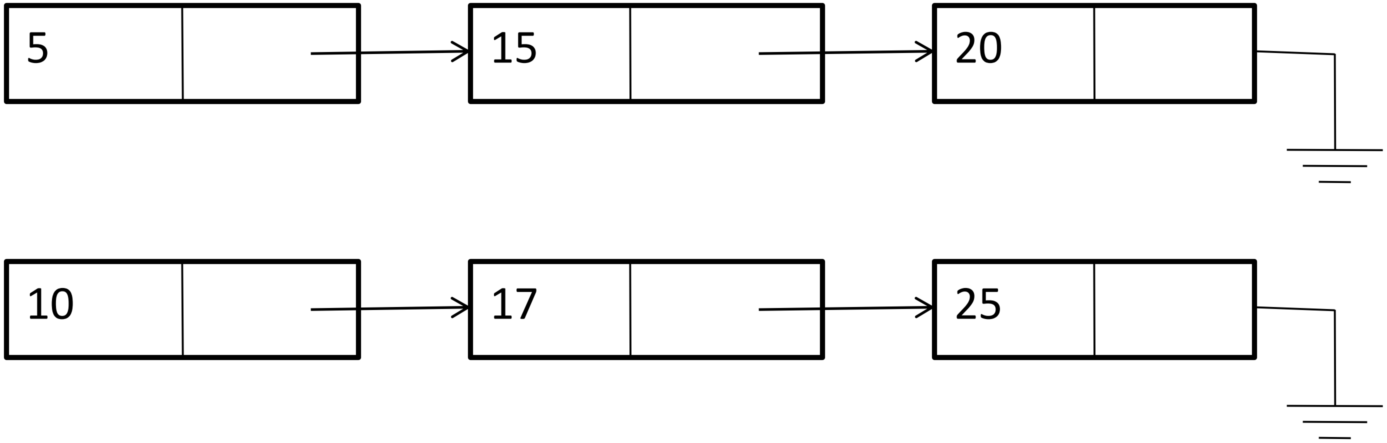
Now after doing

prev→link=temp;

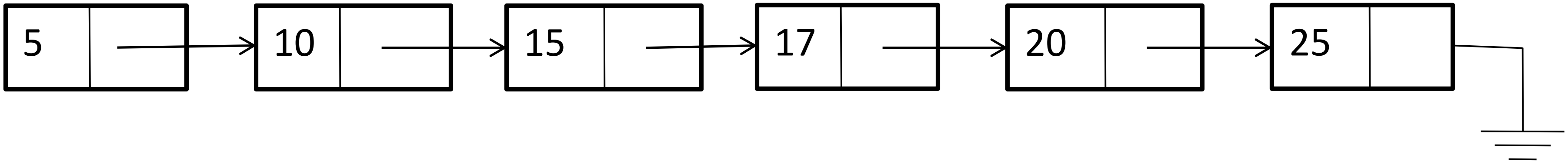
temp→link=cur;



Merging 2 ordered lists:

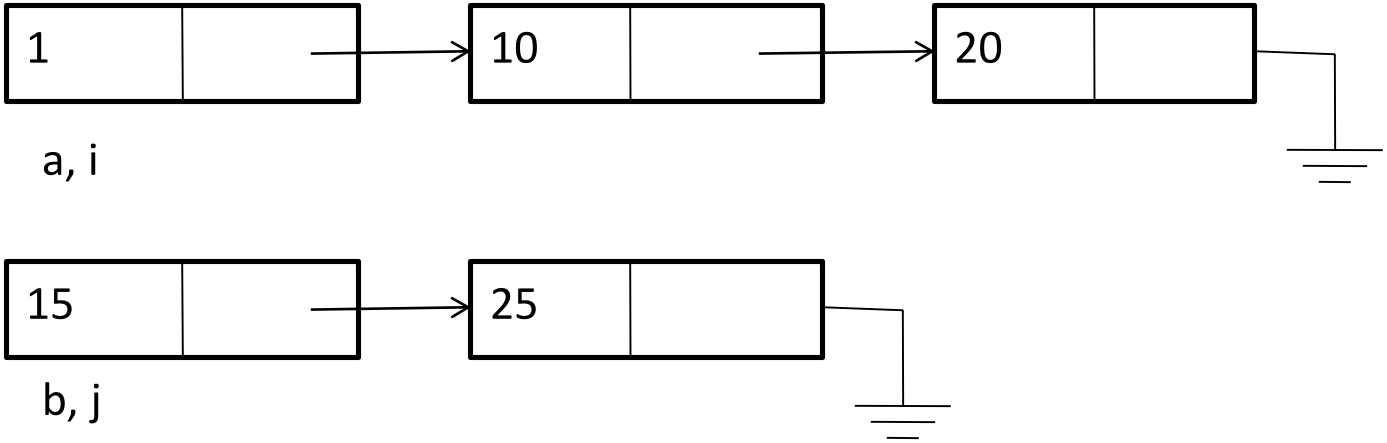


If the above two lists are merged, the resulting list would look like

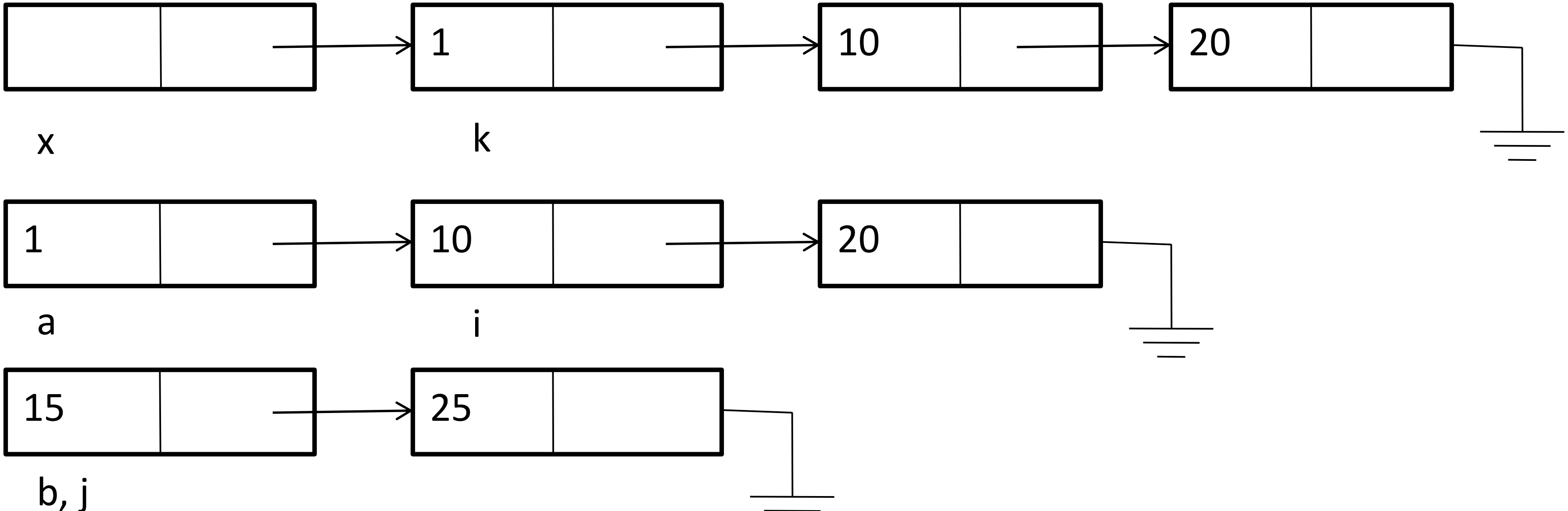


```
/*program to merge two ordered lists*/
NODEPTR merge (NODEPTR a, NODEPTR b)
{
    NODEPTR i,j,k,x;
    i=a; j=b;
    x=k=getnode( );
    while( i!=NULL && j!=NULL)
    {
          if(i→info< j→info)
          {
                    k→link=i;
                    i=i→link;
          }
else
{
    k→link=j;
    j=j→link;
}
k=k→link;
}
if(i!=NULL)
    k→link=i;
else
    k→link=j;
return (x→link);
}
```

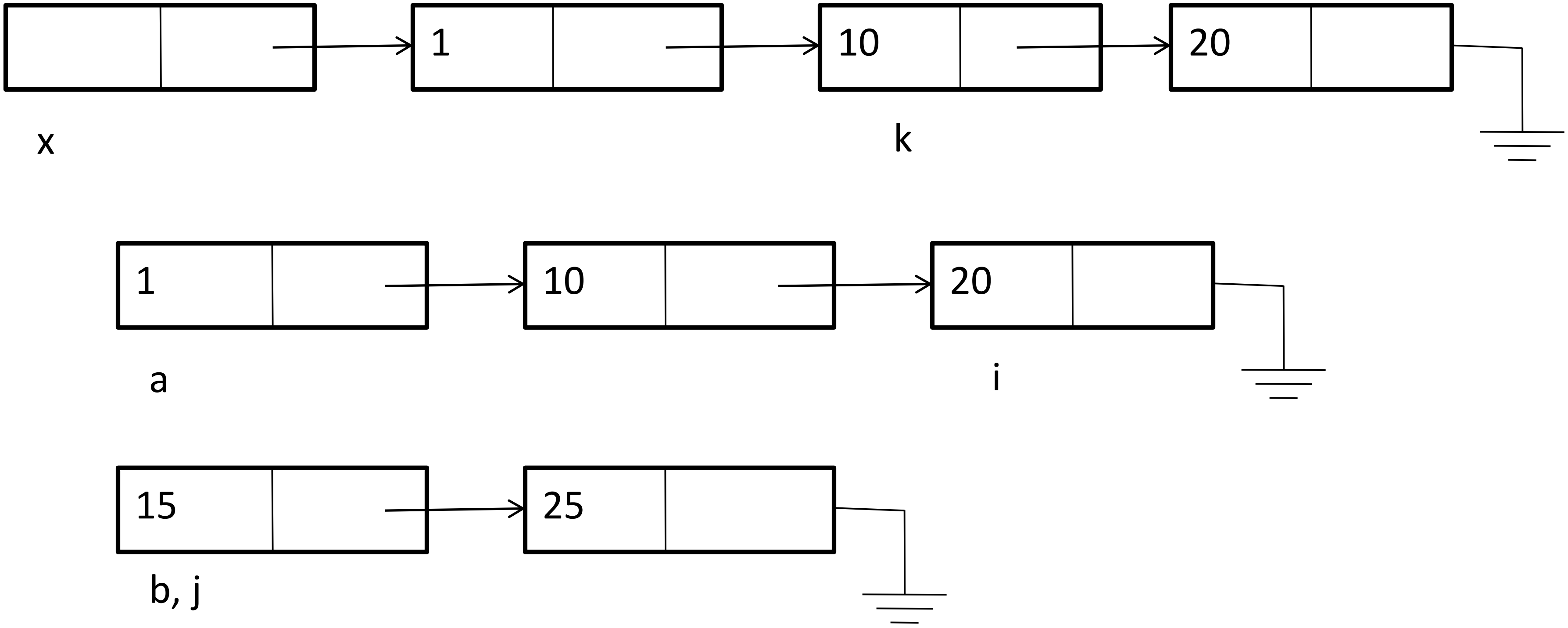
Merging 2 ordered lists:



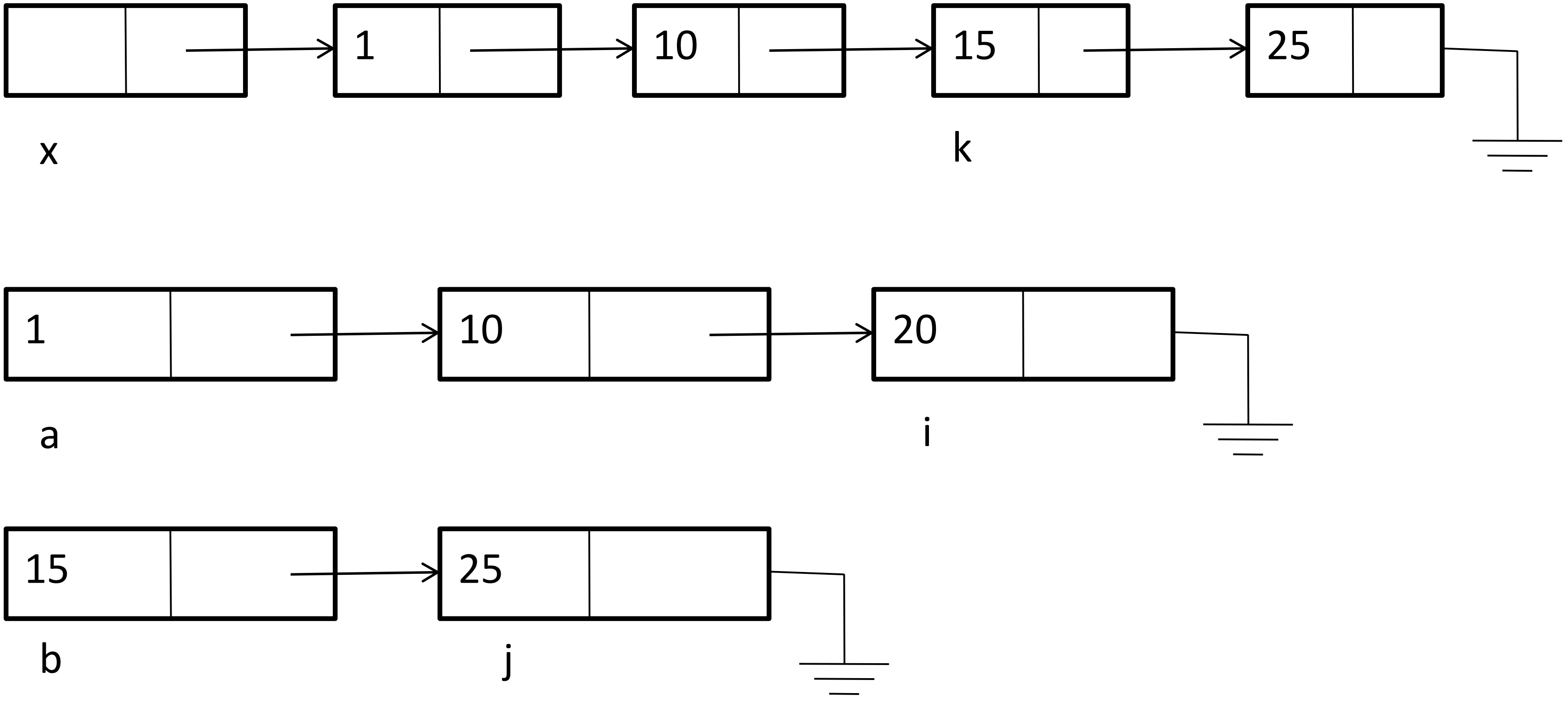
In the first iteration, i→info<j→info. Hence after 1st iteration, lists look like



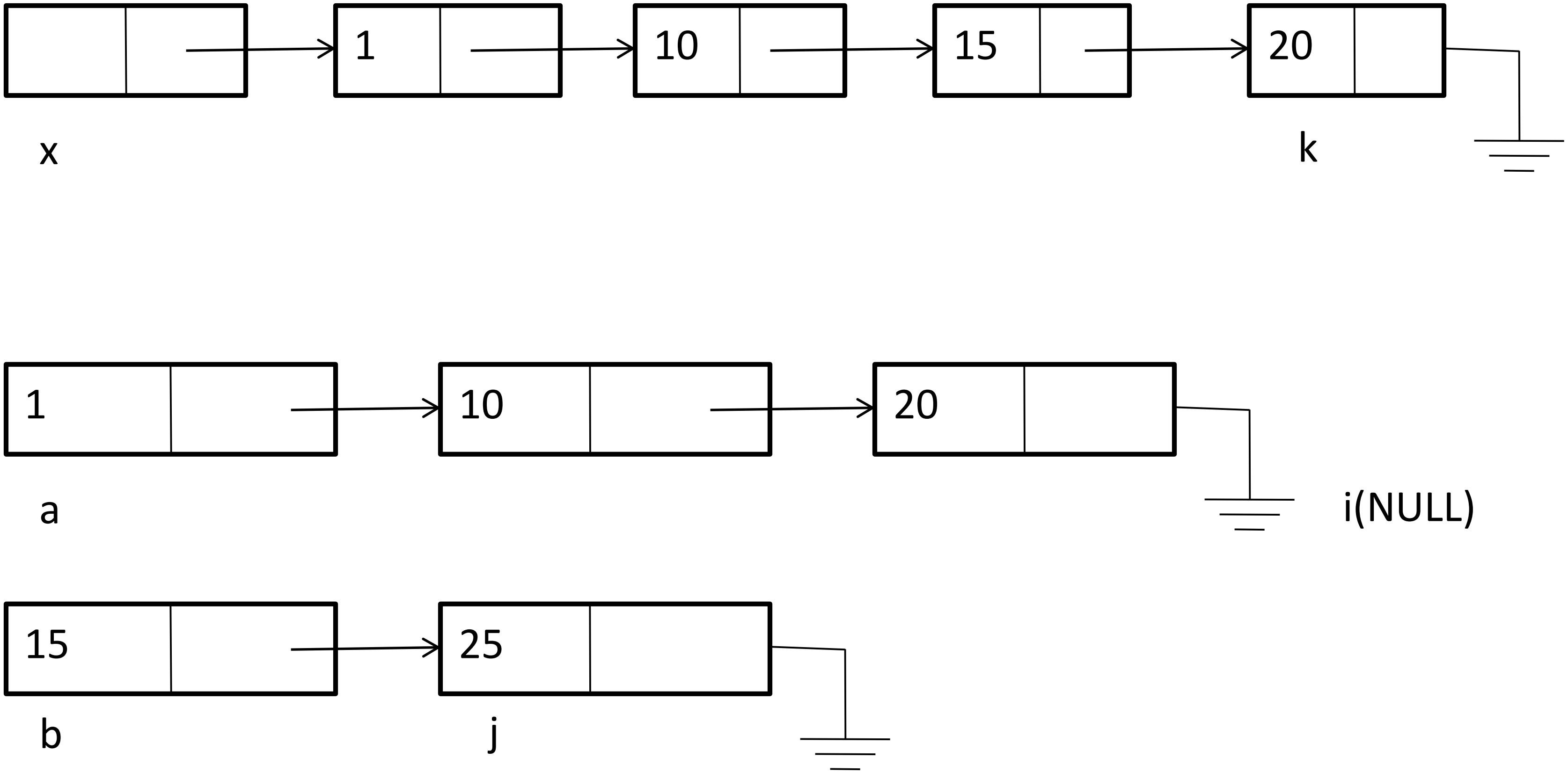
In the second iteration, i→info<j→info. Hence after 2nd iteration, lists look like



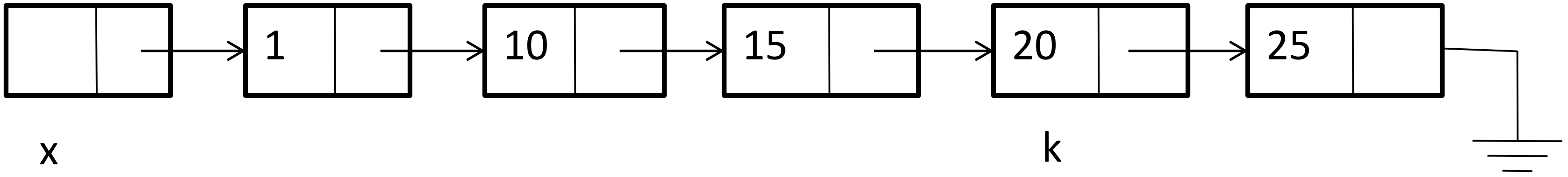
In the third iteration, i→info>j→info. Hence after 3rd iteration, lists look like



In the fourth iteration, i→info<j→info. Hence after 4th iteration, lists look like



Since i==NULL, while loop terminates. Now else condition after while loop is executed and final list will be



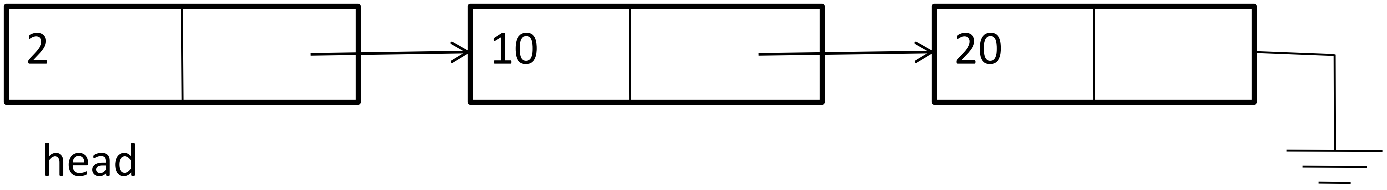
Finally x→link, i.e node with item 1 is returned.

Exercise:

WAP to sort a linked list.

Header nodes:

- Dummy nodes which are present at the beginning and their info field does not represent any item in the list.
- Info field is usually used to specify the number of nodes in the list.



Searching an item in the list:

- Start from first node and traverse until item is found.
- If item is found, display the position.
- If item not found(end of list is reached), display the error.

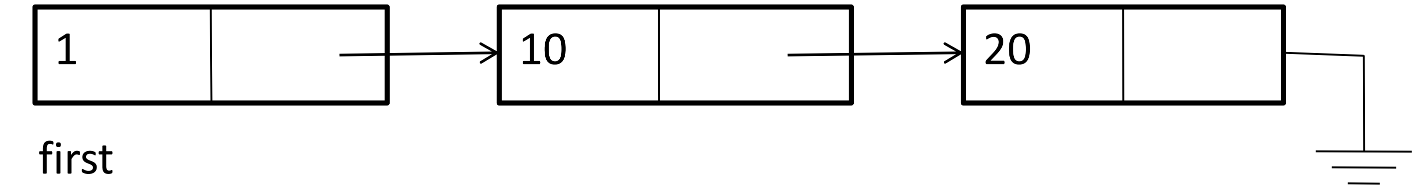
Concatenating 2 lists:

- If first list is empty, return first node of second list.
- If second list is empty, return first node of first list.
- If both are non empty, traverse till the end of first list and make the link field of last node point to the first node of second list.
- Finally return the first node of first list.

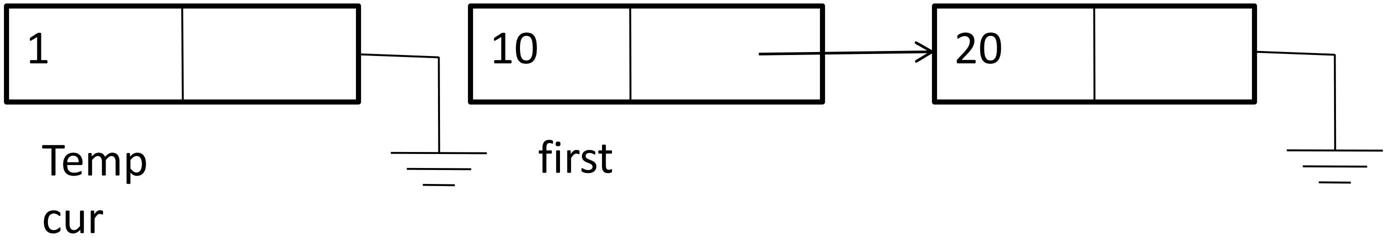
Reversing a list without creating new nodes:

```
NODEPTR reverse( NODEPTR first)
{
    NODEPTR cur,temp;
    cur = NULL;
    while(first!=NULL)
    {
          temp=first;
          first=first→link;
          temp→link=cur;
          cur=temp;
    }
return cur;                              // or first =cur;
}
```

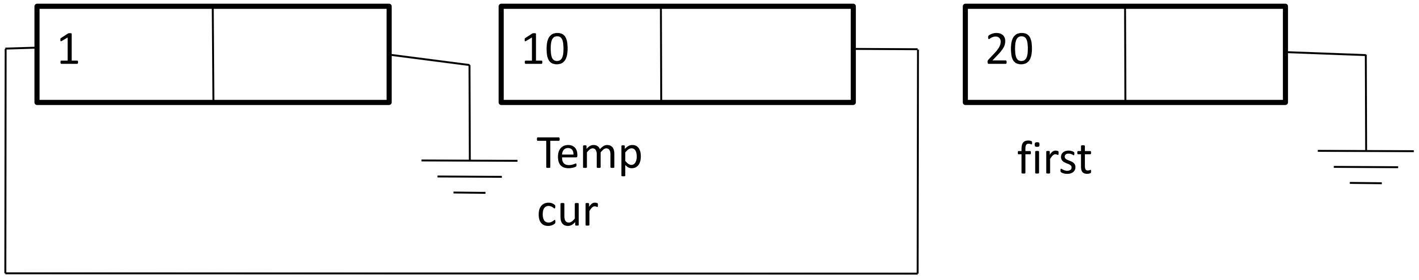
Ex:initially cur==NULL



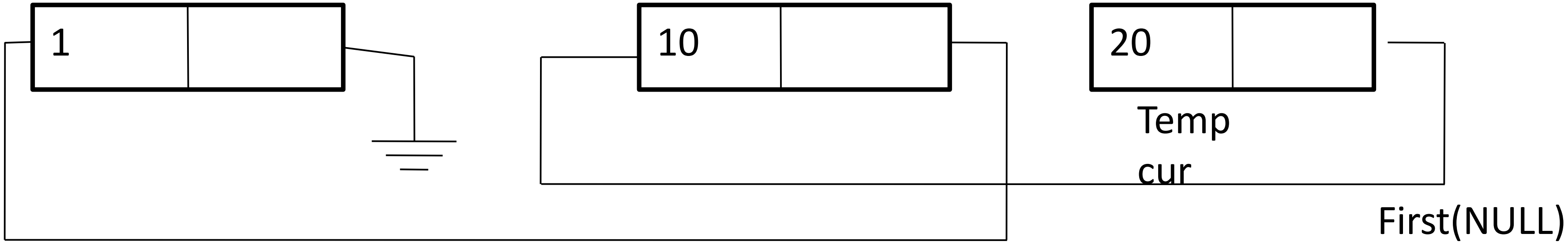
After 1st iteration



After 2nd iteration



After 3rd iteration



Since first is NULL, while loop terminates and cur is the first node and hence it is returned.

Non homogeneous lists:

```
struct employee
{
   int EID;
   double salary;
   char name[15];
   struct employee *link;
};
Typedef struct employee *NODEPTR;
```

Operations performed can be

1.Insert_front(int id,double sal, char nam[], NODEPTR first)

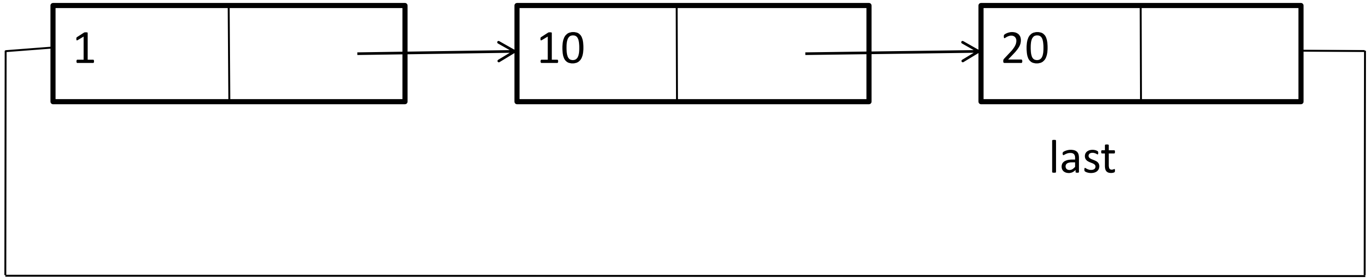
2.insert_rear(/* same as above*/)

3. delete(int id, NODEPTR first)

4.display(NODEPTR first)

Circular linked list

- In singly linked list, given a node x, we can reach all the nodes after that node but not nodes before node x.
- This happens because last node points to NULL and hence there is no way to get back to the beginning of list.
- In circular linked list, last node points to first node.



- Circular linked list is implemented by keeping track of last node. Hence we use last node instead of first node.
- Initially last is set to NULL.
- Insert and delete functions of circular linked list return last node unlike singly linked list functions.

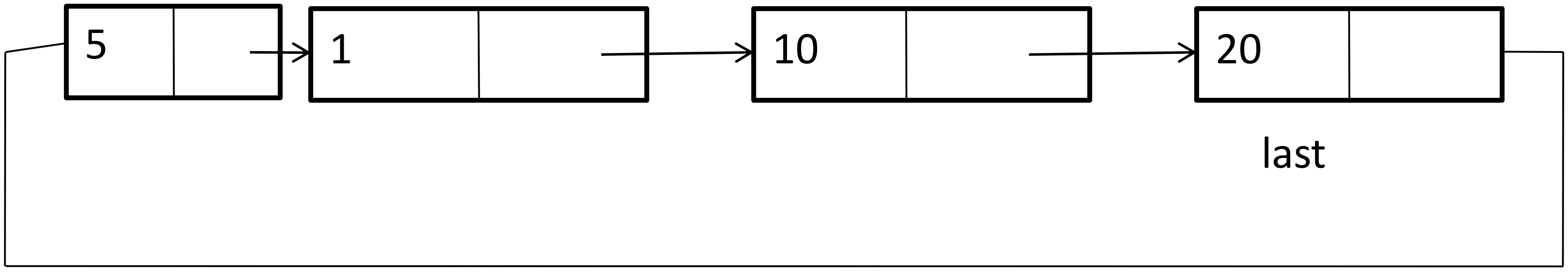
Operations on circular linked list:

1. Insert node at front end of list.
2. Insert node at rear end of list.
3. Delete node at front end.
4. Delete node at rear end.

Insert node at front end of list:

- New node should point to first node and last node should point to new node.

After adding a node with item 5 at front of the previous list



```
/*insert_front of a circular linked list*/
void insert_front(int item, NODEPTR last)
{
    NODEPTR temp;
    temp=getnode( );
    temp→info=item;
    if(last== NULL)
    {
          last=temp;
          last→link=temp;
    }
    else
    {
          temp→link=last→link;
          last→link=temp;
    }


}
```

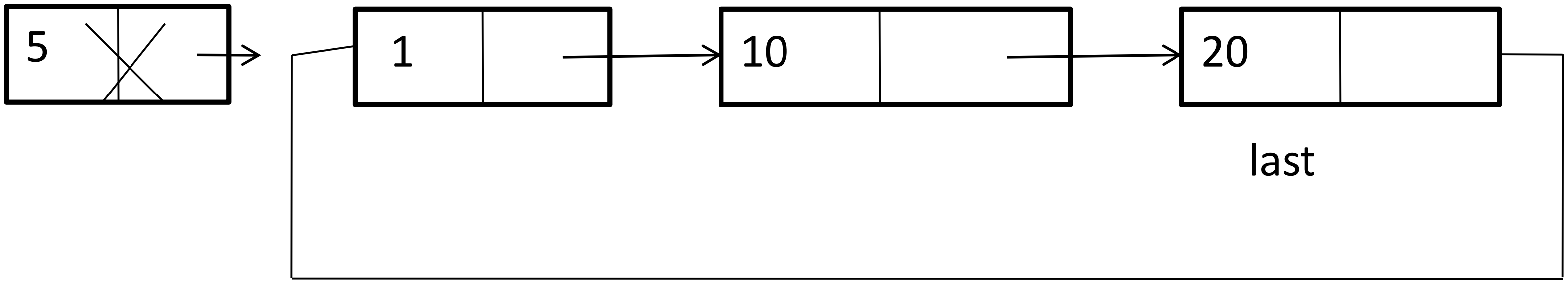
Inserting a node at rear end of list:

- New node should point first node and last node should point to new node.
- Finally new node is made the last node.
- The only difference between insert_front and insert_rear is that in insert_rear, new node is made the last node after insertion.

```
if(last==NULL)
{
        last=temp;
        last→link=temp;
}
else
{
        temp→link=last→link;
        last→link=temp;
        last=temp;
}
```

Deleting a node from front end:

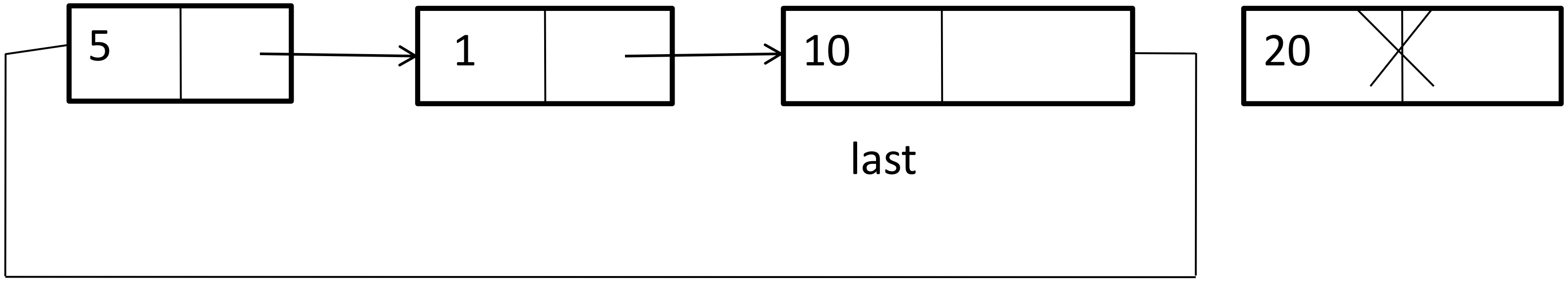
- Last node should be pointed to second node and first node is deleted.
- Last node is returned from the function.



```
/*delete front of a circular linked list*/
void delete_front(NODEPTR last)
{
    NODEPTR temp;
    if(last== NULL)
    {
          printf(“list empty”);
    }
    if(last→link==last)          /*if only one is present, delete it
    {                                 and reset last==NULL*/
          free(last);
          last==NULL
    }
    else
    {
    temp=last→link;    /*obtain address of first node*/
    last→link=temp→link;  /*make last node point to 2nd */
    free(temp);              /*free first node*/
    }
}
```

Deleting a node from rear end:

- Last but one node is made to point to first node and current last node is deleted.
- New last node(previous last but one node) is returned from the function.



```
/*delete rear of a circular linked list*/
void delete_rear(NODEPTR last)
{
    NODEPTR prev;
    if(last== NULL)
    {
          printf(“list empty”);
    }
    if(last→link==last)                    /*if only one is present, delete it
    {                                   and reset last==NULL*/
          free(last);
          last=NULL
    }
    else
    {prev=last→link;             /*obtain address of first node*/

    while(prev→link!=last)      /*traverse till last but one node */
          prev=prev→link;
    prev→link=last→link;        /*make prev node point to 1st node

    free (last);                  /*free last node*/
    last=prev;  }                 /* make prev as last node */
}
```

Displaying the circular list

```
Void display( NODEPTR last)
{
    NODEPTR temp;
    if(last==NULL)
    {
           printf( “list is empty”);
           return;
    }
printf(“contents are”);
/* traverse till last but one node and print info*/
for(temp=last→link;temp!=last;temp=temp→link)
    printf(“%d”, temp→info);
/*print the last info*/
Printf(“%d”, temp→info);
}
```

Circular list as a queue

- To implement a queue using circular linked list, insert at front and delete at rear.

Circular list as a stack

- To implement a stack using circular linked list, insert at front and delete at front.

Deleting a node in a list whose info is specified

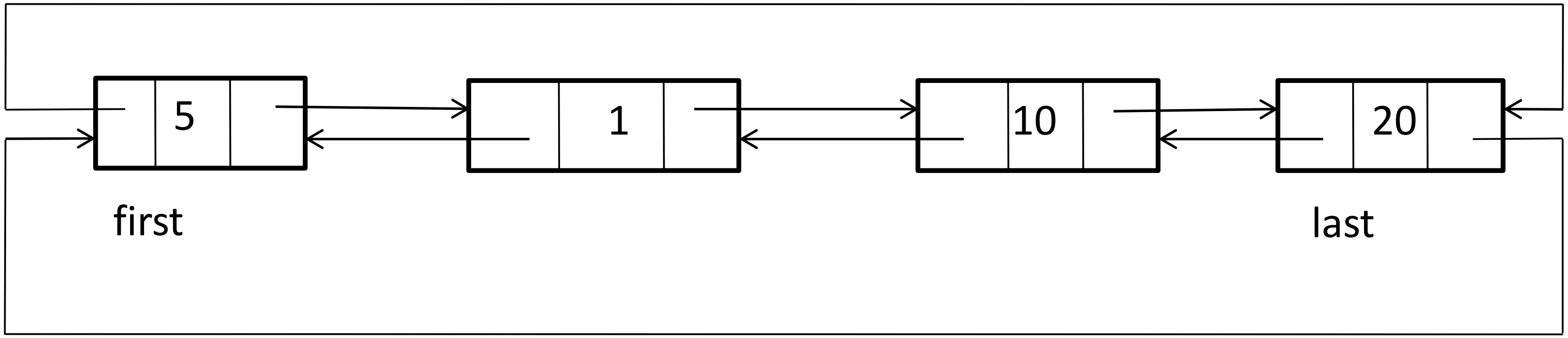
- Traverse the list until the info is obtained.
- Delete the node and set the links accordingly.

Inserting a node at specified position

- Traverse the list till the specified position is reached.
- Insert the node and set the links accordingly.

Doubly circular linked list

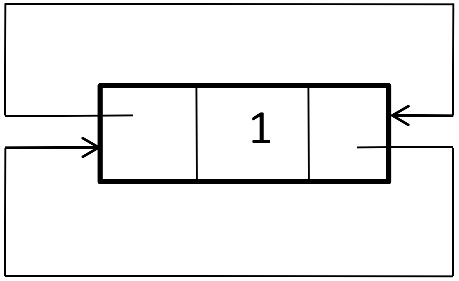
- In singly list, given a node, all its previous are not reachable.
- In circular list, given a node, we need to traverse the entire list to reach the previous node.
- If we can have another link in a node which points to its immediate previous node, it would be more useful.



- In doubly circular linked list, each node has 3 fields
  1. Item or info
  2. Left link
  3. Right link

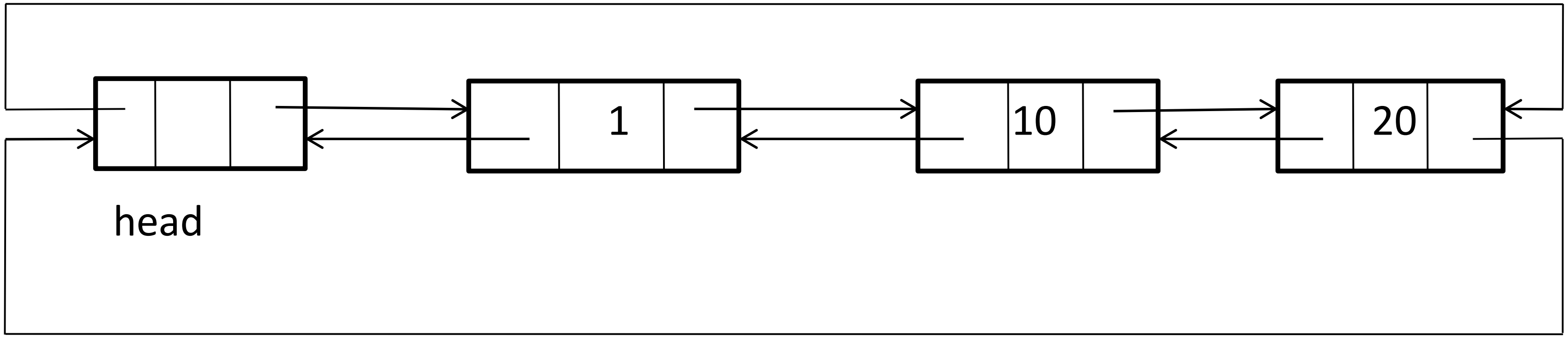
```
Struct dlist
{
    int item;
    struct dlist *llink;
    struct dlist *rlink;
};
Typedef struct dlist *NODEPTR;
```

<u>Doubly linked list with a single node</u>



Implementing a doubly circular linked list:

- To simplify the implementation of a linked list, we make use of a header node.
- This header node is a dummy node which is at the beginning of a list and whose info is of no importance to the list.
- Info of header may be left vacant or it may contain the number of nodes in the list.
- Here we do not use first or last nodes.

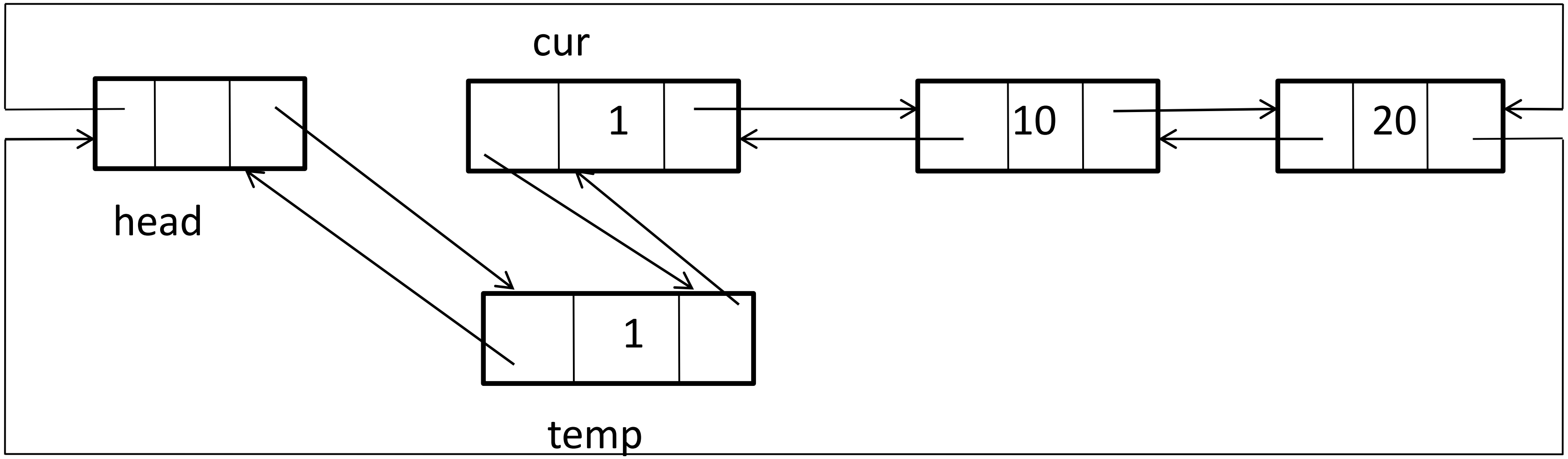


C program to implement doubly circular linked list:

```
#include<stdio.h>
#include<alloc.h>
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};
typedef struct node * NODEPTR;
Void main( )
{
    NODEPTR head;
    int choice,item;
    head = getnode( );
    head→rlink=head;          head→llink=head;
    for(; ;)
    {
          printf(“1.insert 2. insert rear 3. delete front 4. delete rear 5.display 6. exit”);
printf(“enter the choice”);
scanf(“%d”,&choice);
switch(choice)
{
    case 1:printf(“enter the item to be inserted”);
              scanf(“%d”,&item);
              head=insert_front(item,head);
              break;
    case 2:printf(“enter the item to be inserted”);
              scanf(“%d”,&item);
              head=insert_rear(item,head);
              break;
    case 3:head=delete_front(head);
              break;
    case 4:head=delete_front(head);
              break;
    case 5:display(head);
              break;
    default: exit(0);
}}}
```

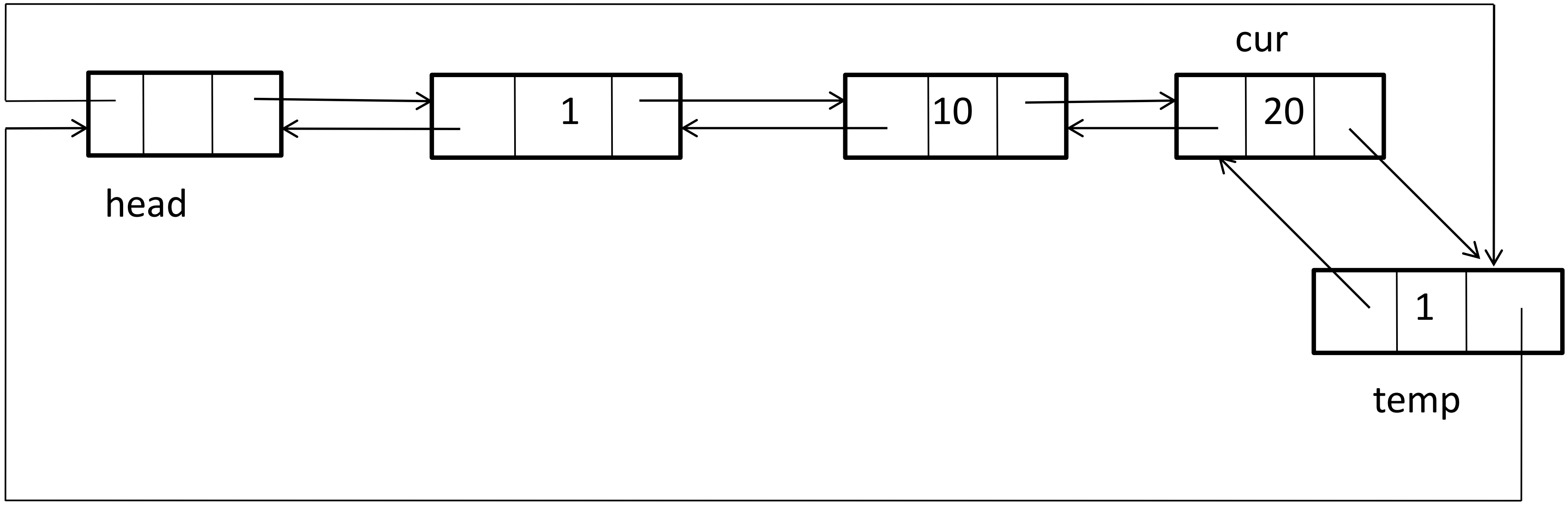
```
/*function to insert an item at front */
NODEPTR insert_front(int item, NODEPTR head)
{
    NODEPTR temp,cur;
    temp=getnode( ) ;          /* obtain a new node*/
    temp→info =item;
    cur=head→rlink;            /*obtain first node*/
    head→rlink=temp;           /* point rlink of head to new node
    temp→llink=head;           /*point llink of new node to head
    temp→rlink=cur;            /*point rlink of new node to
                                 first node*/
    cur→llink=temp;            /*point llink of  first node
                                 to new node*/
    return head;
}
```

Inserting a node at front of doubly circular list



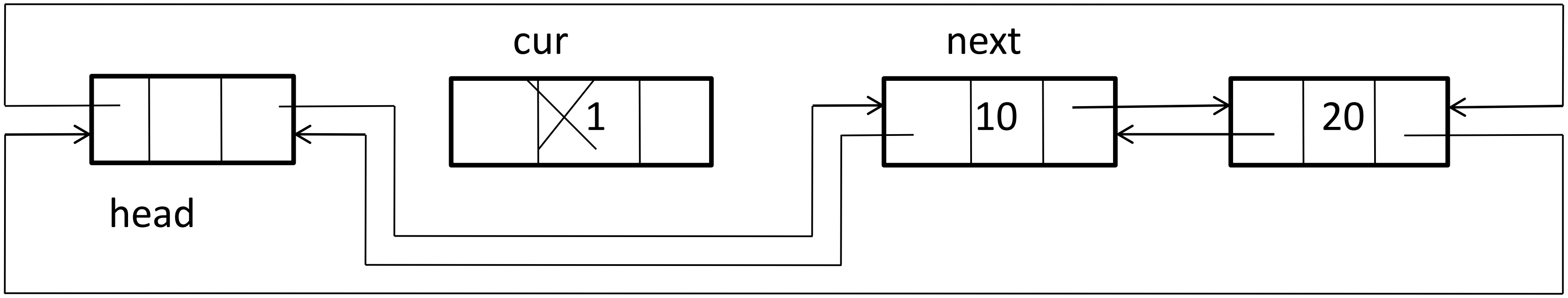
```
/*function to insert an item at rear */
NODEPTR insert_rear(int item, NODEPTR head)
{
    NODEPTR temp,cur;
    temp=getnode( ) ;          /* obtain a new node*/
    temp→info =item;
    cur=head→llink;            /*obtain last node*/
    head→llink=temp;           /* point llink of head to new node
    temp→rlink=head;           /*point rlink of new node to head
    temp→llink=cur;            /*point llink of new node to
                                 last node*/
    cur→rlink=temp;            /*point rlink of  last node
                                 to new node*/
    return head;
}
```

Inserting a node at rear of circular doubly list



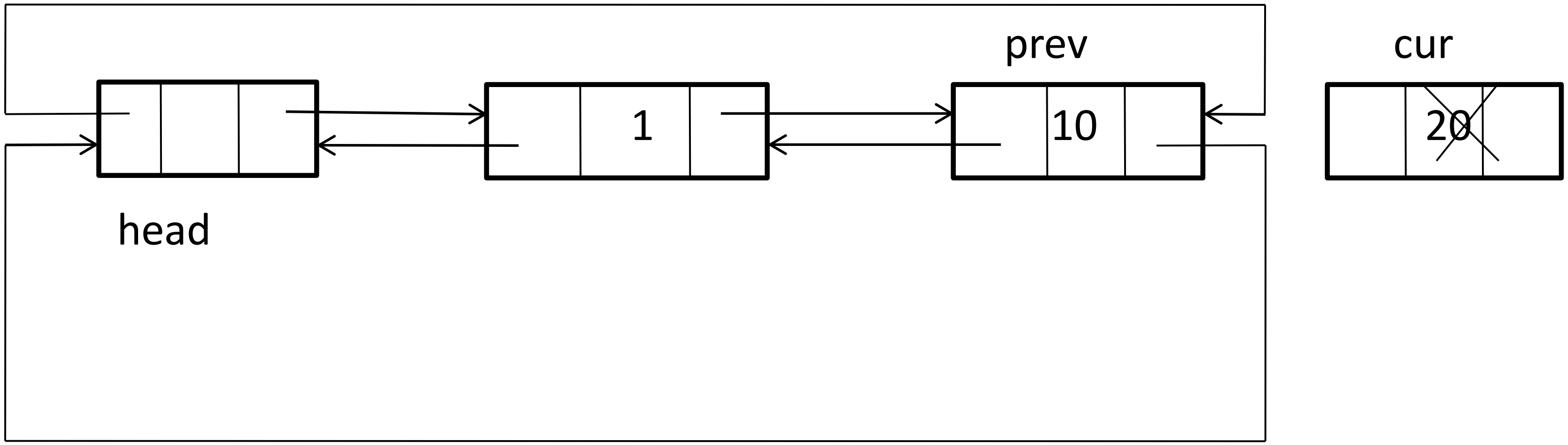
```
/*function to delete a node at fornt*/
NODEPTR delete_front(NODEPTR head)
{
   NODEPTR cur,next;
   if(head→rlink==head)
   {
        printf(“list empty”);
        return head;
   }
   cur=head→rlink;       /*get 1st node which is to be deleted*/
   next=cur→rlink;       /*get 2nd node which will become
                           the new 1st node after deletion*/

   head→rlink=next;     /*point rlink of head to new 1st node*/

   next→llink=head;      /*point llink of new 1st node to head*/

   free(cur);                /*free previous first node*/

   return head;

}
```

Deleting a node at front of doubly circular list



```
/*function to delete a node at rear*/
NODEPTR delete_rear(NODEPTR head)
{
    NODEPTR cur,prev;
    if(head→rlink==head)
    {
          printf(“list empty”);
          return head;
    }
    cur=head→llink;       /*get last node which is to be deleted*/
    prev=cur→llink;       /*get last but one node which will be
                            the new last node after deletion*/

    head→llink=prev;      /*point llink of head to new last node*/

    prev→rink=head;       //point rlink of new last node to head

    free(cur);            /*free previous last node*/

    return head;

}
```

Deleting a node at rear of doubly circular list



```
/*function to display the list*/
NODEPTR display(NODEPTR head)
{
    NODEPTR temp;
    if(head→rlink==head)
    {
        printf(“list empty”);
        return;
    }
    printf(“contents are”);
    for(temp=head→rlink; temp!=head; temp=temp→rlink)
        printf(“%d”, temp→info);
}
```

Advantages of doubly linked list:

- Both sides traversal is possible, whereas in singly list only one side.

Disadvantages of doubly list

- Requires additional storage for one extra link.

Exercise:

1. Implement deque using doubly list.
2. Implement doubly list without a header node.
3. Implement singly list with a header node.
4. Write a function to insert a node after a particular node in a doubly list whose info is given.
5. Write a function to delete a node in a doubly list whose info is given.

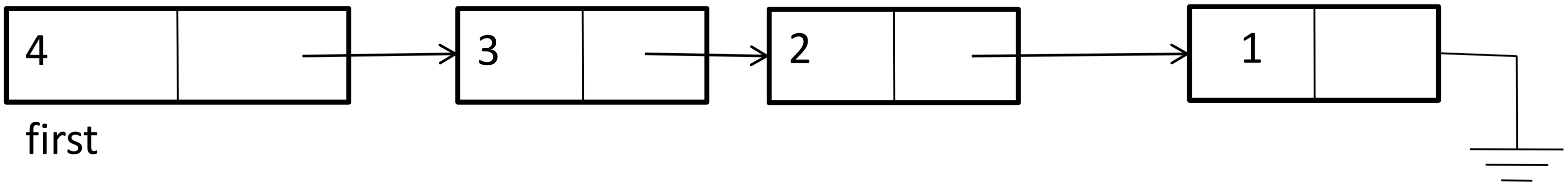
Reading a long positive number into singly linked list

Steps:

- create a node for the every digit entered by the user and add the node to the front of the list.
- Repeat the process until the number is entered.

For ex:

1234 can be represented as



Algorithm to add 2 long positive numbers using linked list

- Read the positive numbers into 2 singly linked lists by adding every digit encountered to the front of the list.

Algo:

```
int carry=0;
temp1=first1;     /* first node of first positive number*/
temp2=first2;      /* first node of 2nd  positive number*/
while(temp1!=NULL && temp2!=NULL)
{
      sum=temp1→info+temp2→info+carry;
      dig=sum%10;
      carry=sum/10;
      insert_front(dig,first3);    /* insert into resultant list*/
      temp1=temp1→link;
      temp2=temp2→link;
}
```

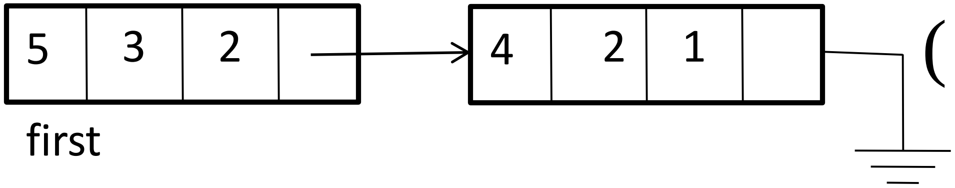
```
if(temp1!=NULL)
    temp=temp1;
else
    temp=temp2;
while(temp!=NULL)                    /* insert the digits remained
                                      in the non empty list to the
{                                     resultant list*/
    sum=temp→info+carry;
    dig=sum%10;
    carry=sum/10;
    insert_front(dig,first3);
    temp=temp→link;
}

if(carry)
    insert_front(carry, first3);
```

Display the list

Reading a polynomial with 2 variables into a singular linked list:

```
struct poly
{
    int info;
    int px;
    int py;
    struct poly *link;
};
typedef struct poly *NODEPTR;
```

5pow(x,3)pow)(y,2)+ 4pow(x,2)pow(y,1) can be represented as



(inserting node at rear)

```
/* function to read a polynomial*/
NODEPTR read_poly( NODEPTR first)
{
    int info, px, py;
    printf(“ To end polynomial enter -999 as info”);
    for(;;)
    {
        printf(“ enter the info”);
        scanf(“%d”, &info);
        if(info==-999)
            break;
        printf(“ enter power of x and y”);
        scanf(“%d %d”,&px,&py);
        insert_rear(info,px,py,first);
    }
}
```

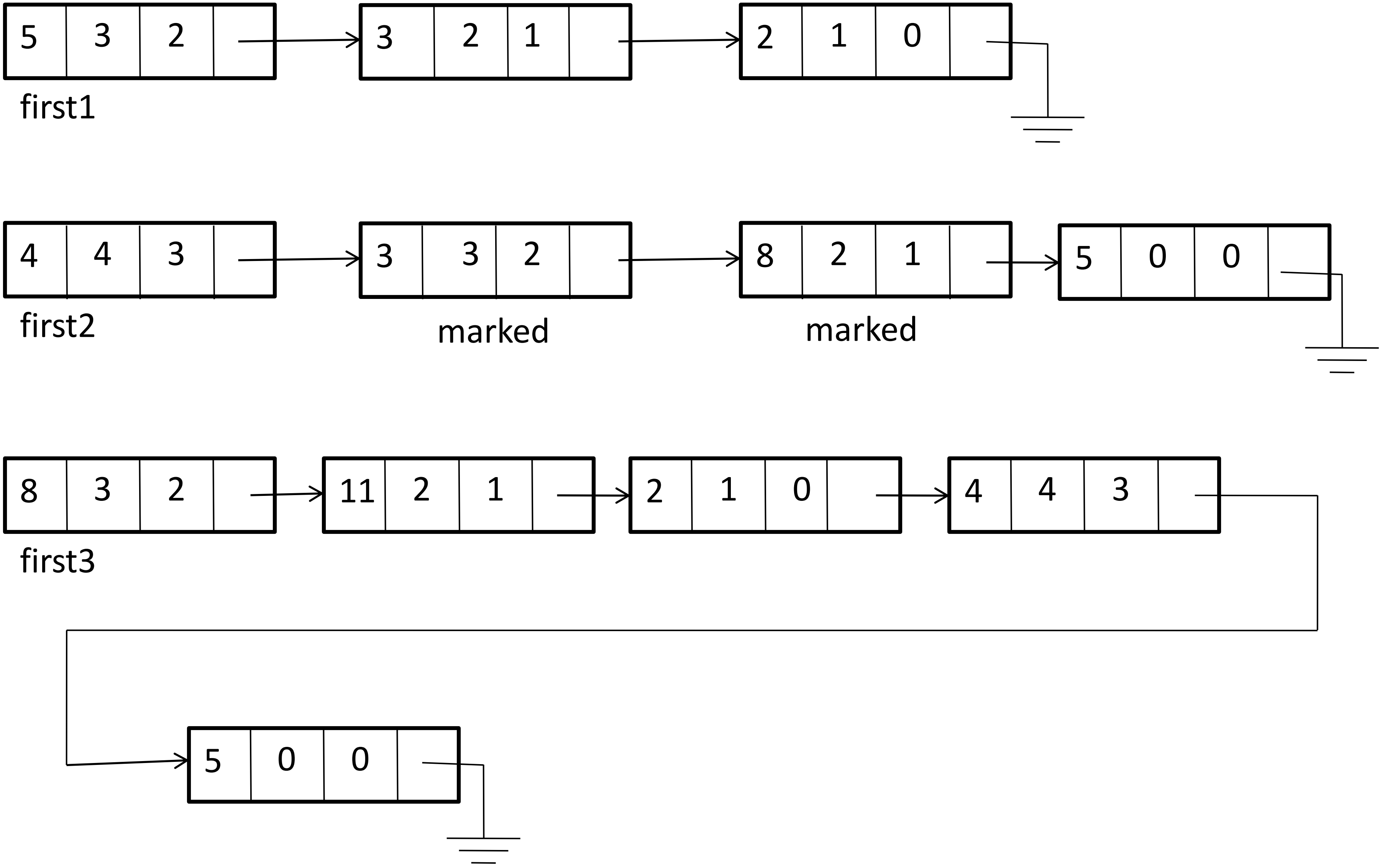
Adding 2 polynomials:

Steps:

- Read the given 2 polynomials into 2 singly linked lists.
- Perform the polynomial addition algorithm on the above lists.

Algo:

- For every node in the first list, check for the matching x and y node in the second polynomial list.
- If found, add the info of both nodes. If sum is not equal to zero, add the result(node) to resulting polynomial list at the rear end.
- Else if not found, add the first list node to resulting polynomial list at rear end.
- Repeat the process for each node in the first list.
- At the end, add whatever node is not matched in the second polynomial list to the resulting polynomial list at rear end.


5 3 2
3 2 1
2 1 0
first1
4 4 3
3 3 2
8 2 1
5 0 0
first2
marked
marked
8 3 2
11 2 1
2 1 0
4 4 3
first3
5 0 0

```
/*function to add 2 polynomials*/
int px1,px2,py1,py2, info1, info2, sumcf;
temp1=first1;
while(temp1!=NULL)
{
    px1=temp1→px;   py1=temp1→py; info1=temp1→info;
    temp2=first2;
    while(temp2!=NULL)
    {
        px2=temp2→px;
        py2=temp2→py;
        info2=temp2→info;
        if(px1==px2 && py1==py2)        /* if matching found,
                break;                   break to add info*/
        else
                temp2=temp2→link;       /* else move to next
                                         node in 2nd list*/
    }
if (temp2!= NULL)          /* matching found, hence
{                           add info of matching nodes*/
   sumcf=info1+info2;
   temp2→flag=1;           /* mark matching node in 2nd list
   if(sumcf!=0)
        insert_rear(sumcf, px1,py1,first3);
}
else
   insert_rear(info1, px1,py1,first3);
temp1=temp1→link;          /* move to next node in 1st list*/
}//end while
temp2=first2;
While(temp2!=NULL)         /* add unmatched nodes in 2nd
{                           list to result list*/
   if(temp2→flag==0)
   insert_rear(temp2→info,temp2→px, temp2→py, first3);
   temp2=temp2→link;
}
}// end function
```