Listas doblemente enlazadas
Colas dinámicas

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar

drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Clase número 6 - Ciclo lectivo 2022

Agenda

Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

Listas doblemente enlazadas: introducción

Una lista doble es también una colección de nodos, donde cada uno contiene la parte de datos (almacenamiento útil de la lista) y dos punteros a estructuras del mismo tipo. Uno de estos punteros apunta al nodo previo mientras que el otro hará lo correspondiente con el nodo siguiente.

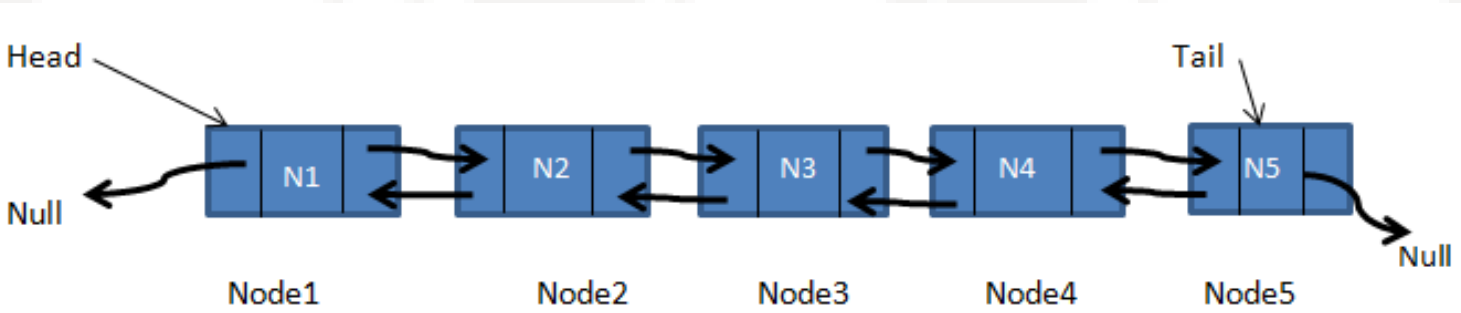


Figure: Representación gráfica de una lista doblemente enlazada.

Listas doblemente enlazadas: operaciones

- Creación de la estructura auto-referenciada
- Inserción
  - Al comienzo de la lista
  - Al final de la lista
  - Antes de un nodo en particular
  - Después de un nodo en particular
- Borrado
  - Primer nodo
  - Último nodo
  - Nodo que contenga un dato en particular
- Recorrido
  - Forward (de derecha a izquierda)
  - Backward (de izquierda a derecha)

Listas doblemente enlazadas: creación de la estructura auto-referenciada I

Cada nodo de la lista, debe contener:

- Una carga útil: por simplicidad se supondrá sólo un dato de tipo entero
- Un puntero a una estrucutura del mismo tipo apuntando al nodo anterior
- Un puntero a una estrucutura del mismo tipo apuntando al nodo siguiente

Listas doblemente enlazadas: creación de la estructura auto-referenciada II

```
1  struct Node
2  {
3      int data;
4      struct Node* next;
5      struct Node* prev;
6  };
```

Para el manejo de esta estructura desde la función principal main(), se definirá un puntero a la estructura de tipo Node:

Listas doblemente enlazadas: creación de la estructura auto-referenciada III

```
1  int main()
2  {
3  /* puntero al comienzo de la lista */
4  struct Node* head = NULL;
5
6  ...
```

Listas doblemente enlazadas: inserción de un nodo al comienzo de la lista I

1. Generar un nuevo nodo, si hay memoria suficiente, almacenar el dato en la variable correspondiente
2. Verificar si "head" es distinto de NULL, es decir, si existen nodos ya insertos en la lista:
   - Verdadero: el puntero al nodo siguiente debe asignarse al nodo que actualmente es el "head" de la lista. Es decir, al que actualmente es el primero. También debe asignarse el puntero prev del "head" al nodo recientemente creado
   - Falso: Asignar los punteros next y prev a NULL.
3. Apuntar con "head" al nuevo nodo.

Listas doblemente enlazadas: inserción de un nodo al comienzo de la lista II

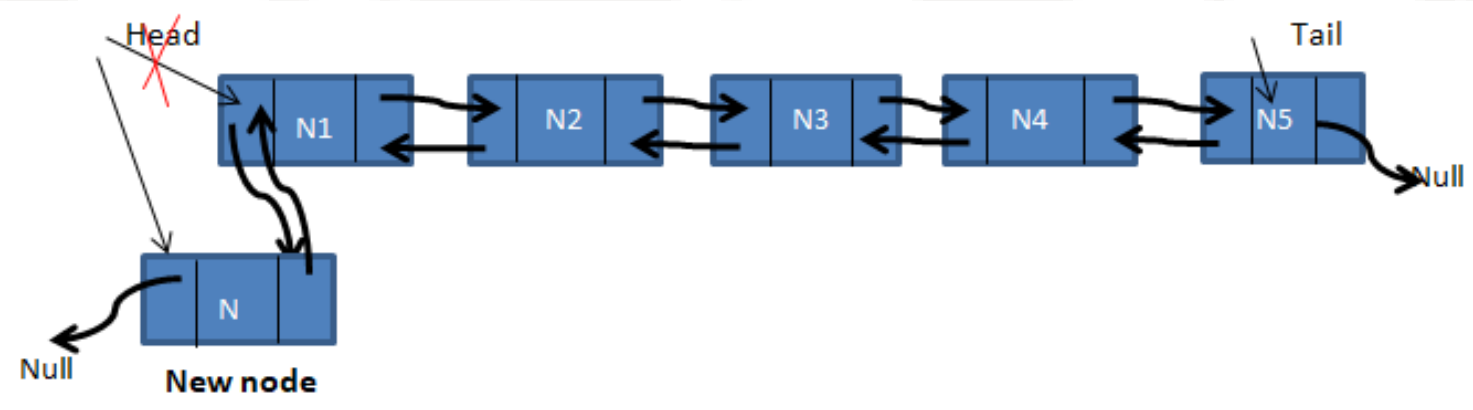


Figure: Inserción al inicio de una lista doble.

Listas doblemente enlazadas: inserción de un nodo al comienzo de la lista III

```
1  void insert_front(struct Node** head, int data)
2  {
3      struct Node* newNode = NULL;
4      try
5      {
6          newNode = new Node;
7          newNode->data = data;
8      }
9      catch (...)
10     {
11         cout << "No hay suficiente memoria";
12         exit(0);
13     }
14
```

Listas doblemente enlazadas: inserción de un nodo al comienzo de la lista IV

```
15      if ((*head) != NULL)
16      {
17          newNode->next = (*head);
18          newNode->prev = NULL;
19          (*head)->prev = newNode;
20      }
21      else
22      {
23          newNode->next = NULL;
24          newNode->prev = NULL;
25      }
26
27      (*head) = newNode;
28  }
```

Listas doblemente enlazadas: inserción de un nodo al comienzo de la lista V

Listas doblemente enlazadas: inserción al final de la lista I

1. Generar un nuevo nodo, si hay memoria suficiente, almacenar el dato en la variable correspondiente.
2. Como el nuevo nodo ingresado va al final de la lista, el puntero a "next" es asignado a NULL
3. Crear un nuevo puntero "last" que apuntará al último elemento de la lista, luego de realizar una búsqueda
4. Verificar si "head" es igual a NULL, es decir, NO existen nodos ya insertos en la lista:
   - Verdadero: el puntero a "prev" es asignado a NULL. Puesto a que el nodo ingresado es el único, head apunta a la dirección de este

Listas doblemente enlazadas: inserción al final de la lista II

- Falso: se busca cual es el último nodo de la lista y se lo apunta mediante "last". Luego, hacemos que el puntero "next" de last, apunte al nuevo nodo y el puntero "prev" del nuevo nodo a last.

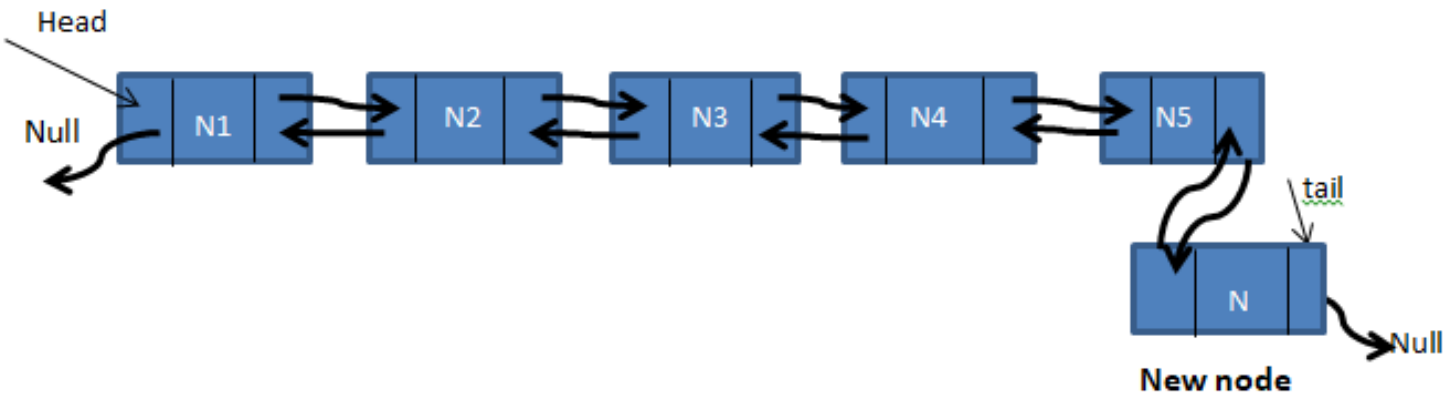


Figure: Inserción al final de una lista doble.

Listas doblemente enlazadas: inserción al final de la lista III

```
1  void insert_end(struct Node ** head, int new_data) {
2    struct Node * newNode = new Node;
3    struct Node * last = NULL;
4
5    try {
6      newNode = new Node;
7      newNode -> data = new_data;
8      newNode -> next = NULL;
9
10   } catch (...) {
11     cout << "No hay suficiente memoria";
12     exit(0);
13   }
14
15   last = * head;
```

Listas doblemente enlazadas: inserción al final de la lista IV

```
16
17      if ( *head == NULL) {
18          newNode -> prev = NULL;
19          *head = newNode;
20      } else {
21          while (last -> next != NULL)
22              last = last -> next;
23      }
24
25      last -> next = newNode;
26      newNode -> prev = last;
27
28  }
```

Listas doblemente enlazadas: inserción antes/después de un nodo en particular I

1. Verificar que el puntero al nodo recibido es valido. Si apunta a NULL, se debe abortar la estrategia
2. Generar un nuevo nodo, si hay memoria suficiente, almacenar el dato en la variable correspondiente.
3. Apuntar el puntero "next" del nuevo nodo, al "next" del nodo previo
4. Apuntar el puntero "next" del nodo previo, al nodo actual
5. Apuntar el puntero "prev" del nuevo nodo al nodo previo
6. Verificar si "next" del nuevo nodo es distinto de NULL
   - Verdadero: apuntamos "prev" del puntero siguiente al nuevo nodo a nuevo nodo

Listas doblemente enlazadas: inserción antes/después de un nodo en particular II

- Falso: lo dejamos en NULL. Es el último nodo de la lista

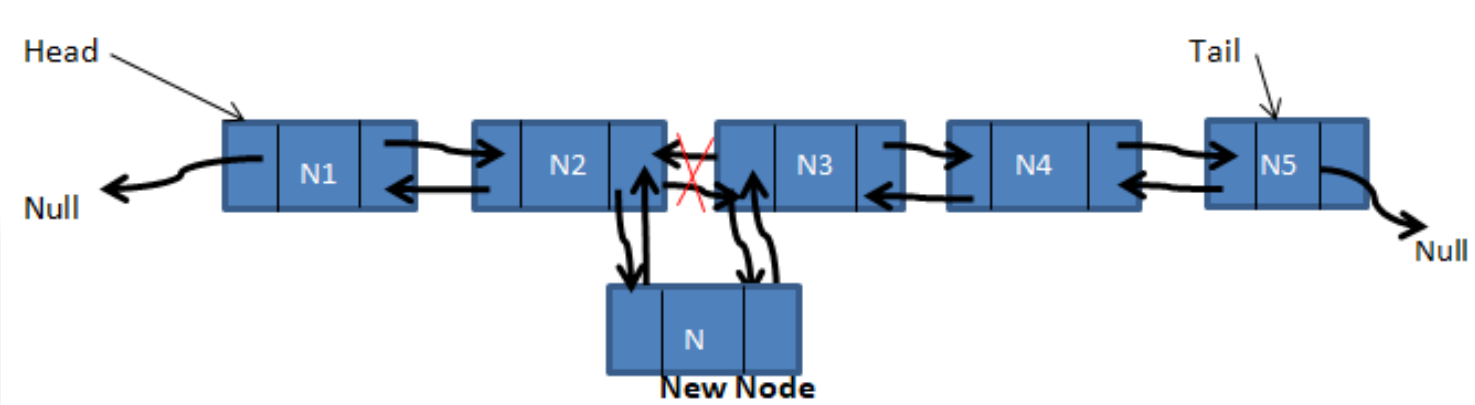


Figure: Inserción al final de una lista doble.

Listas doblemente enlazadas: inserción antes/después de un nodo en particular III

```
1  void insert_After(struct Node * prev_node, int new_d
2  {
3     struct Node * newNode = new Node;
4     newNode -> data = new_data;
5     if (prev_node == NULL)
6     {
7        cout << "Nodo previo invalido";
8        exit(0);
9     }
10
11
12
13
14
```

Listas doblemente enlazadas: inserción antes/después de un nodo en particular IV

```
15
16      try
17      {
18          newNode = new Node;
19          newNode -> data = new_data;
20      } catch (...)
21      {
22          cout << "No hay suficiente memoria";
23          exit(0);
24      }
25
26
27
28      newNode -> next = prev_node -> next;
```

Listas doblemente enlazadas: inserción antes/después de un nodo en particular V

```
29      prev_node -> next = newNode;
30      newNode -> prev = prev_node;
31
32      if (newNode -> next != NULL)
33        newNode -> next -> prev = newNode;
34  }
```

Listas doblemente enlazadas: borrado de un nodo I

- Verificar si el nodo a eliminar es el primero.
  - Verdadero: apuntar "head" a "head" next.
  - Falso:
    - Recorrer la lista hasta encontrar el nodo que se desea eliminar y almacenar un puntero a él en "del". Al hacer esto, también se debe obtener un puntero al nodo anterior al que se desea eliminar
    - Apuntar "next" del nodo previo a "next" del nodo a eliminar
    - Apuntar "del" del nodo siguiente al nodo a eliminar, a "prev" del nodo a eliminar

Listas doblemente enlazadas: borrado de un nodo II

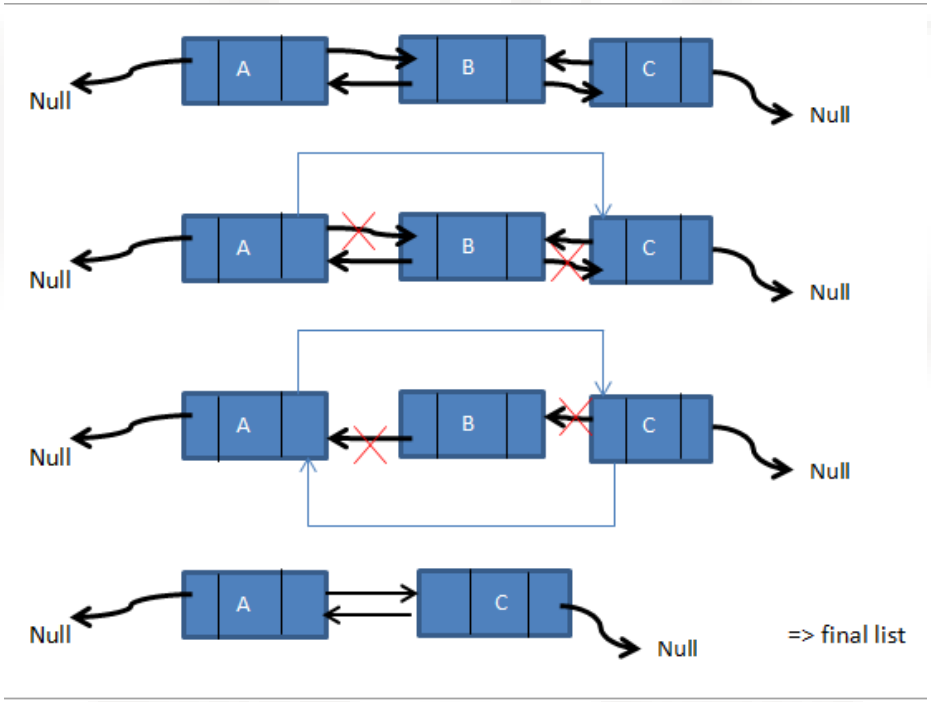


Figure: Eliminando un nodo de una lista doble.

Listas doblemente enlazadas: impresión I

```
1  void printing(struct Node* head)
2  {
3      struct Node* p = NULL;
4      struct Node* q = NULL;
5      printf("Reocorrido forward\n");
6      p=head;
7      while (p != NULL)
8      {
9          printf("%d\n",p->data);
10         q = p;
11         p = p->next;
12     }
13
14
15
```

Listas doblemente enlazadas: impresión II

```
16      printf("Reocorrido Backward\n");
17      while (q != NULL)
18      {
19          printf("%d\n",q->data);
20          q = p->prev;
21      }
22  }
```

Colas dinámicas: introducción

Una cola de espera es similar a la fila de pagos en una caja de un supermercado. La primera persona en la línea es atendida primero y los otros clientes ingresan por la parte final y esperan ser atendidos. Los nodos de la cola son solamente eliminados por la **parte delantera** y son incluidos o insertos únicamente or **la parte trasera**.

Este tipo de dato también es conocido por sus siglas en ingles FIFO[1].

[1] First Input First Output.

Colas dinámicas: aplicaciones

- Colas de impresion
- Sincronización de procesos de forma asíncrona: uso de la FIFO para sincronizar
- Manejo de interrupciones en sistemas de tiempo real
- Sistemas de esperas en Call Centers

Colas dinámicas: gráfico

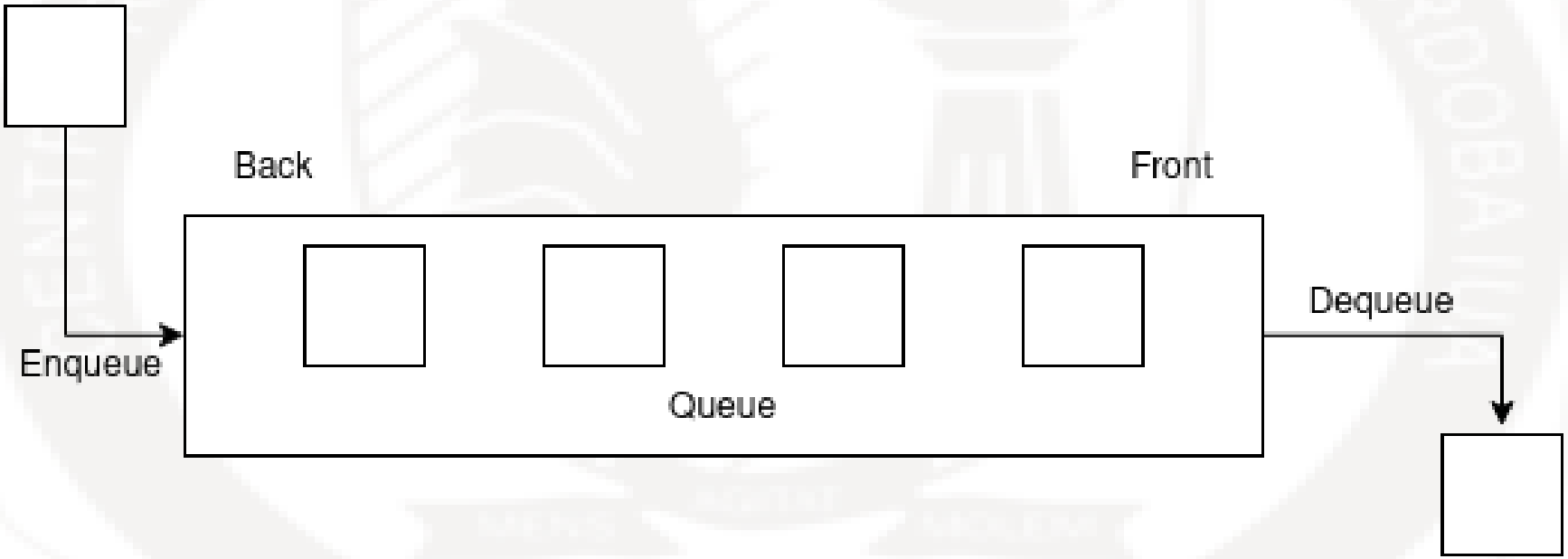


Figure: Representación gráfica de una cola.

Colas dinámicas: operaciones

Las operaciones más comunes con colas dinámicas son:

- Agregar un nodo
- Eliminar un nodo
- Calcular la cantidad de elementos almacenados en la FIFO
- Imprimir todos los nodos de la FIFO

Colas dinámicas: creación

```
1  struct node
2  {
3      int data;
4      struct node *link;
5  };
```

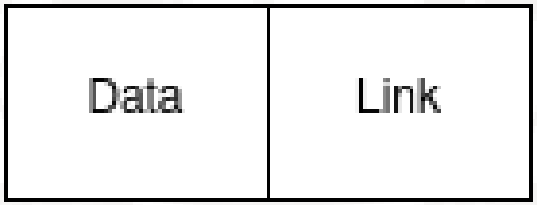


Figure: Representación gráfica de una cola.

Colas dinámicas: punteros a front y back

```
1    struct node *front=NULL;
2    struct node *back = NULL;
```

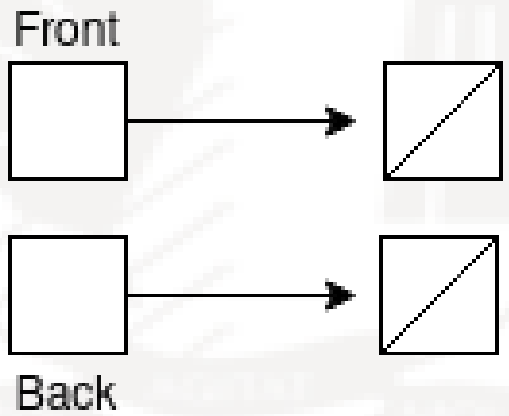
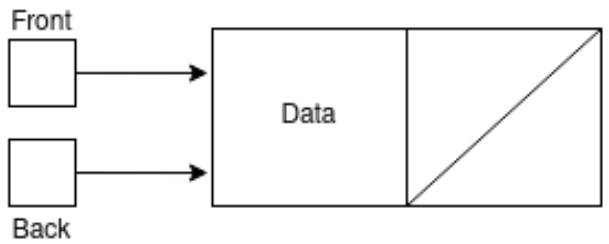


Figure: Punteros a la estructura front y rear de la queue.

Colas dinámicas: push del primer nodo

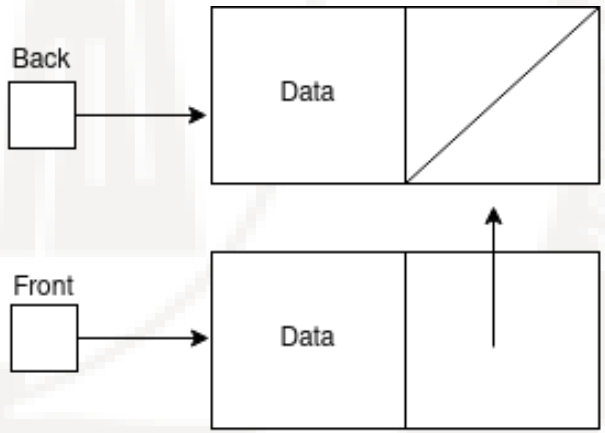
```
1  new_node=(struct node *)malloc(sizeof(struct node));
2
3  new_node->data = value;
4  new_node->link=NULL;
5
6  if(back == NULL)
7  {
8      back = new_node;
9      front = back;
10 }
11 else
12 {
13     back->link = new_node;
14     back = new_node;
15 }
```



Push del primer nodo

Colas dinámicas: push de los nodos siguientes

```
1  new_node=(struct node *)malloc(sizeof(struct node));
2
3  new_node->data = value;
4  new_node->link=NULL;
5
6  if(back == NULL)
7  {
8      back = new_node;
9      front = back;
10 }
11 else
12 {
13     back->link = new_node;
14     back = new_node;
15 }
```



Push del primer nodo

Colas dinámicas: push de los nodos siguientes

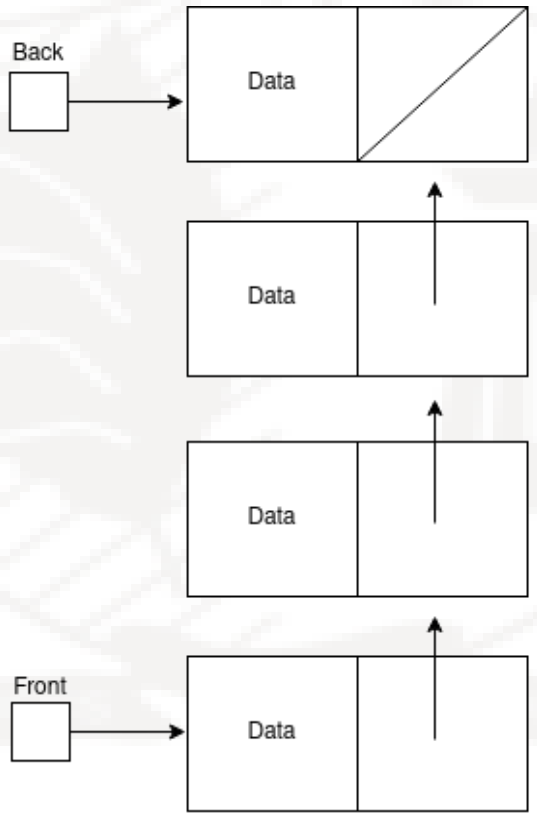
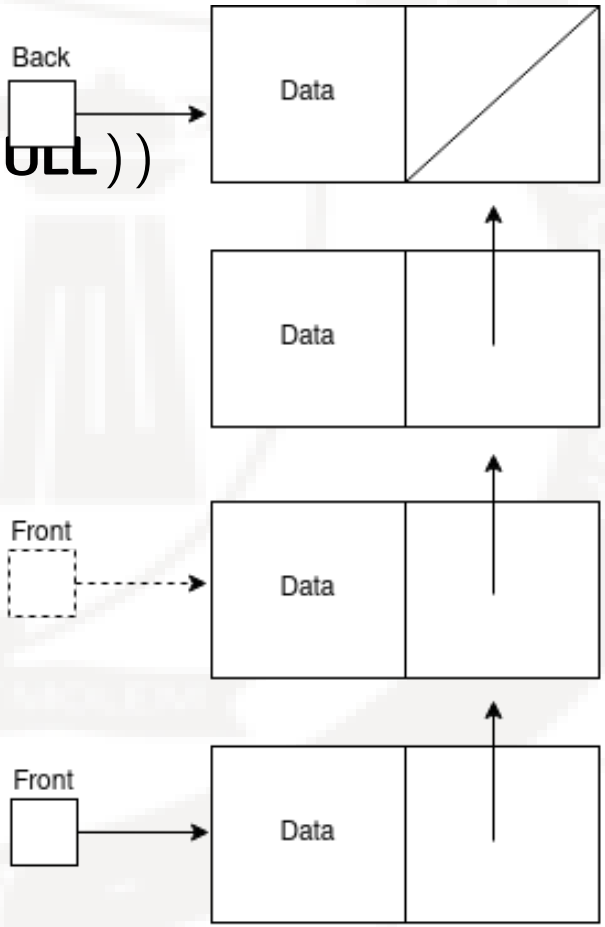


Figure: Representación gráfica de una cola.

Colas dinámicas: pop de nodos

```
1  if((front == back) && (back == NULL))
2  {
3      printf("Vacia");
4      exit(0);
5  }
6      temp = front;
7      front = (front)->link;
8      if(front == NULL)
9          rear = NULL;
10     free(temp);
11 }
```



Pop de un nodo

Colas dinámicas: pop de nodos

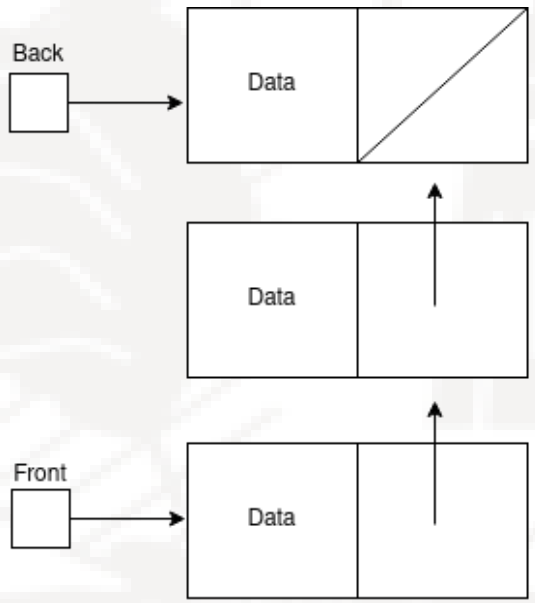


Figure: Representación gráfica de una cola.

Colas dinámicas: implementación I

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4  /*Definicion de la esctructura*/
5  struct node
6  {
7      int data;
8      struct node *link;
9  };
10
11
12
13
14
15
```

Colas dinámicas: implementación II

```
16  void main()
17  {
18      struct node *front  =NULL;
19      struct node *back   = NULL;
20      struct node *temp   =NULL;
21      int n,value,op=1;
22      printf("Ingrese un elemento\n");
23      scanf("%d",&value);
24      do
25      {   /*Creacion de un nuevo nodo*/
26          temp=(struct node *)malloc(sizeof(struct node));
27          if(temp==NULL)
28          {   printf("No Memory available\n");
29              exit(0);
30          }
```

Colas dinámicas: implementación III

```
31          temp->data = value;
32          temp->link=NULL;
33          if(back == NULL)   /*Insercion del primer nodo*/
34          {
35              back = temp;
36              front = back;
37          }
38          else /*Insercion del resto de los nodo*/
39          {
40              back->link = temp;
41              back = temp;
42          }
43          printf("0 para salir\n");
44          scanf("%d",&value);
45      }while(value!=0);
```

Colas dinámicas: implementación IV

```
46
47      /*Impresion de toda la FIFO*/
48      temp=front;
49      while(temp!=NULL)
50      {
51          printf("%d",temp->data);
52          temp=temp->link;
53      }
54
55
56
57
58
59
60
```

Colas dinámicas: implementación V

```
61      do /*Eliminacion de nodos*/
62      {
63          if((front == back) && (back == NULL))
64          {
65              printf("Vacia");
66              exit(0);
67          }
68          temp = front;
69          front = (front)->link;
70          if(front == NULL)
71              back = NULL;
72          free(temp);
73          printf("0 para salir\n");
74          scanf("%d",&value);
75      }while(value!=0);
```

Colas dinámicas: implementación VI

```
76
77          /*Impresion de toda la FIFO*/
78          temp=front;
79          while(temp!=NULL)
80          {
81                  printf("%d",temp->data);
82                  temp=temp->link;
83          }
84
85  }
```

Colas dinámicas: implementación con punteros I

```
1  # include <stdio.h>
2  # include <stdlib.h>
3
4  /*Definicion de la esctructura*/
5  struct node
6  {
7      int data;
8      struct node *link;
9  };
10
11 int menu (void);
12 void push(struct node **,struct node **, int );
13 void pop(struct node **,struct node **);
14 void print(struct node *);
15
```

Colas dinámicas: implementación con punteros II

```
16  int menu (void)
17  {
18      int op;
19      do
20      {
21          printf("1-Agregar un nodo\n");
22          printf("2-Borrar un nodo\n");
23          printf("3-Imprimir cola\n");
24          printf("4-Salir\n");
25          scanf("%d",&op);
26      }while((op<1)||(op>4));
27      return(op);
28  }
29
30
```

Colas dinámicas: implementación con punteros III

```
31  void push(struct node **front,struct node **back, int d)
32  {
33      struct node *temp;
34      temp=(struct node *)malloc(sizeof(struct node));
35      if(temp==NULL)
36      {   printf("No Memory available\n");
37          exit(0);
38      }
39      temp->data = d;
40      temp->link=NULL;
41      if(*back == NULL)   /*Insercion del primer nodo*/
42      {
43          *back = temp;
44          *front = *back;
45      }
```

Colas dinámicas: implementación con punteros IV

```
46          else /*Insercion del resto de los nodo*/
47          {
48              (*back)->link = temp;
49              *back = temp;
50          }
51      }
52
53      void pop(struct node **front,struct node **back)
54      {
55          struct node *temp;
56          if((*front == *back) && (*back == NULL))
57          {
58              printf("Vacia\n");
59              exit(0);
60          }
```

Colas dinámicas: implementación con punteros V

```
61          temp = *front;
62          *front = (*front)->link;
63          if(*back == temp)
64          {
65              *back = (*back)->link;
66          }
67
68          free(temp);
69      }
70
71
72
73
74
75
```

Colas dinámicas: implementación con punteros VI

```
76  void print(struct node *front)
77  {
78      struct node *temp   =NULL;
79      /*Impresion de toda la FIFO*/
80      temp=front;
81      while(temp!=NULL)
82      {
83          printf("%d->",temp->data);
84          temp=temp->link;
85      }
86      printf("\n");
87  }
88
89
90
```

Colas dinámicas: implementación con punteros VII

```
91  void main()
92  {
93      struct node *front  =NULL;
94      struct node *back   = NULL;
95      struct node *temp   =NULL;
96      int dato;
97      int value;
98      int op;
99      do
100     {   op=menu();
101         switch(op)
102         {
103             case 1:
104                 printf("Ingrese el dato a insertar\n");
105                 scanf("%d",&dato);
```

Colas dinámicas: implementación con punteros VIII

```
106                     push(&front,&back,dato);
107                     break;
108                 case 2:
109                     pop(&front,&back);
110                     break;
111                 case 3:
112                     print(front);
113                     break;
114             }
115         }while(op!=4);
116
117     }
```

Colas circulares (ring buffer): introducción I

Una cola circular es esencialmente una cola como las estudiadas anteriormente, con un tamaño o capacidad máxima que continuará volviendo sobre sí misma con un movimiento circular.

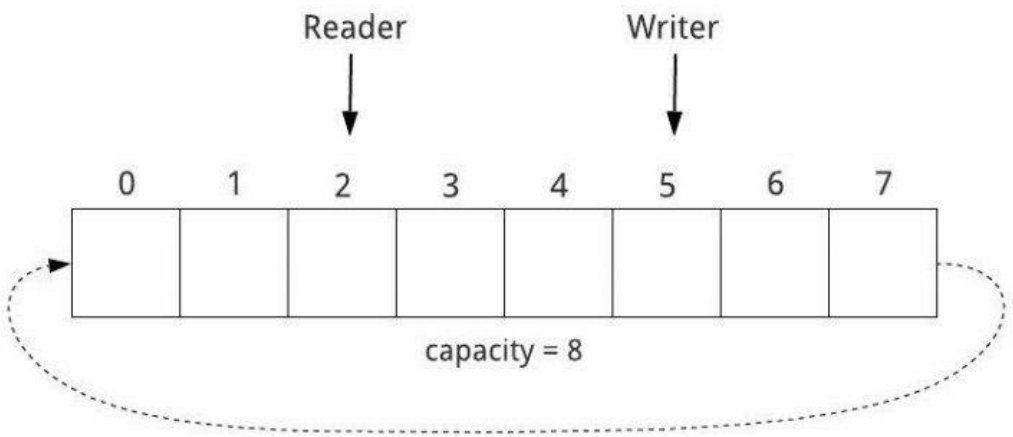


Figure: Representación gráfica de una cola circular

Colas circulares (ring buffer): introducción II

Ventajas del uso de colas circulares:

- Ofrecen una forma rápida y limpia de almacenar datos FIFO con un tamaño predeterminado
- Como tienen un tamaño fijo, no utilizan memoria dinámica y se evitan fugas de memoria
- Implementación simple
- Las operaciones ocurren en tiempo constante

Desventajas del uso de colas circulares:

- Sólo permiten el almacenamiento de un número fijo de elementos

¡Muchas gracias!

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar