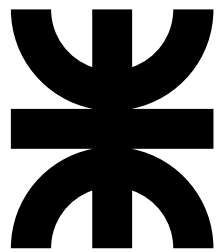
# Informática II

## Estructuras de datos

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

Estructuras dinámicas de datos (4):

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

Estructuras dinámicas de datos (4):

1. *Listas enlazadas:* colección de elementos de datos alineados en una fila. La inserción y eliminación se efectúa en cualquier parte.

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

Estructuras dinámicas de datos (4):

1. *Listas enlazadas:* colección de elementos de datos alineados en una fila.
   La inserción y eliminación se efectúa en cualquier parte.

2. *Pilas:* apilamiento de datos (LIFO).
   La inserción y eliminación se efectúa en un extremo (parte superior).

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

Estructuras dinámicas de datos (4):

1. *Listas enlazadas:* colección de elementos de datos alineados en una fila.
   La inserción y eliminación se efectúa en cualquier parte.

2. *Pilas:* apilamiento de datos (LIFO).
   La inserción y eliminación se efectúa en un extremo (parte superior).

3. *Colas:* representan listas de espera (FIFO).
   La inserción se efectúa en la parte trasera (cola), y la eliminación de la parte delantera (cabeza).

# Estructuras de datos
## Introducción

### Estructuras de datos

- Tamaño fijo $\longrightarrow$ arreglos (uno o múltiples subíndices) y `struct`
- Tamaño variable $\longrightarrow$ *estructuras dinámicas de datos*

Estructuras dinámicas de datos (4):

1. *Listas enlazadas:* colección de elementos de datos alineados en una fila.
   La inserción y eliminación se efectúa en cualquier parte.

2. *Pilas:* apilamiento de datos (LIFO).
   La inserción y eliminación se efectúa en un extremo (parte superior).

3. *Colas:* representan listas de espera (FIFO).
   La inserción se efectúa en la parte trasera (cola), y la eliminación de la parte delantera (cabeza).

4. *Árboles binarios:* facilita la búsqueda y clasificación de datos a alta velocidad, la eliminación eficiente de elementos duplicados de datos, etc.

# Estructuras de datos

## Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

# Estructuras de datos

## Estructuras auto-referenciadas

Estructura que contiene un miembro de puntero al mismo tipo de estructura

Ejemplo:

```
struct node {
  int data;
  struct node *next_ptr;
};
```

# Estructuras de datos

## Estructuras auto-referenciadas

**Estructura que contiene un miembro de puntero al mismo tipo de estructura**

Ejemplo:

```
struct node {
  int data;
  struct node *next_ptr;
};
```

Se define un tipo de estructura **`struct node`** con:

1. un miembro de dato entero **`data`**
2. un miembro de puntero al mismo tipo de estructura **`next_ptr`**

# Estructuras de datos

## Estructuras auto-referenciadas

**Estructura que contiene un miembro de puntero al mismo tipo de estructura**

Ejemplo:

```
struct node {
  int data;
  struct node *next_ptr;
};
```

Se define un tipo de estructura **`struct node`** con:

1. un miembro de dato entero **`data`**
2. un miembro de puntero al mismo tipo de estructura **`next_ptr`**

El miembro puntero (**`next_ptr`**) se conoce como *enlace* o *vínculo*, y permite vincular la estructura **`struct node`** a otra estructura.

# Estructuras de datos

## Estructuras auto-referenciadas

**Estructura que contiene un miembro de puntero al mismo tipo de estructura**

Ejemplo:

```
struct node {
  int data;
  struct node *next_ptr;
};
```

Se define un tipo de estructura **`struct node`** con:

1. un miembro de dato entero **`data`**
2. un miembro de puntero al mismo tipo de estructura **`next_ptr`**

El miembro puntero (**`next_ptr`**) se conoce como *enlace* o *vínculo*, y permite vincular la estructura **`struct node`** a otra estructura.

Se pueden enlazar varias estructuras auto-referenciadas para formar estructuras de datos útiles como: listas, pilas, colas, y árboles.

# Estructuras de datos

## Estructuras auto-referenciadas

**Estructura que contiene un miembro de puntero al mismo tipo de estructura**

Ejemplo:

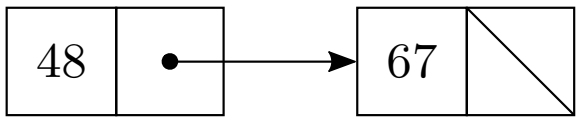
```
struct node {
  int data;
  struct node *next_ptr;
};
```

Se define un tipo de estructura **`struct node`** con:

1. un miembro de dato entero **`data`**
2. un miembro de puntero al mismo tipo de estructura **`next_ptr`**

El miembro puntero (**`next_ptr`**) se conoce como *enlace* o *vínculo*, y permite vincular la estructura **`struct node`** a otra estructura.

Se pueden enlazar varias estructuras auto-referenciadas para formar estructuras de datos útiles como: listas, pilas, colas, y árboles.



# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo
- Se pude acceder a los otros nodos mediante el puntero de enlace de cada nodo

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo
- Se pude acceder a los otros nodos mediante el puntero de enlace de cada nodo
- El final de la lista queda indicado por un puntero de enlace a `NULL`

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo
- Se pude acceder a los otros nodos mediante el puntero de enlace de cada nodo
- El final de la lista queda indicado por un puntero de enlace a `NULL`

Ventajas de las listas enlazadas con respecto a arreglos

- Adecuadas cuando no se sabe el nro de elementos a guardar en la estructura

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo
- Se pude acceder a los otros nodos mediante el puntero de enlace de cada nodo
- El final de la lista queda indicado por un puntero de enlace a `NULL`

Ventajas de las listas enlazadas con respecto a arreglos

- Adecuadas cuando no se sabe el nro de elementos a guardar en la estructura
- Son dinámicas, por lo que puede aumentar y disminuir su tamaño

# Estructuras de datos

## Listas enlazadas

Colección lineal de estructuras auto-referenciadas llamadas *nodos* conectados por enlaces de *punteros*.

- Se tiene acceso a la lista mediante un puntero al primer nodo
- Se pude acceder a los otros nodos mediante el puntero de enlace de cada nodo
- El final de la lista queda indicado por un puntero de enlace a `NULL`

Ventajas de las listas enlazadas con respecto a arreglos

- Adecuadas cuando no se sabe el nro de elementos a guardar en la estructura
- Son dinámicas, por lo que puede aumentar y disminuir su tamaño

Normalmente, los nodos de las listas enlazadas no están almacenados en memoria en forma contigua. Sin embargo, lógicamente, los nodos de una lista enlazada aparecen como contiguos.

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3

Ver ejemplo en funcionamiento (insertar `'a'`, `'m'`, `'p'`, `'e'`)

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3

Ver ejemplo en funcionamiento (insertar `'a'`, `'m'`, `'p'`, `'e'`)

```
The list is:
a --> e --> m --> p --> NULL
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3

Ver ejemplo en funcionamiento (insertar `'a'`, `'m'`, `'p'`, `'e'`)

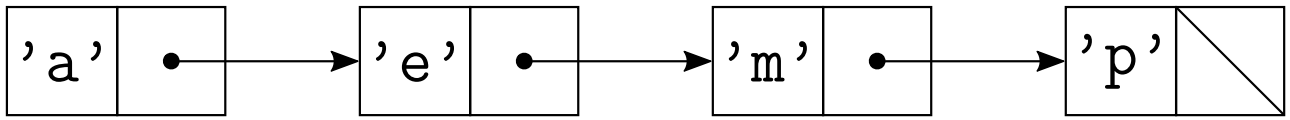
```
The list is:
a --> e --> m --> p --> NULL
```

Explicación del programa:

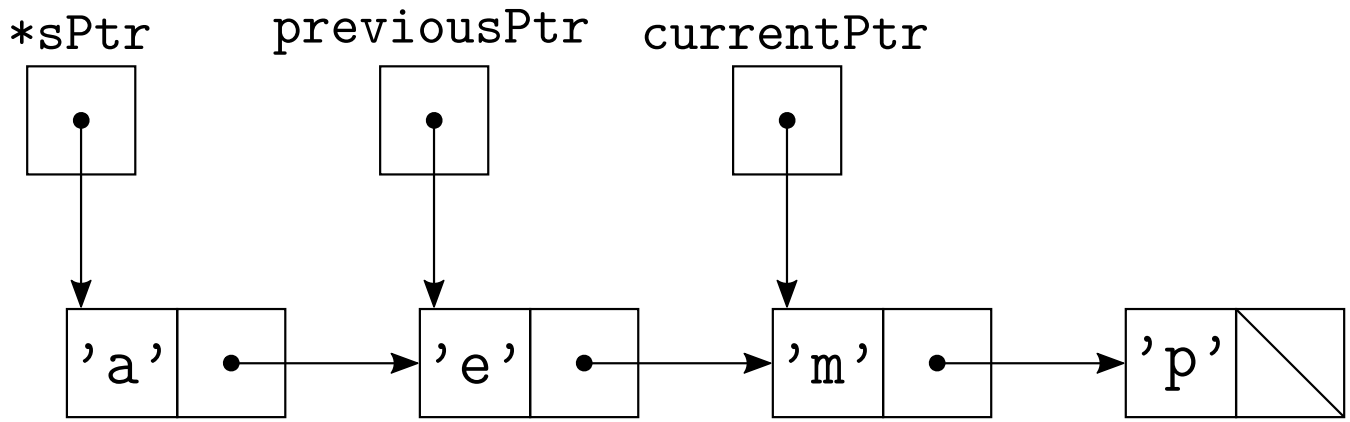
- Se insertan caracteres en la lista en orden alfabético
- La función `insert` recibe la *dirección* de la lista y el caracter a insertar
- La función `delete` recibe la *dirección* de la lista y el caracter a eliminar

# Estructuras de datos

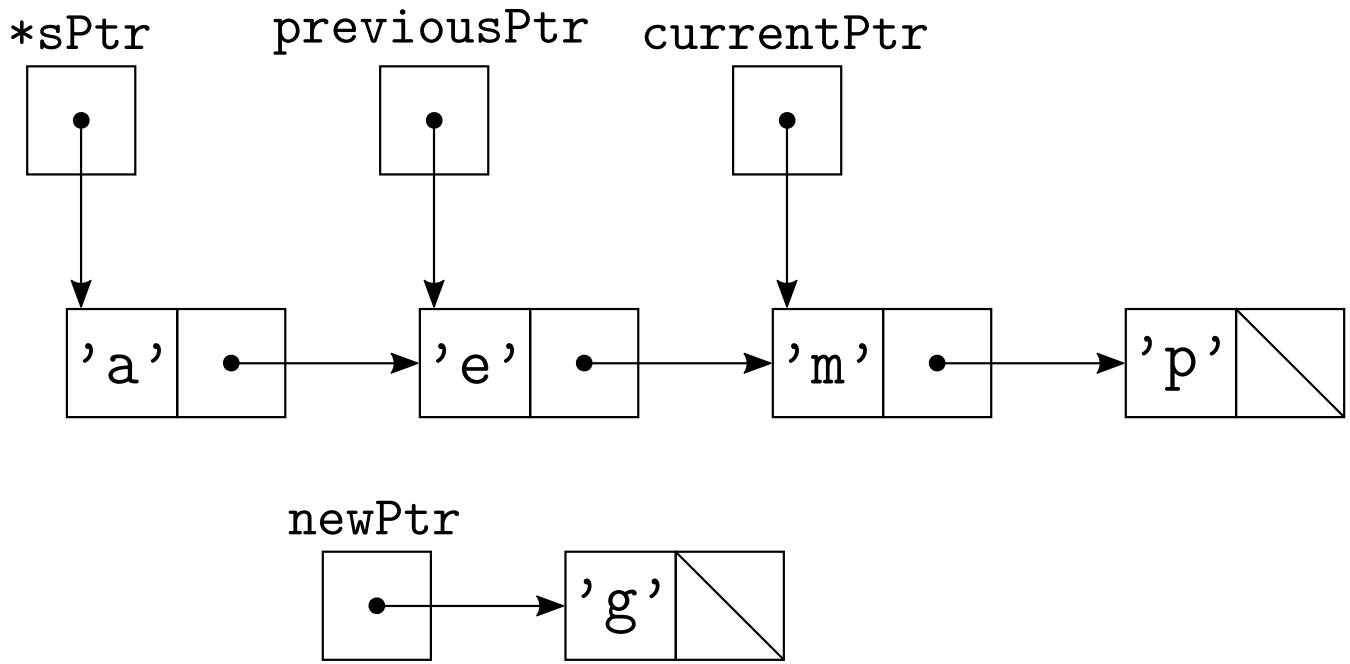
## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar



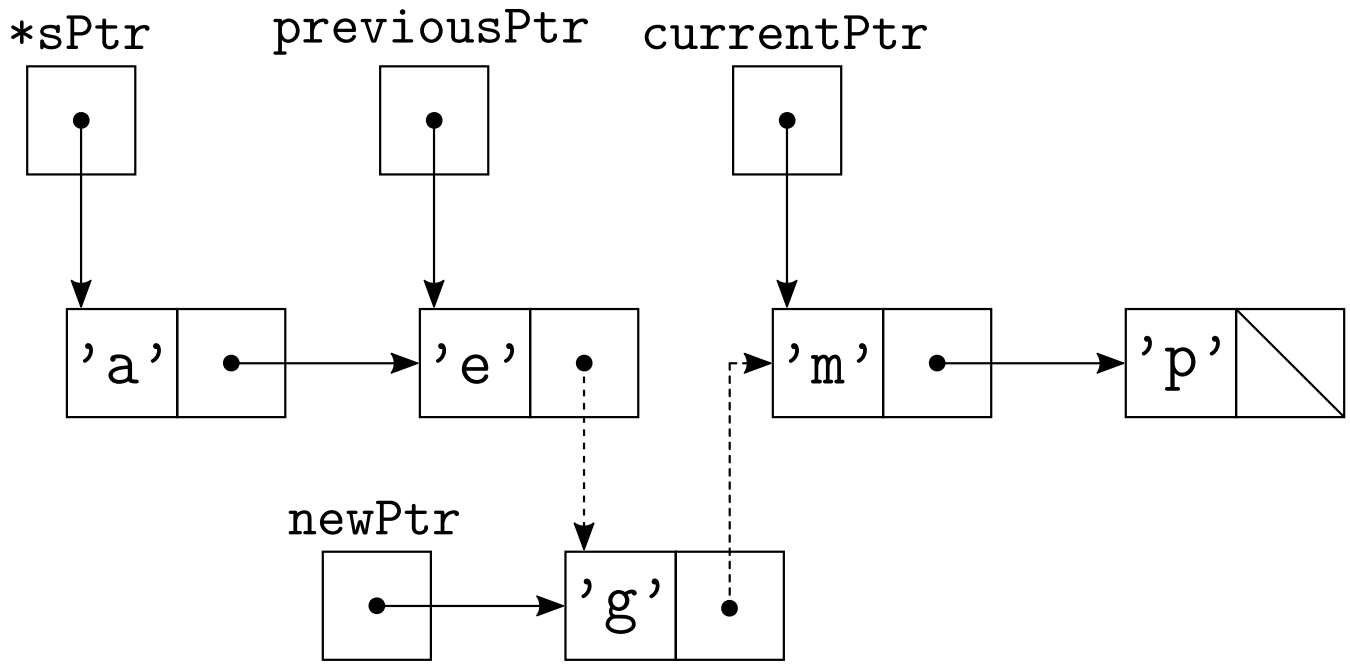
## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar



## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar



## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar



Dentro de la función `insert()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL` (`newPtr->nextPtr`)

Dentro de la función `insert()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL` (`newPtr->nextPtr`)
2. Inicializa `previousPtr` a `NULL` y `currentPtr` a `*sPtr`

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar

Dentro de la función `insert()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL` (`newPtr->nextPtr`)
2. Inicializa `previousPtr` a `NULL` y `currentPtr` a `*sPtr`
3. En tanto `currentPtr` no sea `NULL` y el valor a insertar sea mayor que `currentPtr->data` se van moviendo los punteros. Determina el punto de inserción en la lista.

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar

Dentro de la función `insert()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL` (`newPtr->nextPtr`)
2. Inicializa `previousPtr` a `NULL` y `currentPtr` a `*sPtr`
3. En tanto `currentPtr` no sea `NULL` y el valor a insertar sea mayor que `currentPtr->data` se van moviendo los punteros. Determina el punto de inserción en la lista.
4. Inserta en nuevo nodo
   - Si `previousPtr` es `NULL` el nuevo nodo es el primero de la lista

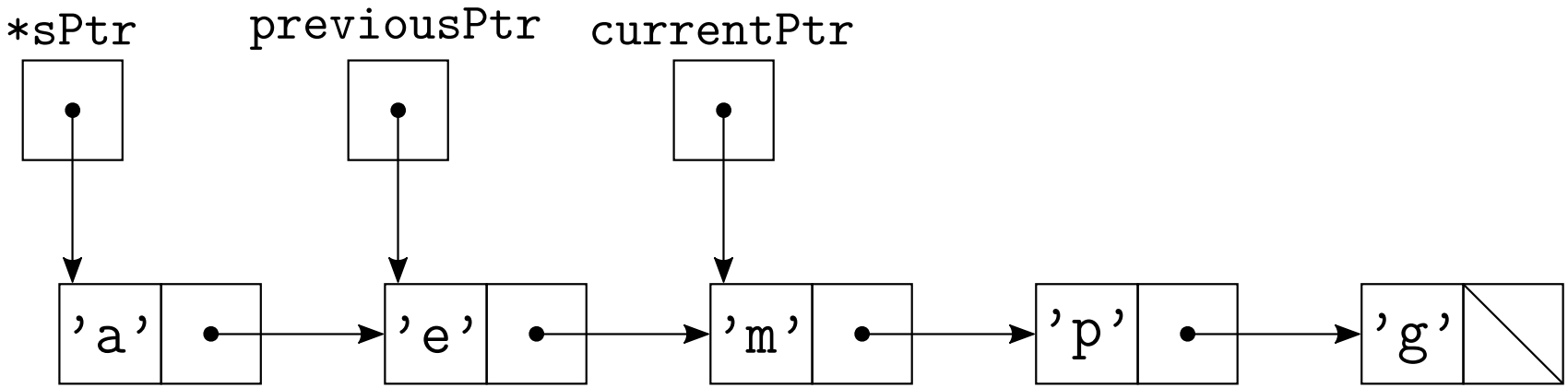
# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para insertar

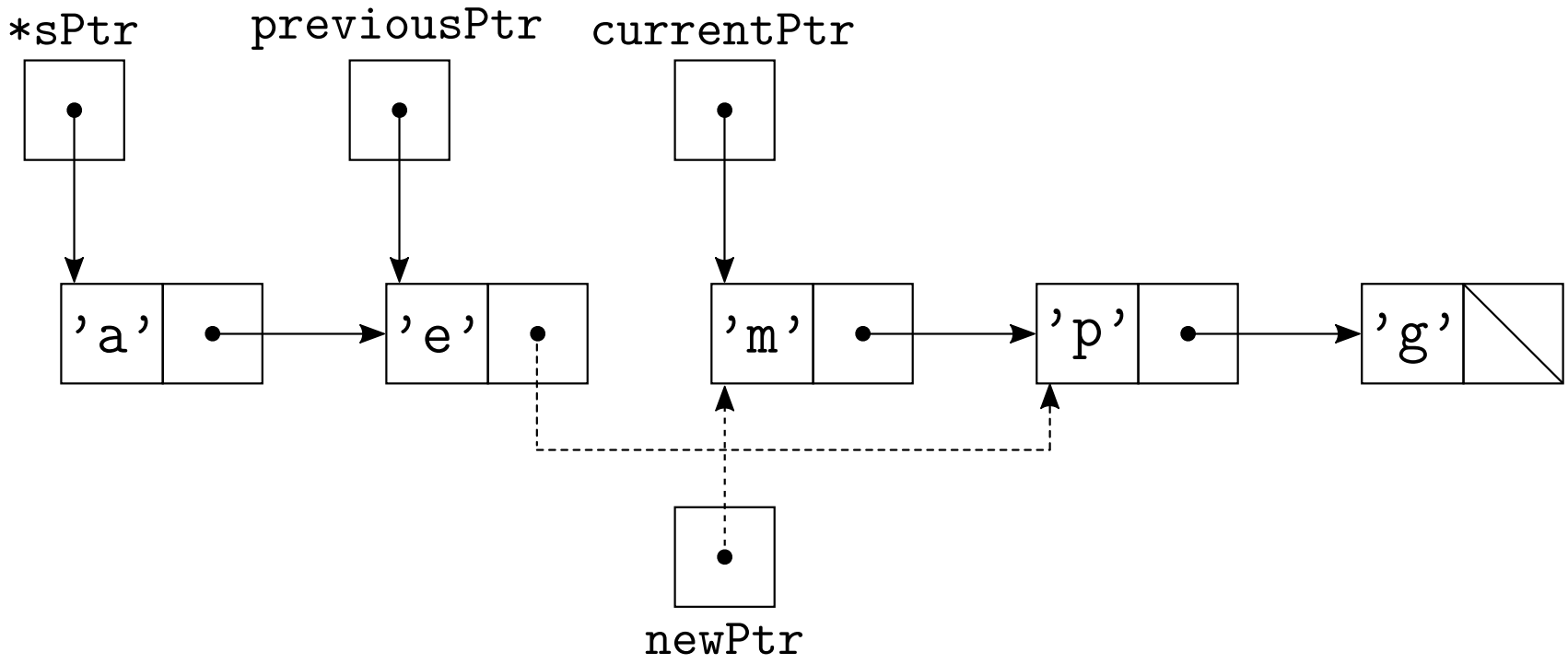
Dentro de la función `insert()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL` (`newPtr->nextPtr`)
2. Inicializa `previousPtr` a `NULL` y `currentPtr` a `*sPtr`
3. En tanto `currentPtr` no sea `NULL` y el valor a insertar sea mayor que `currentPtr->data` se van moviendo los punteros. Determina el punto de inserción en la lista.
4. Inserta en nuevo nodo
   - Si `previousPtr` es `NULL` el nuevo nodo es el primero de la lista
   - Si no, el nuevo nodo se inserta en su lugar, y ordena los punteros

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para eliminar



## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para eliminar



# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Pasos para eliminar

Dentro de la función `delete()`:

1. Si el caracter es el primero de la lista
   - Asigna `*sPtr` a `tempPtr`
   - Asigna `(*sPtr)->nextPtr` a `*sPtr`
   - Libera la memoria usando `tempPtr`
2. Si no, inicializa `previousPtr` con `*sPtr` y `currentPtr` con `(*sPtr)->newPtr`
3. Mientras `currentPtr` no sea `NULL` y el valor a borrar no sea `currentPtr->data` reasigna los punteros. Esto ubica el caracter a borrar.
4. Elimina el caracter
   - Si `currentPtr` no es `NULL`, asigna `currentPtr` a `tempPtr` y `currentPtr->nextPtr` a `previousPtr->nextPtr`, libera el nodo apuntado por `tempPtr`, y regresa el caracter borrado
   - Si `currentPtr` es `NULL` regresa el caracter nulo (`'\0'`) que indica que no se encontró el caracter a borrar

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente

```
1  /* Operating and maintaining a list */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct listNode { /* self-referenctial structure */
6    char data;
7    struct listNode *nextPtr;
8  };
9
10 typedef struct listNode ListNode;
11 typedef ListNode *ListNodePtr;
12
13 void insert(ListNodePtr * , char );
14 char delete(ListNodePtr * , char );
15 int isEmpty(ListNodePtr );
16 void printList(ListNodePtr );
17 void instructions(void);
18
19 int main(void)
20 {
21   ListNodePtr startPtr = NULL;
22   int choice;
23   char item;
24
25   instructions(); /* display the menu */
26   printf("? ");
27   scanf("%d", &choice);
28
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
29    while(choice != 3)
30    {
31      switch(choice)
32      {
33        case 1:
34          printf("Enter␣a␣character:␣");
35          scanf("\n%c", &item);
36          insert(&startPtr, item);
37          printList(startPtr);
38          break;
39        case 2:
40          if( !isEmpty(startPtr) )
41          {
42            printf("Enter␣character␣to␣be␣deleted:␣");
43            scanf("\n%c", &item);
44
45            if( delete(&startPtr, item) )
46            {
47              printf("%c␣deleted.\n", item);
48              printList(startPtr);
49            }
50            else
51              printf("%c␣not␣found.\n\n", item);
52          }
53          else
54            printf("List␣is␣empty.\n\n");
55          break;
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
56        default:
57          printf("Invalid␣choice.\n\n");
58          instructions();
59          break;
60      }
61
62      printf("?␣");
63      scanf("%d", &choice);
64    }
65
66    printf("End␣of␣run.\n");
67    return 0;
68  }
69
70  /* Print the instructions */
71  void instructions(void)
72  {
73    printf("Enter␣your␣choice:\n"
74           "␣␣␣1␣to␣insert␣an␣element␣into␣the␣list.\n"
75           "␣␣␣2␣to␣delete␣an␣element␣from␣the␣list.\n"
76           "␣␣␣3␣to␣end.\n");
77  }
78
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
79 /* Insert a new value into the list in sorted order */
80 void insert(ListNodePtr *sPtr, char value)
81 {
82   ListNodePtr newPtr, previousPtr, currentPtr;
83
84   newPtr = malloc(sizeof(ListNode));
85
86   if(newPtr != NULL)
87   {
88     newPtr->data = value;
89     newPtr->nextPtr = NULL;
90
91     previousPtr = NULL;
92     currentPtr = *sPtr;
93
94     while( currentPtr != NULL && value > currentPtr->data )
95     {
96       previousPtr = currentPtr; /* walk to... */
97       currentPtr = currentPtr->nextPtr; /* ... next node */
98     }
99
100    if(previousPtr == NULL)
101    {
102      newPtr->nextPtr = *sPtr;
103      *sPtr = newPtr;
104    }
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
105     else
106     {
107       previousPtr->nextPtr = newPtr;
108       newPtr->nextPtr = currentPtr;
109     }
110   }
111   else
112     printf("%c␣not␣inserted.␣No␣memory␣available.\n", value);
113 }
114
115 /* Delete a list element */
116 char delete(ListNodePtr *sPtr, char value)
117 {
118   ListNodePtr previousPtr, currentPtr, tempPtr;
119
120   if(value == (*sPtr)->data)
121   {
122     tempPtr = *sPtr;
123     *sPtr = (*sPtr)->nextPtr; /* de-thread the node */
124     free(tempPtr); /* free the de-threaded node */
125     return value;
126   }
127   else
128   {
129     previousPtr = *sPtr;
130     currentPtr = (*sPtr)->nextPtr;
131
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
131
132     while(currentPtr != NULL && currentPtr->data != value)
133     {
134       previousPtr = currentPtr; /* walk to ... */
135       currentPtr = currentPtr->nextPtr; /* ... next node */
136     }
137
138     if(currentPtr != NULL)
139     {
140       tempPtr = currentPtr;
141       previousPtr->nextPtr = currentPtr->nextPtr;
142       free(tempPtr);
143       return value;
144     }
145   }
146
147   return '\0';
148 }
149
```

# Estructuras de datos

## Listas enlazadas – ejemplo D&D Fig. 12.3 – Código fuente (cont.)

```
150 /* Return 1 if the list is empty, 0 otherwise */
151 int isEmpty(ListNodePtr sPtr)
152 {
153   return sPtr == NULL;
154 }
155
156 /* Print the list */
157 void printList(ListNodePtr currentPtr)
158 {
159   if(currentPtr == NULL)
160     printf("List␣is␣empty.\n\n");
161   else
162   {
163     printf("The␣list␣is:\n");
164
165     while(currentPtr != NULL)
166     {
167       printf("␣%c␣-->␣", currentPtr->data);
168       currentPtr = currentPtr->nextPtr;
169     }
170
171     printf("NULL\n\n");
172   }
173 }
```

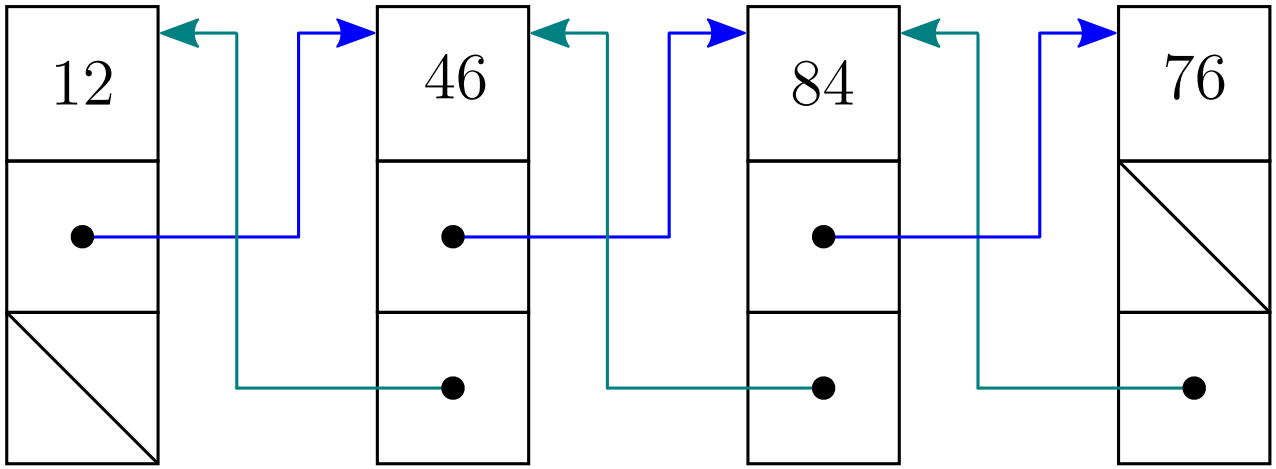
- Las listas anteriores se denominas *listas enlazadas simples*

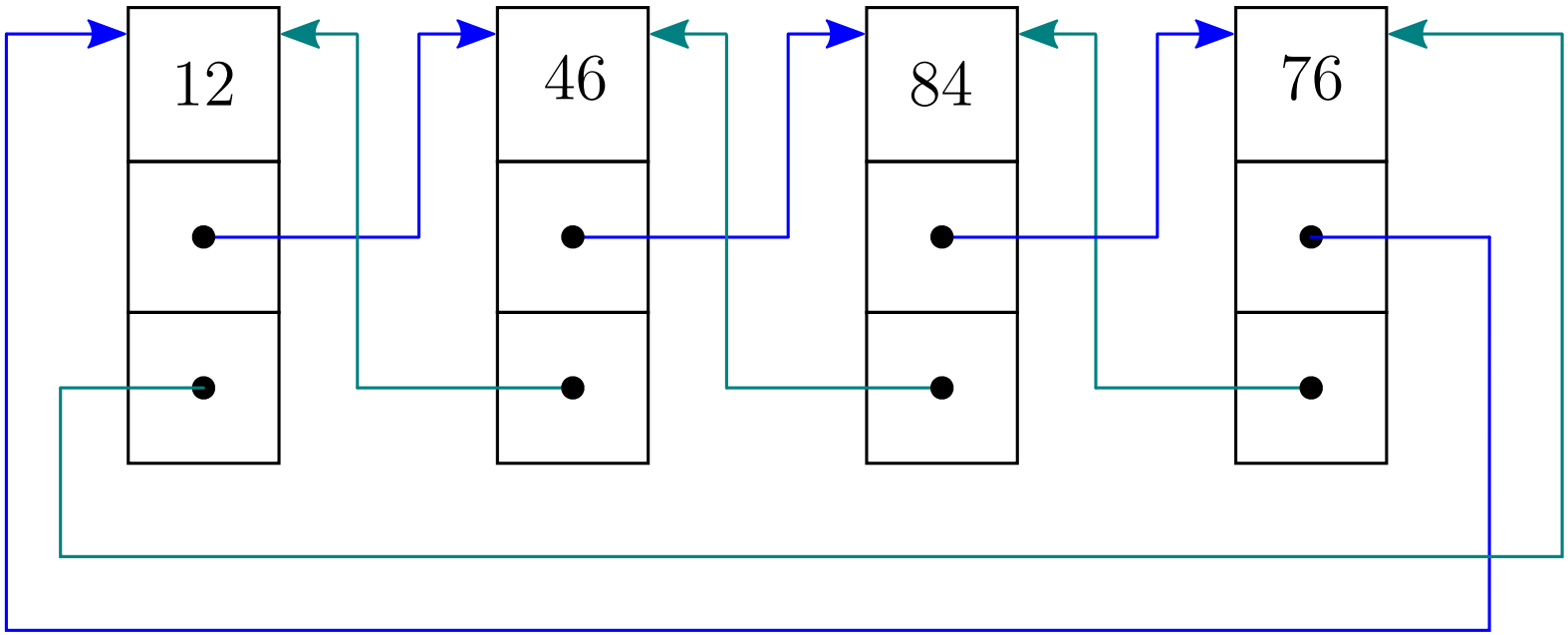
- Las listas anteriores se denominas *listas enlazadas simples*
- También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior

- Las listas anteriores se denominas *listas enlazadas simples*
- También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- Una lista doblemente enlazada se puede recorrer en ambos sentidos

- Las listas anteriores se denominas *listas enlazadas simples*
- También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- Una lista doblemente enlazada se puede recorrer en ambos sentidos
- Si los extremos se apuntan entre sí se tiene un *anillo lógico*

- Las listas anteriores se denominas *listas enlazadas simples*
- También existen *listas doblemente enlazadas* en las que cada elemento tiene dos punteros, uno al elemento anterior y otro al posterior
- Una lista doblemente enlazada se puede recorrer en ambos sentidos
- Si los extremos se apuntan entre sí se tiene un *anillo lógico*
- Una lista de anillo lógico se puede recorrer de forma circular en ambos sentidos


12
46
84
76


12
46
84
76

# Estructuras de datos

## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

# Estructuras de datos

## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- Se referencia a una pila mediante un puntero al elemento superior de la misma

# Estructuras de datos

## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- Se referencia a una pila mediante un puntero al elemento superior de la misma
- Hay que definir a `NULL` el mimbro de enlace del último nodo para indicar que se trata de la parte inferior del a pila

# Estructuras de datos

## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- Se referencia a una pila mediante un puntero al elemento superior de la misma
- Hay que definir a `NULL` el mimbro de enlace del último nodo para indicar que se trata de la parte inferior del a pila

Funciones básicas para manipular una pila:

- `push()`: crea un nodo (asigna espacio en memoria), almacena el nuevo dato y lo coloca en la parte superior de la pila

# Estructuras de datos

## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- Se referencia a una pila mediante un puntero al elemento superior de la misma
- Hay que definir a `NULL` el mimbro de enlace del último nodo para indicar que se trata de la parte inferior del a pila

Funciones básicas para manipular una pila:

- `push()`: crea un nodo (asigna espacio en memoria), almacena el nuevo dato y lo coloca en la parte superior de la pila
- `pop()`: elimina un nodo de la parte superior, libera el espacio de memoria, y devuelve el dato obtenido

# Estructuras de datos

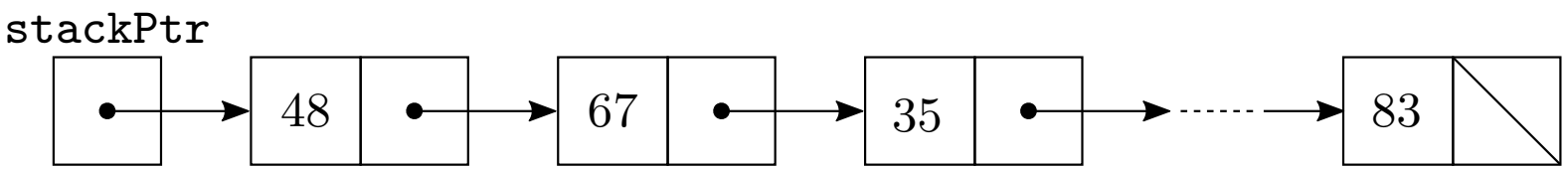
## Pilas

Se la conoce como una estructura de datos *último en entrar, primero en salir* (LIFO, last-in, first-out). Puede interpretarse como una versión restringida de una lista enlazada.

- Se referencia a una pila mediante un puntero al elemento superior de la misma
- Hay que definir a `NULL` el mimbro de enlace del último nodo para indicar que se trata de la parte inferior del a pila

Funciones básicas para manipular una pila:

- `push()`: crea un nodo (asigna espacio en memoria), almacena el nuevo dato y lo coloca en la parte superior de la pila
- `pop()`: elimina un nodo de la parte superior, libera el espacio de memoria, y devuelve el dato obtenido



# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8

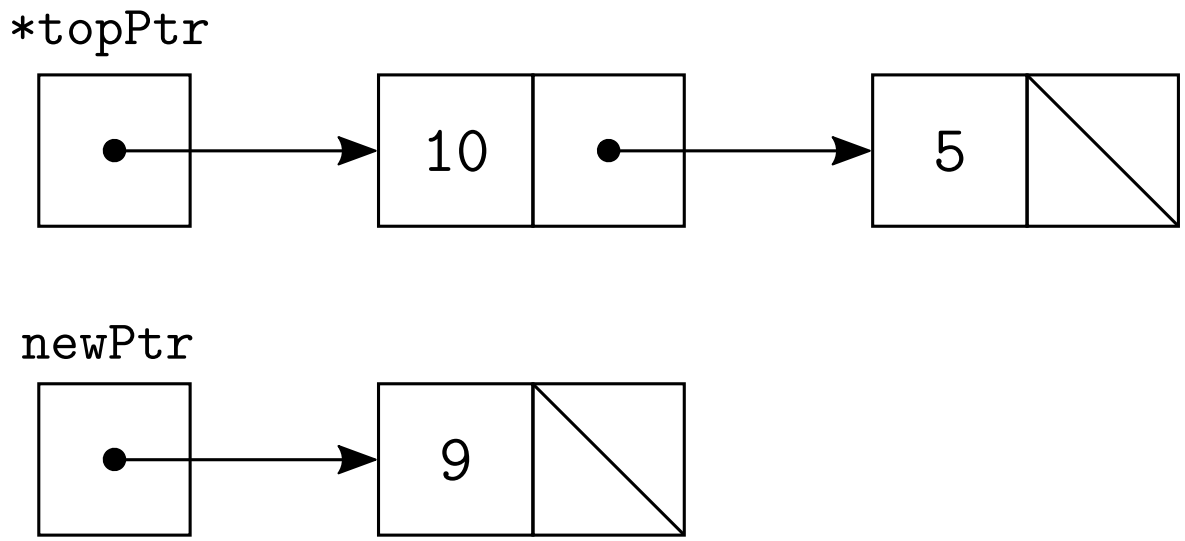
Ver ejemplo en funcionamiento

# Estructuras de datos

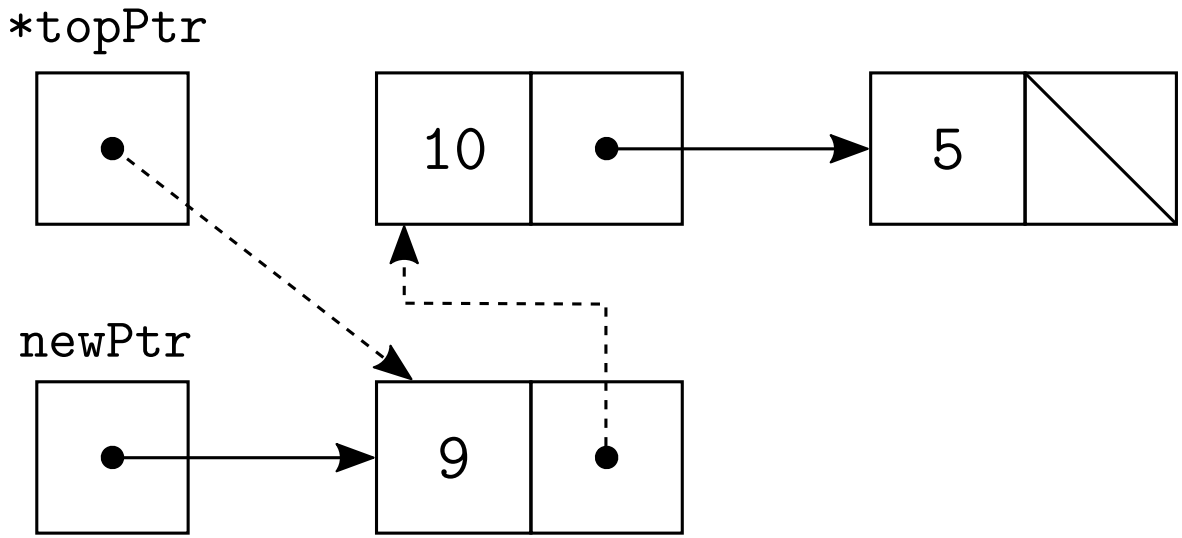
## Pilas – ejemplo D&D Fig. 12.8

Ver ejemplo en funcionamiento

```
Enter choice:
   1 to push a value on the stack
   2 to pop a value off the stack
   3 to end program
. . .
. . .
? 1
Enter an integer: 4
The stack is:
4 --> 9 --> 5 --> 10 --> NULL

? 2
The popped value is 4.
The stack is:
9 --> 5 --> 10 --> NULL
```


*topPtr
10
5
newPtr
9

## Pilas – ejemplo D&D Fig. 12.8 – Pasos para insertar



Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL`

Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*topPtr` a `newPtr->nextPtr`. El nuevo enlace ahora apunta al nodo superior de la pila.

Dentro de la función `push()`:

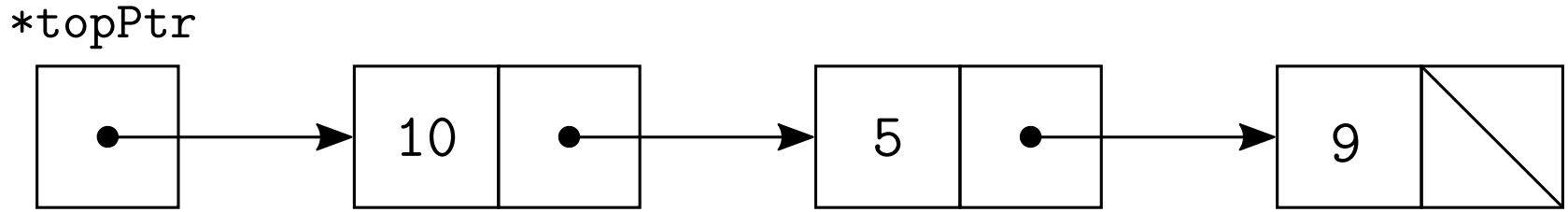
1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*topPtr` a `newPtr->nextPtr`. El nuevo enlace ahora apunta al nodo superior de la pila.
3. Asigna `newPtr` a `*topPtr` – `*topPtr` apunta ahora a la nueva parte superior de la pila.

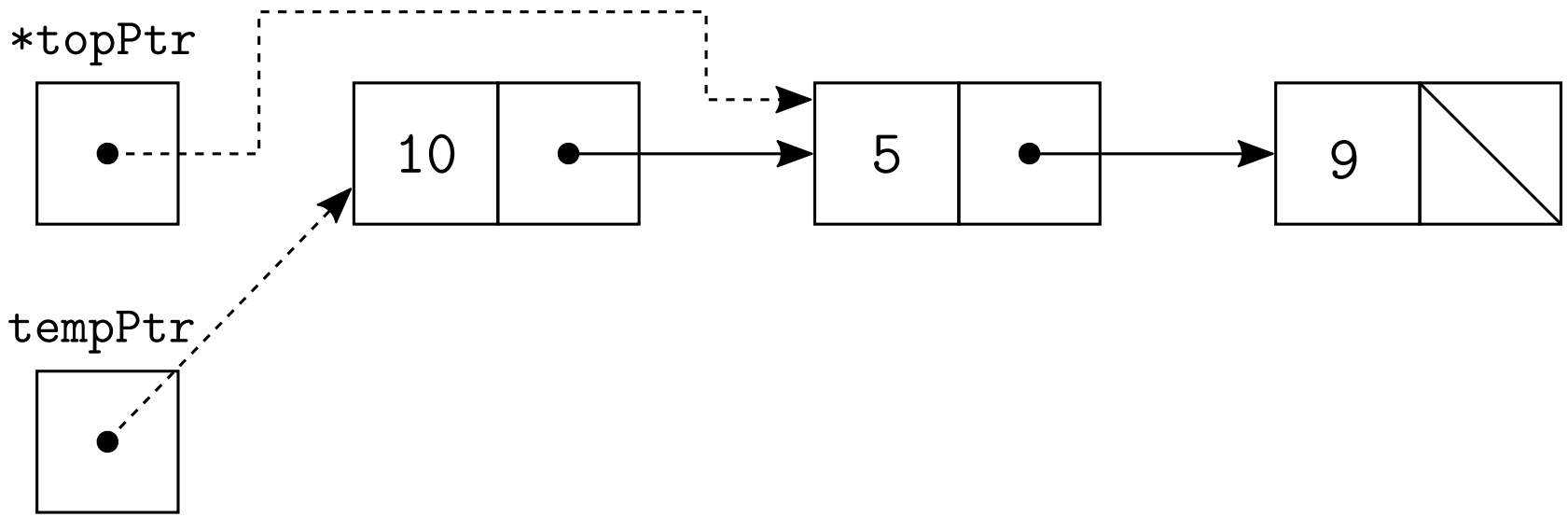
Dentro de la función `push()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, almacena el nuevo dato en `newPtr->data`, y pone el enlace a `NULL`
2. Asigna el puntero de la pila `*topPtr` a `newPtr->nextPtr`. El nuevo enlace ahora apunta al nodo superior de la pila.
3. Asigna `newPtr` a `*topPtr` – `*topPtr` apunta ahora a la nueva parte superior de la pila.

Las modificaciones realizadas a `*topPtr` modifican el valor de `stackPtr` en `main()`.

## Pilas – ejemplo D&D Fig. 12.8 – Pasos para eliminar




*topPtr
10
5
9
tempPtr

Dentro de la función `pop()`:

1. Asignar `*topPtr` a `tempPtr` (para luego liberar memoria)

Dentro de la función `pop()`:

1. Asignar `*topPtr` a `tempPtr` (para luego liberar memoria)
2. Asignar `(*topPtr)->data` a `popValue` (guarda el valor del nodo)

Dentro de la función `pop()`:

1. Asignar `*topPtr` a `tempPtr` (para luego liberar memoria)
2. Asignar `(*topPtr)->data` a `popValue` (guarda el valor del nodo)
3. Asignar `(*topPtr)->nextPtr` a `*topPtr` (nuevo nodo superior)

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Pasos para eliminar

Dentro de la función `pop()`:

1. Asignar `*topPtr` a `tempPtr` (para luego liberar memoria)
2. Asignar `(*topPtr)->data` a `popValue` (guarda el valor del nodo)
3. Asignar `(*topPtr)->nextPtr` a `*topPtr` (nuevo nodo superior)
4. Liberar la memoria apuntada por `tempPtr` con `free()`

Dentro de la función `pop()`:

1. Asignar `*topPtr` a `tempPtr` (para luego liberar memoria)
2. Asignar `(*topPtr)->data` a `popValue` (guarda el valor del nodo)
3. Asignar `(*topPtr)->nextPtr` a `*topPtr` (nuevo nodo superior)
4. Liberar la memoria apuntada por `tempPtr` con `free()`
5. Regresar `popValue` a la función llamadora (en este caso `main()`)

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Código fuente

```c
/* Dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

struct stackNode { /* self-referential structure */
  int data;
  struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;

void push(StackNodePtr * , int );
int pop(StackNodePtr * );
int isEmpty(StackNodePtr );
void printStack(StackNodePtr );
void instructions(void);

int main(void)
{
  StackNodePtr stackPtr = NULL; /* points to the stack top */
  int choice, value;

  instructions();
  printf("? ");
  scanf("%d", &choice);

```

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Código fuente (cont.)

```
28    while(choice != 3)
29    {
30      switch(choice)
31      {
32        case 1: /* push value onto stack */
33          printf("Enter␣an␣integer:␣");
34          scanf("%d", &value);
35          push(&stackPtr, value);
36          printStack(stackPtr);
37          break;
38        case 2: /* pop value off stack */
39          if( !isEmpty(stackPtr) )
40            printf("The␣popped␣value␣is␣%d.\n", pop(&stackPtr));
41          printStack(stackPtr);
42          break;
43        default:
44          printf("Invalid␣choice.\n\n");
45          instructions();
46          break;
47      }
48
49      printf("?␣");
50      scanf("%d", &choice);
51    }
52
53    printf("End␣of␣run.\n");
54    return 0;
55  }
```

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Código fuente (cont.)

```
56
57 /* Print the instructions */
58 void instructions(void)
59 {
60   printf("Enter␣choice:\n"
61          "1␣to␣push␣a␣value␣on␣the␣stack\n"
62          "2␣to␣pop␣a␣value␣off␣the␣stack\n"
63          "3␣to␣end␣program\n");
64 }
65
66 /* Insert a node at the stack top */
67 void push(StackNodePtr *topPtr, int info)
68 {
69   StackNodePtr newPtr;
70
71   newPtr = malloc(sizeof(StackNode));
72   if(newPtr != NULL)
73   {
74     newPtr->data = info;
75     newPtr->nextPtr = *topPtr;
76     *topPtr = newPtr;
77   }
78   else
79     printf("%d␣not␣inserted.␣No␣memory␣available.\n", info);
80 }
81
```

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Código fuente (cont.)

```
82  /* Remove a node from the stack top */
83 int pop(StackNodePtr *topPtr)
84 {
85   StackNodePtr tempPtr;
86   int popValue;
87
88   tempPtr = *topPtr;
89   popValue = (*topPtr)->data;
90   *topPtr = (*topPtr)->nextPtr;
91   free(tempPtr);
92   return popValue;
93 }
94
```

# Estructuras de datos

## Pilas – ejemplo D&D Fig. 12.8 – Código fuente (cont.)

```
95  /* Print the stack */
96  void printStack(StackNodePtr currentPtr)
97  {
98    if(currentPtr == NULL)
99      printf("The␣stack␣is␣empty.\n\n");
100   else
101   {
102     printf("The␣stack␣is:\n");
103
104     while(currentPtr != NULL)
105     {
106       printf("␣%d␣-->␣", currentPtr->data);
107       currentPtr = currentPtr->nextPtr;
108     }
109
110     printf("NULL\n\n");
111   }
112 }
113
114 /* Is the stack empty? */
115 int isEmpty(StackNodePtr topPtr)
116 {
117   return topPtr == NULL;
118 }
```

# Estructuras de datos

## Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

# Estructuras de datos
## Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,

# Estructuras de datos
## Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- y son incluidos o insertos únicamente en la *parte trasera* de la cola.

# Estructuras de datos
## Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- y son incluidos o insertos únicamente en la *parte trasera* de la cola.

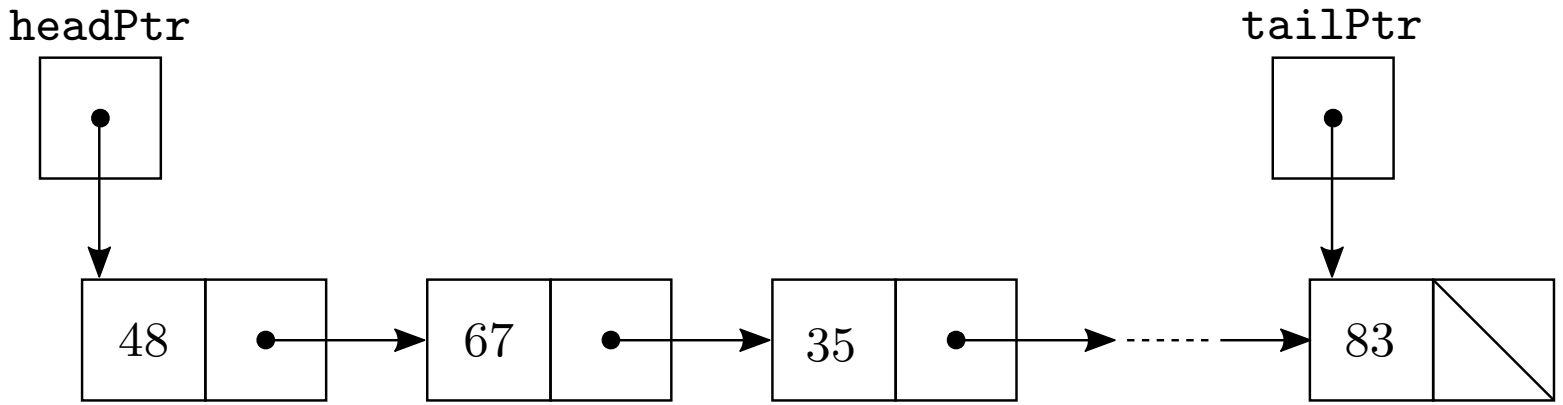
Las operaciones de insertar y retirar se conoce como `enqueue` y `dequeue`.

# Estructuras de datos
## Colas

Se la conoce como una estructura de datos *primero en entrar, primero en salir* (FIFO, first-in, first-out).

- Se eliminan nodos de la cola solo de la parte delantera o *cabeza* de la cola,
- y son incluidos o insertos únicamente en la *parte trasera* de la cola.

Las operaciones de insertar y retirar se conoce como `enqueue` y `dequeue`.



# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13

Ver ejemplo en funcionamiento

# Estructuras de datos

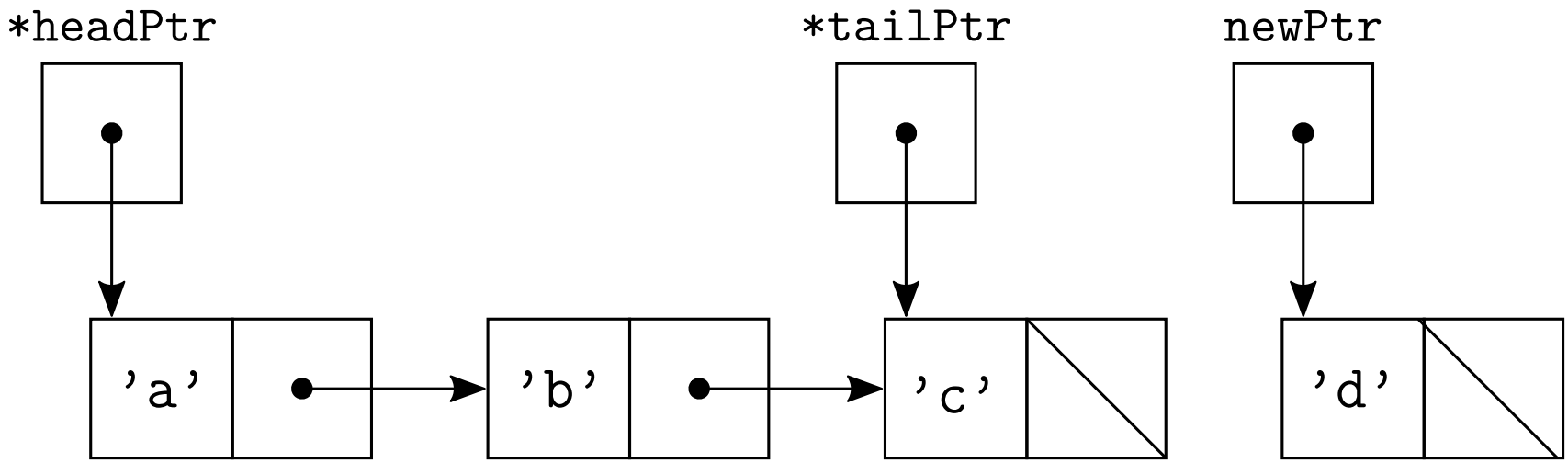
## Colas – ejemplo D&D Fig. 12.13

Ver ejemplo en funcionamiento

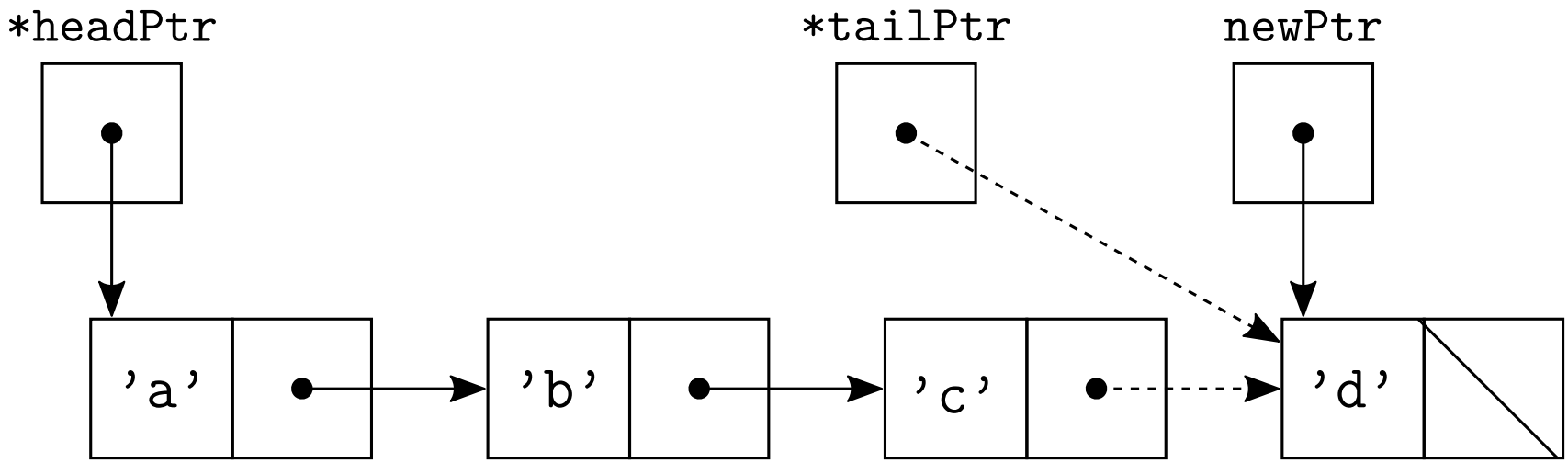
```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
. . .
. . .
? 1
Enter a character: c
The queue is:
a --> b --> c --> NULL

? 2
a has been dequeue.
The queue is:
b --> c --> NULL
```

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para insertar



## Colas – ejemplo D&D Fig. 12.13 – Pasos para insertar



# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para insertar

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, guarda el valor a insertar en la cola `newPtr->data`, y el enlace `newPtr->nextPtr` a `NULL`

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para insertar

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, guarda el valor a insertar en la cola `newPtr->data`, y el enlace `newPtr->nextPtr` a `NULL`
2. Si la cola está vacía, asigna `newPtr` a `*headPtr`; si no, asigna `newPtr` a `(*tailPtr)->nextPtr`

# Estructuras de datos

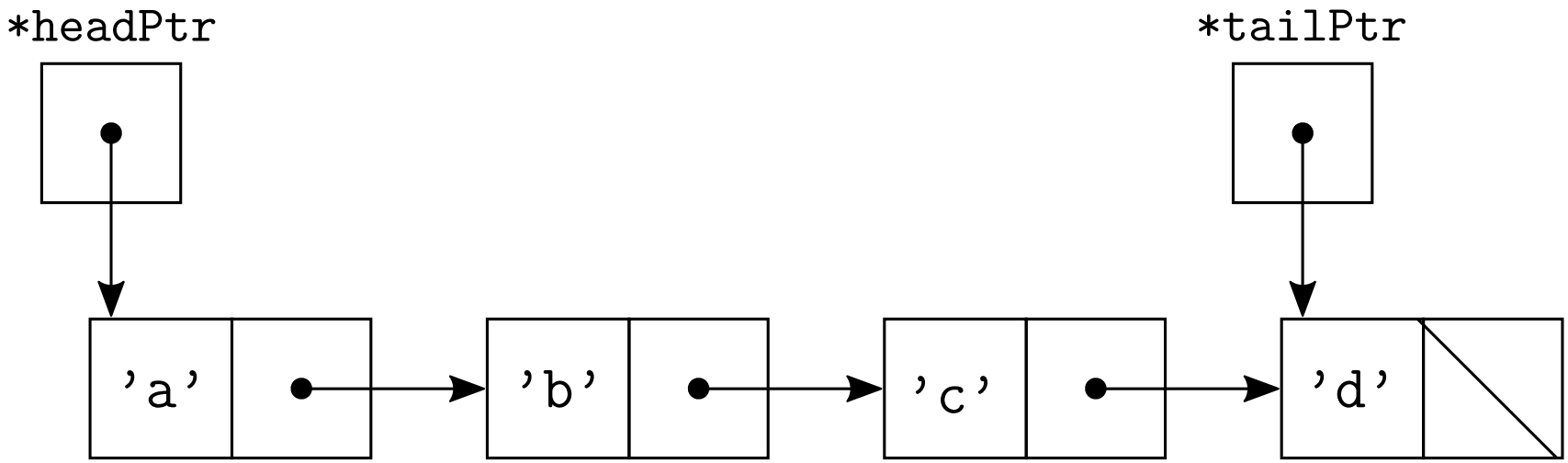
## Colas – ejemplo D&D Fig. 12.13 – Pasos para insertar

Dentro de la función `enqueue()`:

1. Crea un nuevo nodo con `malloc()` apuntado por `newPtr`, guarda el valor a insertar en la cola `newPtr->data`, y el enlace `newPtr->nextPtr` a `NULL`
2. Si la cola está vacía, asigna `newPtr` a `*headPtr`; si no, asigna `newPtr` a `(*tailPtr)->nextPtr`
3. Asigna `newPtr` a `*tailPtr`

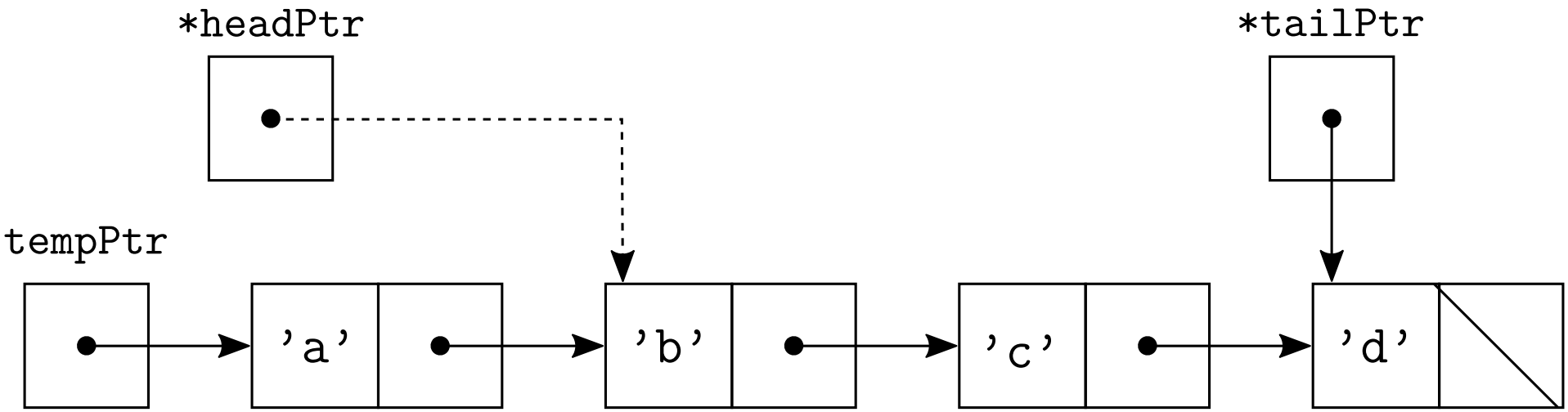
# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar



# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar



Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)

Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)
2. Asigna `*headPtr` a `tempPtr` (para luego liberar memoria)

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar

Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)
2. Asigna `*headPtr` a `tempPtr` (para luego liberar memoria)
3. Asigna `(*headPtr)->nextPtr` a `*headPtr` (apunta al primer nodo de la cola)

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar

Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)
2. Asigna `*headPtr` a `tempPtr` (para luego liberar memoria)
3. Asigna `(*headPtr)->nextPtr` a `*headPtr` (apunta al primer nodo de la cola)
4. Si `*headPtr` es `NULL` asigna `NULL` a `*tailPtr`

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar

Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)
2. Asigna `*headPtr` a `tempPtr` (para luego liberar memoria)
3. Asigna `(*headPtr)->nextPtr` a `*headPtr` (apunta al primer nodo de la cola)
4. Si `*headPtr` es `NULL` asigna `NULL` a `*tailPtr`
5. Libera la memoria apuntada por `tempPtr` con `free()`

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Pasos para eliminar

Dentro de la función `dequeue()`:

1. Asigna `(*headPtr)->data` a `value` (valor del nodo)
2. Asigna `*headPtr` a `tempPtr` (para luego liberar memoria)
3. Asigna `(*headPtr)->nextPtr` a `*headPtr` (apunta al primer nodo de la cola)
4. Si `*headPtr` es `NULL` asigna `NULL` a `*tailPtr`
5. Libera la memoria apuntada por `tempPtr` con `free()`
6. Regresa `value` a la función llamadora (en este caso `main()`)

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Código fuente

```c
/* Operating and maintaining a queue */
#include <stdio.h>
#include <stdlib.h>

struct queueNode { /* self-referential structure */
  char data;
  struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function protptypes */
void printQueue(QueueNodePtr );
int isEmpty(QueueNodePtr );
char dequeue(QueueNodePtr * , QueueNodePtr * );
void enqueue(QueueNodePtr * , QueueNodePtr * , char );
void instructions(void);

int main(void)
{
  QueueNodePtr headPtr = NULL, tailPtr = NULL;
  int choice;
  char item;

  instructions();
  printf("? ");
  scanf("%d", &choice);
```

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Código fuente (cont.)

```
29
30    while(choice != 3)
31    {
32      switch(choice)
33      {
34        case 1:
35          printf("Enter␣a␣character:␣");
36          scanf("\n%c", &item);
37          enqueue(&headPtr, &tailPtr, item);
38          printQueue(headPtr);
39          break;
40        case 2:
41          if( !isEmpty(headPtr) )
42          {
43            item = dequeue(&headPtr, &tailPtr);
44            printf("%c␣has␣been␣dequeue.\n", item);
45          }
46          printQueue(headPtr);
47          break;
48        default:
49          printf("Invalid␣choice.\n\n");
50          instructions();
51          break;
52      }
53
```

```
54      printf("? ");
55      scanf("%d", &choice);
56    }
57
58    printf("End of run.\n");
59    return 0;
60  }
61
62  void instructions(void)
63  {
64    printf("Enter your choice:\n"
65           "   1 to add an item to the queue\n"
66           "   2 to remove an item from the queue\n"
67           "   3 to end\n");
68  }
69
```

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Código fuente (cont.)

```
70 void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value)
71 {
72   QueueNodePtr newPtr;
73
74   newPtr = malloc(sizeof(QueueNodePtr));
75
76   if(newPtr != NULL)
77   {
78     newPtr->data = value;
79     newPtr->nextPtr = NULL;
80
81     if( isEmpty(*headPtr) )
82       *headPtr = newPtr;
83     else
84       (*tailPtr)->nextPtr = newPtr;
85
86     *tailPtr = newPtr;
87   }
88   else
89     printf("%c not inserted. No memory available.\n", value);
90 }
91
```

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Código fuente (cont.)

```
92  char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr)
93  {
94    char value;
95    QueueNodePtr tempPtr;
96
97    value = (*headPtr)->data;
98    tempPtr = *headPtr;
99    *headPtr = (*headPtr)->nextPtr;
100
101   if(*headPtr == NULL)
102     *tailPtr = NULL;
103
104   free(tempPtr);
105   return value;
106 }
107
108 int isEmpty(QueueNodePtr headPtr)
109 {
110   return headPtr == NULL;
111 }
112
```

# Estructuras de datos

## Colas – ejemplo D&D Fig. 12.13 – Código fuente (cont.)

```
113 void printQueue(QueueNodePtr currentPtr)
114 {
115   if(currentPtr == NULL)
116     printf("Queue␣is␣empty.\n\n");
117   else
118   {
119     printf("The␣queue␣is:\n");
120 
121     while(currentPtr != NULL)
122     {
123       printf("␣%c␣-->␣", currentPtr->data);
124       currentPtr = currentPtr->nextPtr;
125     }
126 
127     printf("NULL\n\n");
128   }
129 }
130 
```

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)

# Estructuras de datos
## Árboles binarios

### Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

- El *nodo raíz* es el primer nodo del árbol

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

- El *nodo raíz* es el primer nodo del árbol
- Cada enlace en el nodo raíz apunta a un hijo

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

- El *nodo raíz* es el primer nodo del árbol
- Cada enlace en el nodo raíz apunta a un hijo
- El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

- El *nodo raíz* es el primer nodo del árbol
- Cada enlace en el nodo raíz apunta a un hijo
- El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- El *hijo derecho* es el primer nodo del *sub-árbol derecho*

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

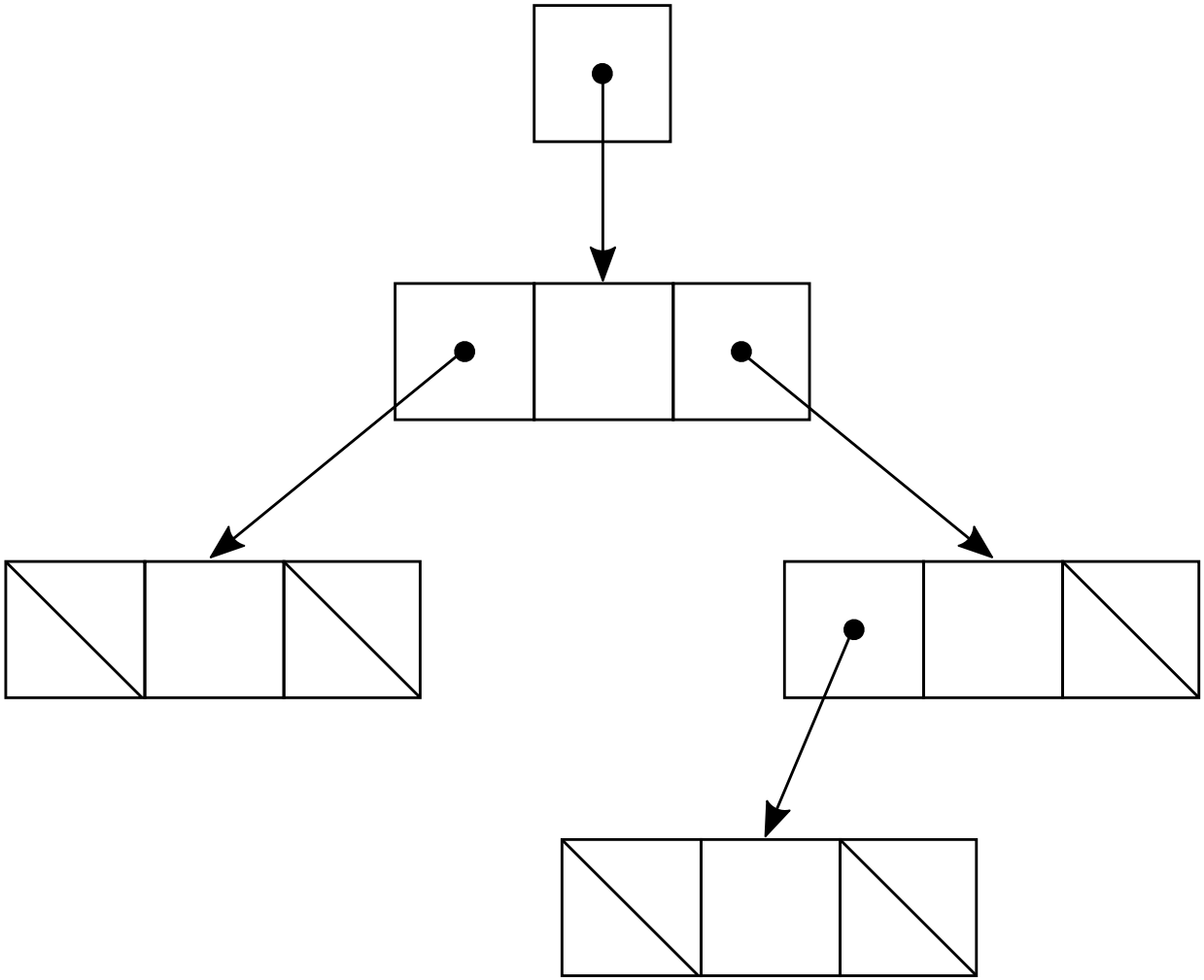
- El *nodo raíz* es el primer nodo del árbol
- Cada enlace en el nodo raíz apunta a un hijo
- El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- El *hijo derecho* es el primer nodo del *sub-árbol derecho*
- Los hijos de los nodos se conocen como *descendientes*

## Árboles

- Son estructura de datos no lineales (a diferencia de las listas, pilas y colas)
- Los nodos de los *árboles* contienen dos o más enlaces

*Árboles binarios:* todos los nodos contienen dos enlaces (ninguno, uno o ambos definidos a `NULL`)

- El *nodo raíz* es el primer nodo del árbol
- Cada enlace en el nodo raíz apunta a un hijo
- El *hijo izquierdo* es el primer nodo del *sub-árbol izquierdo*
- El *hijo derecho* es el primer nodo del *sub-árbol derecho*
- Los hijos de los nodos se conocen como *descendientes*
- Un nodo sin hijos se conoce como *nodo hoja*



# Estructuras de datos
## Árbol de búsqueda binario

- No contiene nodos duplicados

# Estructuras de datos

## Árbol de búsqueda binario

- No contiene nodos duplicados
- Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre

# Estructuras de datos

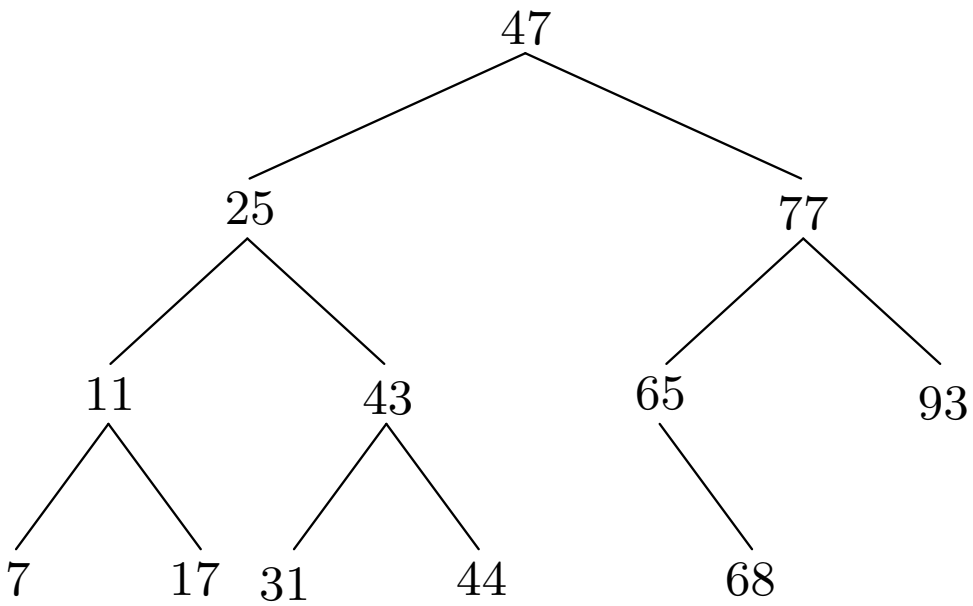
## Árbol de búsqueda binario

- No contiene nodos duplicados
- Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre
- Los valores de cualquier sub-árbol derecho son mayores que el valor de su nodo padre

# Estructuras de datos

## Árbol de búsqueda binario

- No contiene nodos duplicados
- Los valores de cualquier sub-árbol izquierdo son menores que el valor de su nodo padre
- Los valores de cualquier sub-árbol derecho son mayores que el valor de su nodo padre



# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19

Ver ejemplo en funcionamiento

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19

Ver ejemplo en funcionamiento

- En un árbol de búsqueda binario el nodo se inserta únicamente como nodo hoja
- Se recorre de tres formas: *enorden*, *preorden*, y *postorden*.

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19

Ver ejemplo en funcionamiento

- En un árbol de búsqueda binario el nodo se inserta únicamente como nodo hoja
- Se recorre de tres formas: *enorden*, *preorden*, y *postorden*.

```
The numbers being placed in the tree are:
  3 10  1  7  5 14 10dup  5dup  7dup  2

The preOrder traversal is:
  3  1  2 10  7  5 14

The inOrder traversal is:
  1  2  3  5  7 10 14

The postOrder traversal is:
  2  1  5  7 14 10  3
```

Dentro de la función `insertNode()`:

1. Si `*treePtr` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*treePtr`, asigna el valor a almacenar en `(*treePtr)->data`, y los enlaces `(*treePtr)->leftPtr` y `(*treePtr)->rightPtr` a `NULL`.

Dentro de la función `insertNode()`:

1. Si `*treePtr` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*treePtr`, asigna el valor a almacenar en `(*treePtr)->data`, y los enlaces `(*treePtr)->leftPtr` y `(*treePtr)->rightPtr` a `NULL`.

2. Si `*treePtr` no es `NULL`, y el valor a insertar es menor que `(*treePtr)->data` se llama a `insertNode()` con la dirección `(*treePtr)->leftPtr`; si no, se llama a `insertNode()` con la dirección `(*treePtr)->rightPtr`.

Dentro de la función `insertNode()`:

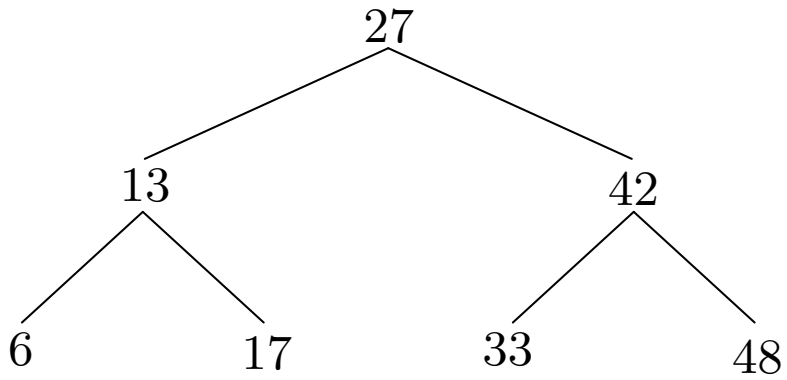
1. Si `*treePtr` es `NULL`, crea un nuevo nodo llamando a `malloc()` y apuntado por `*treePtr`, asigna el valor a almacenar en `(*treePtr)->data`, y los enlaces `(*treePtr)->leftPtr` y `(*treePtr)->rightPtr` a `NULL`.

2. Si `*treePtr` no es `NULL`, y el valor a insertar es menor que `(*treePtr)->data` se llama a `insertNode()` con la dirección `(*treePtr)->leftPtr`; si no, se llama a `insertNode()` con la dirección `(*treePtr)->rightPtr`.

3. Se sigue la recursividad hasta que se encuentre un puntero `NULL` y se ejecuta el paso 1

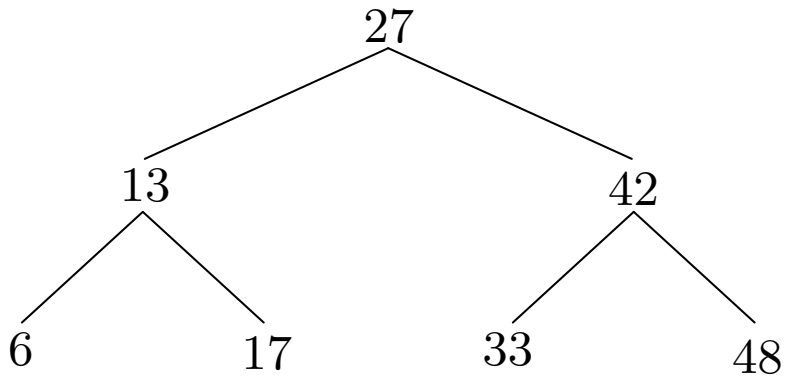
# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos



# Estructuras de datos

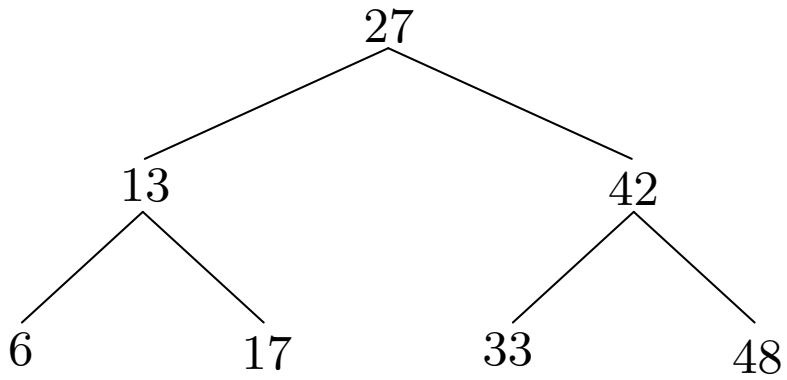
## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos




- *preorden*: `27 13 6 17 42 33 48` [<raíz> <izq> <der>]

# Estructuras de datos

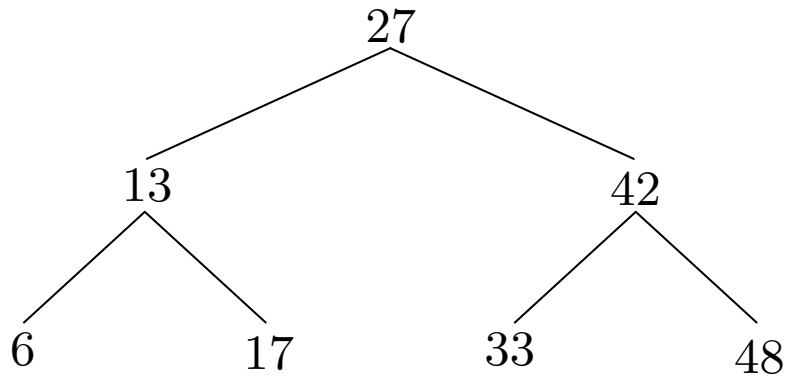
## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos




- *preorden*: `27 13 6 17 42 33 48` [`<raíz> <izq> <der>`]
  se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho

# Estructuras de datos

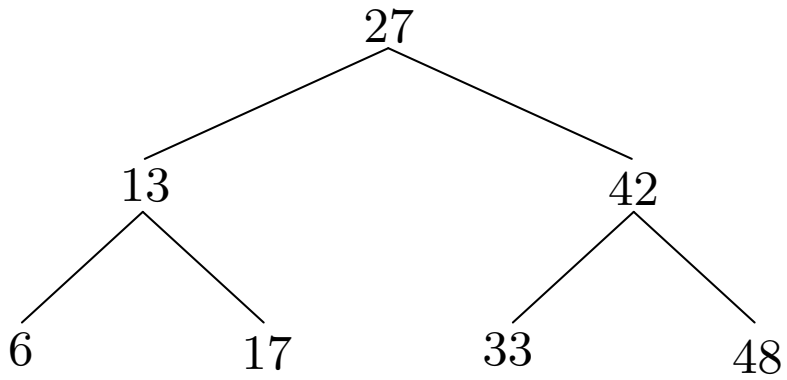
## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos




- *preorden*: `27 13 6 17 42 33 48` [<raíz> <izq> <der>]
  se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho
- *enorden*: `6 13 17 27 33 42 48` [<izq> <raíz> <der>]

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos

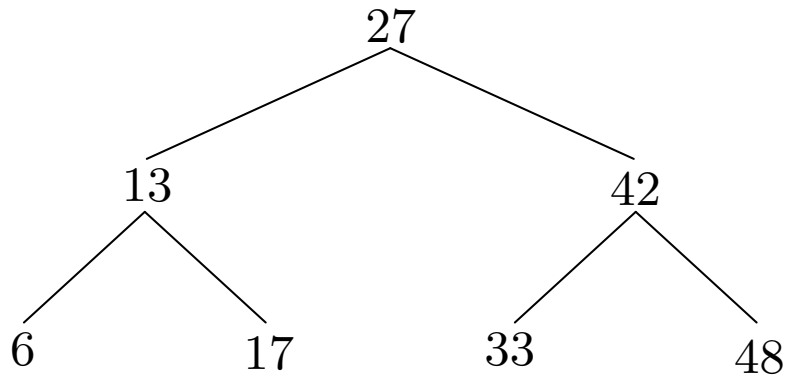



- *preorden*: `27 13 6 17 42 33 48` [`<raíz> <izq> <der>`]
  se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho

- *enorden*: `6 13 17 27 33 42 48` [`<izq> <raíz> <der>`]
  imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos



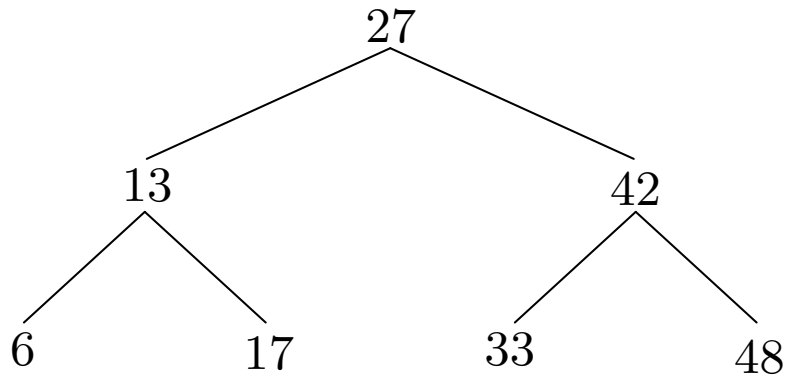

- *preorden*: `27 13 6 17 42 33 48` [`<raíz> <izq> <der>`]
  se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho

- *enorden*: `6 13 17 27 33 42 48` [`<izq> <raíz> <der>`]
  imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)

- *postorden*: `6 17 13 33 48 42 27` [`<izq> <der> <raíz>`]

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Recorridos




- *preorden*: `27 13 6 17 42 33 48` [`<raíz> <izq> <der>`]
  se procesa cada nodo conforme se pasa por el, luego se procesa el sub-árbol izquierdo y a continuación el sub-árbol derecho

- *enorden*: `6 13 17 27 33 42 48` [`<izq> <raíz> <der>`]
  imprime los valores de los nodos en forma ascendente (el proceso de cargar un árbol de búsqueda binario de hecho ordena los datos)

- *postorden*: `6 17 13 33 48 42 27` [`<izq> <der> <raíz>`]
  no se imprime el valor de cada nodo hasta que sean impresos los valores de sus hijos

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Código fuente

```
1  /* Create a binary tree and traverse it
2   * preorder, inorder, and postorder */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  struct treeNode {
8    int data;
9    struct treeNode *leftPtr;
10   struct treeNode *rightPtr;
11 };
12
13 typedef struct treeNode TreeNode;
14 typedef TreeNode *TreeNodePtr;
15
16 void insertNode(TreeNodePtr *, int );
17 void inOrder(TreeNodePtr );
18 void preOrder(TreeNodePtr );
19 void postOrder(TreeNodePtr );
20
21 int main(void)
22 {
23   int i, item;
24   TreeNodePtr rootPtr = NULL;
25
26   srand(time(NULL));
27
```

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Código fuente (cont.)

```
28    /* attempt to insert 10 random values between 0 and 14 in the tree */
29    printf("The␣numbers␣being␣placed␣in␣the␣tree␣are:\n");
30
31    for(i = 1; i <= 10; i++)
32    {
33      item = rand() % 15;
34      printf("%3d", item);
35      insertNode(&rootPtr, item);
36    }
37
38    /* traverse the tree preOrder */
39    printf("\n\nThe␣preOrder␣traversal␣is:\n");
40    preOrder(rootPtr);
41
42    /* traverse the tree inOrder */
43    printf("\n\nThe␣inOrder␣traversal␣is:\n");
44    inOrder(rootPtr);
45
46    /* traverse the tree postOrder */
47    printf("\n\nThe␣postOrder␣traversal␣is:\n");
48    postOrder(rootPtr);
49
50    return 0;
51  }
52
```

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Código fuente (cont.)

```
53 void insertNode(TreeNodePtr *treePtr, int value)
54 {
55   if(*treePtr == NULL)
56   {
57     *treePtr = malloc(sizeof(TreeNode));
58
59     if(*treePtr != NULL)
60     {
61       (*treePtr)->data = value;
62       (*treePtr)->leftPtr = NULL;
63       (*treePtr)->rightPtr = NULL;
64     }
65     else
66       printf("%d␣not␣inserted.␣No␣memory␣available.\n", value);
67   }
68   else
69   {
70     if(value < (*treePtr)->data)
71       insertNode( &((*treePtr)->leftPtr), value );
72     else
73       if(value > (*treePtr)->data)
74         insertNode( &((*treePtr)->rightPtr), value );
75       else
76         printf("dup");
77   }
78 }
79
```

# Estructuras de datos

## Árbol de búsqueda binario – ejemplo D&D Fig. 12.19 – Código fuente (cont.)

```
80  void inOrder(TreeNodePtr treePtr)
81  {
82    if(treePtr != NULL) {
83      inOrder(treePtr->leftPtr);
84      printf("%3d", treePtr->data);
85      inOrder(treePtr->rightPtr);
86    }
87  }
88
89  void preOrder(TreeNodePtr treePtr)
90  {
91    if(treePtr != NULL) {
92      printf("%3d", treePtr->data);
93      preOrder(treePtr->leftPtr);
94      preOrder(treePtr->rightPtr);
95    }
96  }
97
98  void postOrder(TreeNodePtr treePtr)
99  {
100   if(treePtr != NULL) {
101     postOrder(treePtr->leftPtr);
102     postOrder(treePtr->rightPtr);
103     printf("%3d", treePtr->data);
104   }
105 }
106
```