Pilas dinámicas
Calificadores de acceso y almacenamiento

Sofía Beatriz Pérez
Daniel Agustín Rosso

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

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Clase número 9 - Ciclo lectivo 2023

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

Pilas o stacks: introducción

Una pila es una estructura autoreferenciada a la cual se le pueden añadir y retirar nuevos nodos **únicamente** por su parte superior. Por esta razón, a las pilas se las conoce como LIFO (last input, first output).

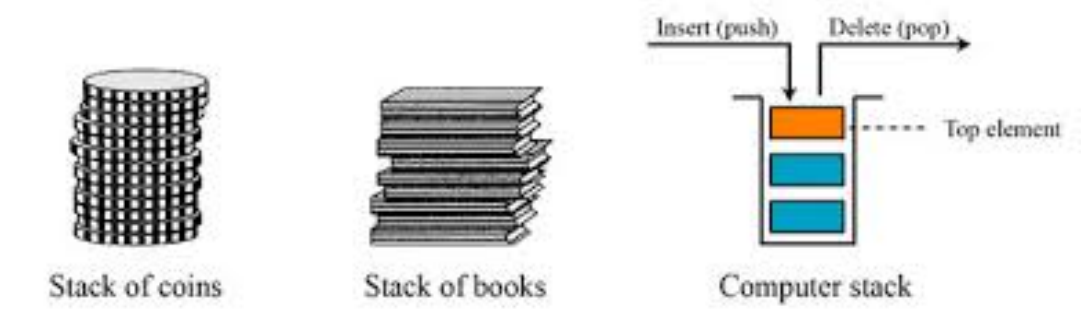


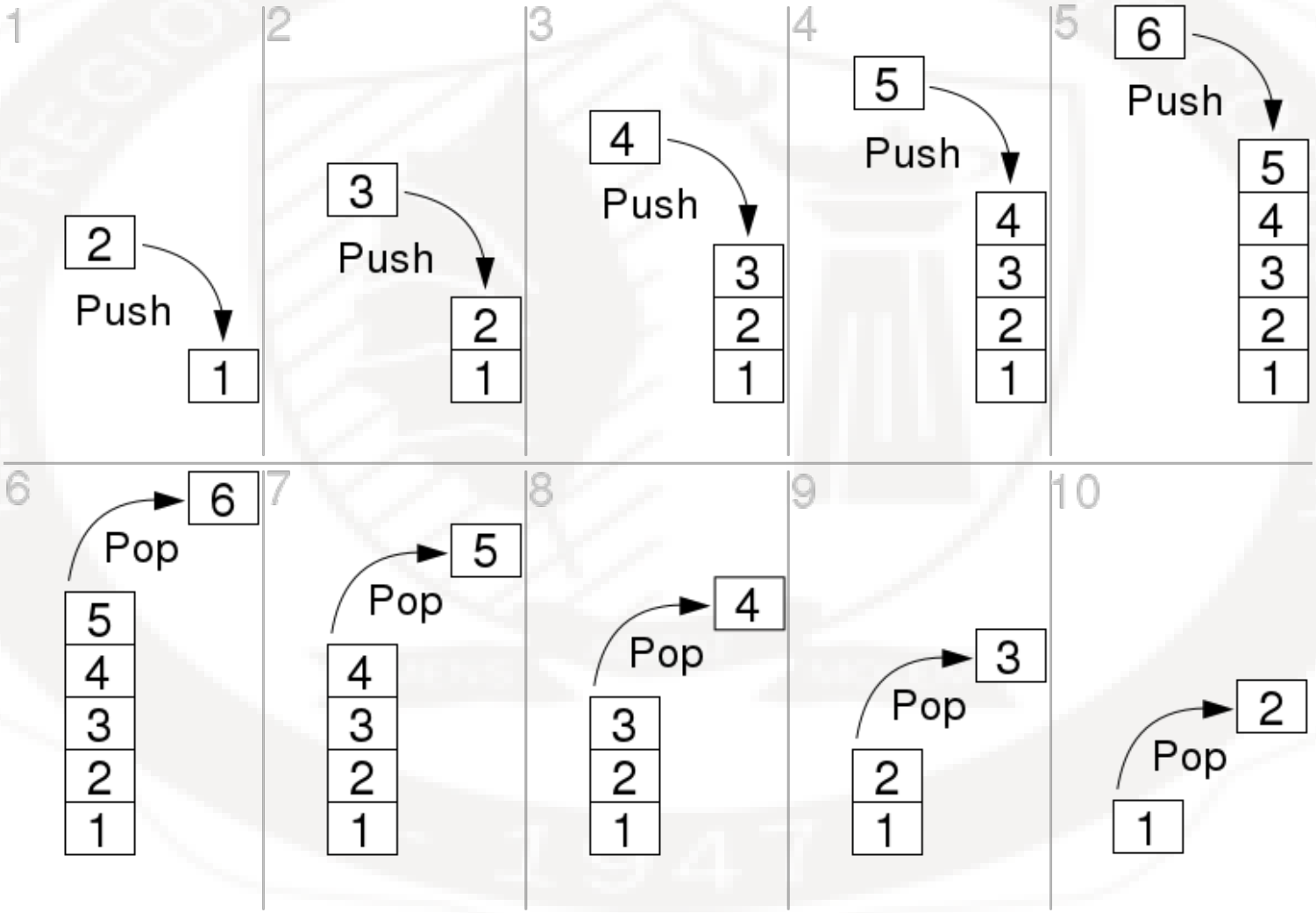
Figure: Representación gráfica de una cola.

stacks: operaciones I

Las operaciones más comunes con pilas dinámicas son:

- Agregar un nodo (push)
- Eliminar un nodo (pop)
- Calcular la cantidad de elementos almacenados en la LIFO
- Imprimir todos los nodos de la LIFO
- Verificar si la LIFO está vacía

stacks: operaciones II



Stacks: creación

```
1  struct nodo
2  {
3     int data;
4     struct nodo *siguiente;
5  };
```

Se utilizara un puntero para localizar al primer elemento de la pila y otro auxiliar para realizar operaciones. Notar que ambos punteros son locales a la función main():

```
1  int main()
2  {
3        struct node *stackptr = NULL;
4        struct node *temp     = NULL;
```

Stacks: push

```
1  struct node *new_node = NULL;
2  /*Creacion de memoria*/
3  new_node = (struct node *) malloc(sizeof(struct node));
4  /*Verificacion de memoria disponible*/
5  if(new_node==NULL)
6  {
7      printf("No Memory available\n");
8      exit(0);
9  }
10 /*Carga util*/
11 new_node->numero = numero;
12 /*Asignamos el siguiente del nuevo nodo
13 al stack pointer actual*/
14 new_node->siguiente = stackptr;
15 /*Ahora el nuevo nodo es el stackptr*/
16 stackptr = new_node;
```

Stacks: pop

```
1  /*Verificamos que la pila no este vacia*/
2  if (stackptr != NULL)
3  {
4      /*Asignamos en temp el stack pointer actual*/
5      temp = stackptr;
6      /*Asignamos al stack pointer, el valor siguiente
7      del primer nodo*/
8      stackptr = stackptr->siguiente;
9      /*Liberamos la memoria ocupada por el primer nodo*/
10     free(temp);
11 }
12 else
13 {
14     printf("Pila vacia\n");
15 }
```

Stacks: impresión

```
1  /*Comenzamos a recorrer desde el stack pointer*/
2  temp = stackptr;
3  while (temp != NULL)
4  {
5      printf("%d",temp->data);
6      temp = temp->siguiente;
7      /*Recordar que el ultimo nodo de la stack,
8      en siguiente apunta a NULL*/
9  }
```

Stacks: cálculo de datos almacenados

```
1  /*Comenzamos a recorrer desde el stack pointer*/
2  temp = stackptr;
3  int number_of_nodes=0;
4  while (temp != NULL)
5  {
6      number_of_nodes+=1;
7      /*Asignamos a temp el puntero al siguiente nodo*/
8
9      temp = temp->siguiente;
10     /*Recordar que el ultimo nodo de la stack,
11     en siguiente apunta a NULL*/
12 }
```

Stack implementación completa I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  struct node
6  {
7    int numero;
8    struct node *siguiente;
9  };
10
11 void menu(void)
12 {
13      printf("1.- Agregar\n2.- Eliminar\n");
14      printf("3.- Imprimir \n-1.- Salir\n\n");
15 }
```

Stack implementación completa II

```
16  int main()
17  {
18      int op=0;
19      int numero;
20      struct node *stackptr = NULL;
21      struct node *temp     = NULL;
22
23      while (op != -1)
24      {
25      menu();
26      scanf("%d", &op);
27      switch (op)
28      {
29
30
```

Stack implementación completa III

```
31      case 1:
32          printf("Ingresa el numero a agregar:\n");
33          scanf("%d", &numero);
34          struct node *new_node = NULL;
35          new_node = malloc(sizeof(struct node));
36          new_node = (struct node *) new_node;
37          if(new_node==NULL)
38          {   printf("No Memory available\n");
39              exit(0);
40          }
41
42          new_node->numero = numero;
43          new_node->siguiente = stackptr;
44          stackptr = new_node;
45      break;
```

Stack implementación completa IV

```
46          case 2:
47              if (stackptr != NULL)
48              {
49                  temp = stackptr;
50                  stackptr = stackptr->siguiente;
51                  free(temp);
52              }
53              printf("Pila vacia\n");
54          break;
55
56
57
58
59
60
```

Stack implementación completa V

```
61          case 3:
62                  printf("Imprimiendo...\n");
63                  temp = stackptr;
64                  while (temp != NULL)
65                  {
66                          printf("%d\n", temp->numero);
67                          temp = temp->siguiente;
68                  }
69          break;
70          }
71      }
72      return(0);
73  }
```

Stack implementación con funciones I

```c
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int numero;
   struct node *siguiente;
};

/*Prototipo de funciones*/
void push(struct node **, int);
void pop(struct node **);
void print(struct node **);
int isempty(struct node *);
int count_nodes(struct node **);
```

Stack implementación con funciones II

```
16
17  void menu(void)
18  {
19      printf("1- Agregar\n2.- Eliminar\n");
20      printf("3- Imprimir\n4- Contar nodos\n\n");
21      printf("-1.- Salir\n");
22  }
23
24
25
26
27
28
29
30
```

Stack implementación con funciones III

```
31  void push (struct node **sp, int dato)
32  {
33      struct node *new_node = NULL;
34      new_node = malloc(sizeof(struct node));
35      new_node = (struct node *) new_node;
36      if(new_node==NULL)
37       {    printf("No Memory available\n");
38            exit(0);
39       }
40      new_node->numero = dato;
41      new_node->siguiente = *(sp);
42      *(sp) = new_node;
43  }
44
45
```

Stack implementación con funciones IV

```
46  int isempty (struct node *sp)
47  {
48      if(sp==NULL)
49          return(1);
50      else
51          return(0);
52  }
53  void pop (struct node **sp)
54  {
55      struct node *temp=NULL;
56      temp = *(sp);
57      **(sp) = *(*sp)->siguiente;
58      //*(sp) = (*sp)->siguiente;
59      free(temp);
60  }
```

Stack implementación con funciones V

```
61
62  void print (struct node **sp)
63  {
64      struct node *temp;
65      temp = *(sp);
66      while (temp != NULL)
67      {
68          printf("%d\n", temp->numero);
69          temp = temp->siguiente;
70      }
71  }
72
73
74
75
```

Stack implementación con funciones VI

```
76  int count_nodes (struct node **sp)
77  {
78      int acum=0;
79      struct node *temp;
80      temp = (*sp);
81      while (temp != NULL)
82      {
83          acum+=1;
84          temp = temp->siguiente;
85      }
86      return(acum);
87  }
88
89
90
```

Stack implementación con funciones VII

```
91  int main()
92  {
93      int op=0;
94      int numero;
95      struct node *stackptr = NULL;
96      struct node *temp;
97
98      while (op != -1)
99      {
100     menu();
101     scanf("%d", &op);
102     switch (op)
103     {
104
105
```

Stack implementación con funciones VIII

```
106         case 1:
107             printf("Ingresa el numero a agregar:\n");
108             scanf("%d", &numero);
109             push(&stackptr,numero);
110             break;
111         case 2:
112             if(isempty(stackptr)==0)
113             {
114                 pop(&stackptr);
115             }
116             else
117             {
118                 printf("Pila vacia");
119             }
120             break;
```

Stack implementación con funciones IX

```
121         case 3:
122             printf("Imprimiendo...\n");
123             print(&stackptr);
124             break;
125         case 4:
126             printf("Nodos activos");
127             printf("%d\n\n",count_nodes(&stackptr));
128 
129             break;
130         }
131     }
132     return(0);
133 }
```

Calificadores de acceso I

Hay dos calificadores que controlan como las variables pueden ser accedidas o modificadas:

Const

El calificador const se utiliza para hacer que una variable sea constante, es decir, su valor no puede ser modificado después de su inicialización. También se utiliza para indicar que un puntero o una referencia no pueden modificar el valor al que apuntan o al que se refieren.

```
1  const int a=10;
```

Calificadores de acceso II

Al calificar una variable como const, el compilador es libre de ubicar este tipo de variables en memoria de sólo lectura por ejemplo.
Si se intenta modificar el valor de una variable calificada como const dentro del programa, el compilador arrojará un error asociado.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     const int a=10;
6     printf("%d",a);
7     return(0);
8 }
```

Calificadores de acceso III

Esto es particularmente útil para prevenir que una función modifique los valores de un objeto enviado a ella por referencia. Por ejemplo, para la impresión de un arreglo.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int valor = 100;
6      const int *ptr = &valor;
7      // (*ptr) = 50; // Esto generara un error
8      int const *ptr2 = &valor; // Sintaxis alternativa
9
10 }
```

Calificadores de acceso IV

Volatile

El modificador **volatile** le comunica al compilador que el valor de la variable puede cambiar de forma no explícita. Por ejemplo una variable global podría leer la hora desde el sistema operativo. En este caso, el valor de la variable puede verse alterado sin ninguna asignación dentro del programa. Esto es útil cuando se trabaja con hardware o variables que pueden ser modificadas por hilos de ejecución diferentes.

```
1 volatile int a=10;
```

Calificadores de acceso V

```
1
2
3  void funcion()
4  {
5      static int contador = 0; // Variable estatica local
6      contador++;
7      printf("Contador: %d\n", contador);
8  }
```

Es importante aclarar que al ver este modificador, el compilador no realizará ningún tipo de optimización sobre esa variable. [1]

[1] Esto se profundizará en Desarrollo de Herramientas de Software.

Calificadores de almacenamiento I

Existen cuatro especificadores de almacenamiento que le indican al compilador como almacenar las variables.

Extern

La palabra reservada **extern** puede ser aplicada a variables globales o funciones. Le especifica al compilador que este símbolo participa de un proceso de linkeo externo.

```
1  //Archivo_1.c
2  extern int ii = 42;
3
4  //Archivo_2.c
5  extern int ii;  //Mismo valor que en archivp_1
```

Calificadores de almacenamiento II

Static

El modificador static se utiliza para controlar el alcance y almacenamiento de variables y funciones:

- Variables estáticas locales: Las variables locales declaradas como static conservan su valor entre las llamadas a la función y tienen un alcance limitado al bloque o archivo en el que se declaran.

Calificadores de almacenamiento III

```c
#include <stdio.h>
void funcion()
{
    static int contador = 0; // Variable estatica local
    contador++;
    printf("Contador: %d\n", contador);
}
int main (void)
{
    funcion();
    funcion();
    funcion();
    funcion();
    return(0);
}
```

Calificadores de almacenamiento IV

- Variables estáticas globales: Las variables globales declaradas como static tienen un alcance limitado al archivo en el que se declaran, lo que significa que no se pueden acceder desde otros archivos.
- Funciones estáticas: Las funciones declaradas como static tienen un alcance limitado al archivo en el que se definen y no se pueden llamar desde otros archivos.

```
1  // Variable estatica global
2  static int variable_global = 5;
3
4  static void funcionEstatica()
5  {
6      ...
7  }
```

Calificadores de almacenamiento V

Creando nuestra primera biblioteca I

- Desde los primeros programas que se han construido, se ha utilizado la directiva *#include*.
- Esto ha sido particularmente útil para no preocuparse por la codificación de las funciones *printf* o *scanf* entre otras.
- La motivación es crear nuestra propia biblioteca de funciones para, por ejemplo, poder trabajar con estructuras dinámicas re-utilizando el esfuerzo de codificación.

Para realizar esto, se trabajará utilizando el preprocesador de C [2]., junto a los calificadores estudiados anteriormente.

Creando nuestra primera biblioteca II

Algunas reglas a seguir:

1. Los archivos .h (headers) contienen sólo los prototipos de las funciones y variables globales si es que existen.
2. Los archivos .c/.cpp contiene la implementación de la función
3. Debemos garantizar que al momento de compilar no existen múltiples definiciones de variables y funciones en los diferentes archivos
4. Para evitar incluir múltiples veces a un mismo archivo .h, se propone el siguiente template

Creando nuestra primera biblioteca III

```
1 #ifndef FILENAME_H
2 #define FILENAME_H
3
4 //Codigo fuente
5
6 #endif
```

▸ Ver ejemplo completo en gitlab ▸ Ver ejemplo completo en gitlab

[2] https://en.wikipedia.org/wiki/C_preprocessor

¡Muchas gracias!

Consultas:

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