Arrays, pilas y colas

...

Algoritmos y Estructuras de Datos
2023

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

¿Cómo organizaría las llamadas recursivas a una función en una estructura de datos?

Llamadas recursivas

- ¿Cómo organizaría las llamadas recursivas a una función en una estructura de datos?
- ¿Cómo se agregarían o quitarían llamadas de dicha estructura?
- ¿Cómo se organizarían las sentencias de un código secuencial?

Tipos de Datos Primitivos y Abstractos

- En general, se distinguen 4 tipos de **datos primitivos** (TDP):
  - Enteros (int)
  - Reales (float)
  - Lógicos (bool)
  - Caracteres (char) => NO EXISTE EN PYTHON!
- Los **tipos de datos abstractos** (TDA) son estructuras que pueden combinar distintos TDP (y otros TDA) y poseen comportamiento propio.
- En Python, por defecto existen 3 TDA:
  - Tuplas
  - Listas
  - Diccionarios

Memoria de una computadora

- La memoria RAM está compuesta por *bits* (0s y 1s), agrupados en unidades más grandes, como los *bytes* (8 bits)
- Al tener una enorme cantidad de *bytes*, la computadora guarda donde está almacenado cada dato usando **direcciones de memoria**.
- Al saber donde está almacenado, puede acceder rápidamente a cada variable y las variables pueden modificar su contenido sin cambios grandes en el código.

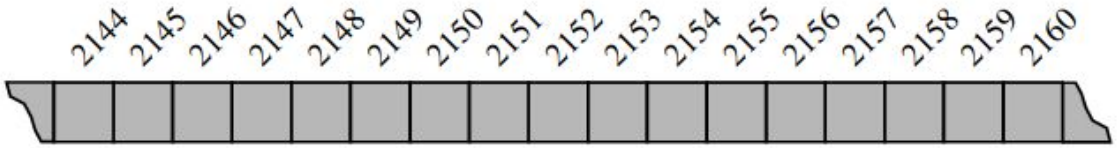


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Arrays

- Un grupo de variables relacionadas pueden guardarse de forma continua en la memoria. Estas estructuras se denominan ***arrays***.
- Un ejemplo son los ***strings***, que se pueden definir como una secuencia ordenada de caracteres individuales con un **tamaño fijo**.
- Todos los datos de un *array* deben ser del mismo tipo, para calcular fácilmente cuántos bytes son necesarios para almacenar todo un array y encontrar un bloque de memoria disponible de ese tamaño.

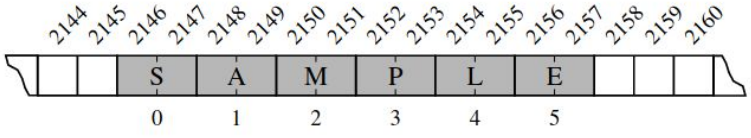


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

Arrays dinámicos

- Por definición, un *array* no puede cambiar de tamaño, ya que puede sobreescribir fragmentos de memoria que estén siendo usados.
- Para agregar elementos a un *array* se debe buscar un nuevo espacio en memoria que permita almacenar todos los datos, crear un nuevo *array* y eliminar el anterior.
- La clase ***list*** de Python es un ejemplo de arrays dinámicos.

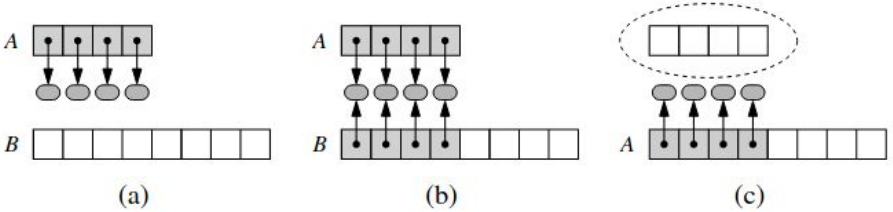


Figure 5.12: An illustration of the three steps for "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

Arrays referenciados

- Es posible almacenar dentro de un array elementos de distintos tamaños.
- En lugar de guardar el elemento, se almacena la dirección de memoria donde se encuentra ese elemento.
- Todas las direcciones de memoria tienen el mismo tamaño, por lo que se cumple la condición necesaria que define un array.
- Una variable que almacena una dirección de memoria se la denomina **puntero**.

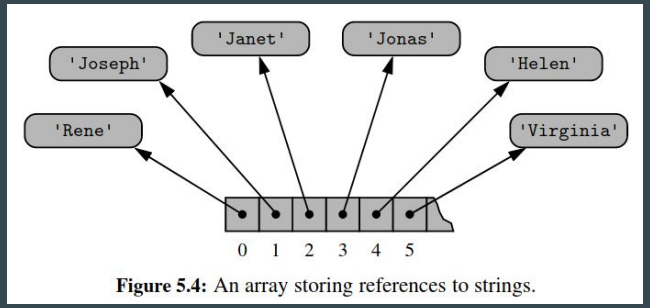
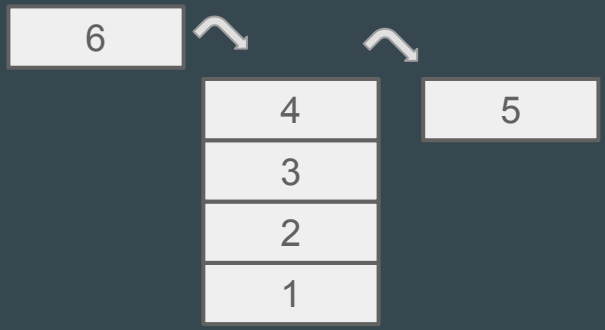


Figure 5.4: An array storing references to strings.

Pilas

- Las pilas son colecciones de objetos con reglas específicas para agregar y quitar nuevos objetos.
- Las pilas siguen el principio ***Last in, First out*** o **LIFO**.



Métodos en pilas

- empty: devuelve *True* si la pila no contiene ningún elemento
- push: agrega un elemento al final de la pila.
- pop: elimina y devuelve el elemento del final de la pila
- top: devuelve una referencia al último elemento de la pila sin eliminarlo
- len(S): devuelve el número de elementos de la pila

Ejemplo de funcionamiento de una pila

Operation	Return Value	Stack Contents
S.push(5)		
S.push(3)		
len(S)		
S.pop()		
S.empty()		
S.pop()		
S.empty()		
S.pop()		
S.push(7)		

Operation	Return Value	Stack Contents
S.push(9)		
S.top()		
S.push(4)		
len(S)		
S.pop()		
S.push(6)		
S.push(8)		
S.pop()		
S.top()		

Ejemplo de funcionamiento de una pila

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.empty()	False	[5]
S.pop()	5	[]
S.empty()	True	[]
S.pop()	Error	[]
S.push(7)	–	[7]

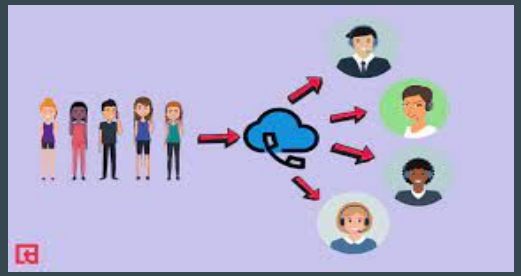
Operation	Return Value	Stack Contents
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]
S.top()	6	[7, 9, 6]

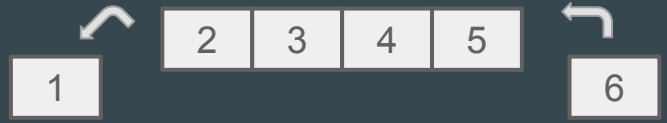
Codificando pilas usando listas

```
class ArrayStack:
    def __init__ (self):
        self._data = []

    def len (self):
        return len(self._data)

    def empty(self):
        if len(self._data) == 0:
            return True

    def push(self, e):
        self._data.append(e)

    def top(self):
        if self.empty():
            raise IndexError("Stack is empty")
        return self._data[-1]
```

Colas

- Otra estructura de datos fundamental son las ***colas***.
- En este caso, el funcionamiento sigue el principio ***First in, first out*** o ***FIFO***.





Métodos en colas

- empty: devuelve *True* si la cola no contiene ningún elemento
- enqueue: agrega un elemento al final de la cola.
- dequeue: elimina y devuelve el primer elemento de la cola
- first: devuelve una referencia al primer elemento de la cola sin eliminarlo
- len(S): devuelve el número de elementos de la cola

Ejemplo de funcionamiento de una cola

Operation	Return Value	Stack Contents
Q.enqueue(5)		
Q.enqueue(3)		
len(Q)		
Q.dequeue()		
Q.empty()		
Q.dequeue()		
Q.empty()		
Q.dequeue()		
Q.enqueue(7)		
Q.enqueue(9)		
Q.first()		
Q.enqueue(4)		
len(Q)		
Q.dequeue()		
Q.enqueue(6)		
Q.enqueue(8)		
Q.dequeue()		
Q.first()		

Ejemplo de funcionamiento de una cola

Operation	Return Value	Stack Contents
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.empty()	False	[3]
Q.dequeue()	3	[]
Q.empty()	True	[]
Q.dequeue()	Error	[]
Q.enqueue(7)	–	[7]

Operation	Return Value	Stack Contents
Q.enqueue(9)	–	[7, 9]
Q.first()	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue()	7	[9, 4]
Q.enqueue(6)	–	[9, 4, 6]
Q.enqueue(8)	–	[9, 4, 6, 8]
Q.dequeue()	9	[4, 6, 8]
Q.first()	4	[4, 6, 8]

Codificando colas usando listas

```
class ArrayQueue:
    def __init__ (self):
        self._data = []

    def len (self):
        return len(self._data)

    def empty(self):
        if len(self._data) == 0:
            return True

    def enqueue(self, e):
        self._data.append(e)

    def first(self):
        if self.empty():
            raise IndexError(“Stack is empty”)
        return self._data[0]
```

Colas dobles

- Otro tipo de dato muy utilizado en informática son las colas dobles.
- Este tipo de datos permiten añadir y eliminar objetos por ambos extremos.

Métodos en colas dobles

- add_first (e): añade un elemento al comienzo de la cola doble.
- add_last (e): añade un elemento al final de la cola doble.
- delete_first (e): elimina el elemento al comienzo de la cola doble.
- delete_last (e): elimina el elemento al final de la cola doble.
- first(): devuelve, sin eliminar, el primer elemento de la cola doble.
- last(): devuelve, sin eliminar, el último elemento de la cola doble.
- empty( ): devuelve *True* si la estructura está vacía.
- len(D): devuelve el número de elementos en la cola doble.

Ejemplo de funcionamiento de una cola doble

Operation	Return Value	Stack Contents
D.add_first(5)		
D.add_last(3)		
len(D)		
D.delete_last()		
D.empty()		
D.delete_last()		
D.empty()		
D.delete_first()		
D.add_last(7)		

Operation	Return Value	Stack Contents
D.add_last(9)		
D.last()		
D.add_last(4)		
len(D)		
D.delete_last()		
D.add_last(6)		
D.add_first(8)		
D.delete_first()		
D.first()		

Ejemplo de funcionamiento de una cola doble

Operation	Return Value	Stack Contents
D.add_first(5)	–	[5]
D.add_last(3)	–	[5, 3]
len(D)	2	[5, 3]
D.delete_last()	3	[5]
D.empty()	False	[5]
D.delete_last()	5	[]
D.empty()	True	[]
D.delete_first()	Error	[]
D.add_last(7)	–	[7]

Operation	Return Value	Stack Contents
D.add_last(9)	–	[7, 9]
D.last()	9	[7, 9]
D.add_last(4)	–	[7, 9, 4]
len(D)	3	[7, 9, 4]
D.delete_last()	4	[7, 9]
D.add_last(6)	–	[7, 9, 6]
D.add_first(8)	–	[8, 7, 9, 6]
D.delete_first()	8	[7, 9, 6]
D.first()	7	[7, 9, 6]

Codificando colas dobles usando listas

```
class ArrayDoubleQueue:
    def __init__ (self):
        self._data = []

    def len (self):
        return len(self._data)

    def empty(self):
        if len(self._data) == 0:
            return True

    def add_first(self, e):
        self._data.insert(0, e)
        # self._data = [e] + self._data

    def add_last(self, e):
        self._data.append(e)
        # self._data = self._data + [e]
```

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 6
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. Capitulo 10