Lecture - 6(Stacks)
On
Data structures

Lecture Outline

- What is a Stack?
- Array implementation of stacks
- Operations on a Stack
- Arithmetic expressions
- stacks are used to evaluate postfix expressions
- Infix expressions into postfix Expressions
- Quicksort

Stacks

- What is a Stack?
- DEFINITION:

  A stack is a homogeneous collection of elements in which an element may be inserted or deleted only at one end, called the top of the stack.

- Formally this type of stack is called a Last In, First Out (LIFO) stack.

- **Special terminology is used for two basic operations associated with stacks:**

a) "Push" is the term used to insert an element into a stack.

b) "Pop" is the term used to delete an element from a stack.

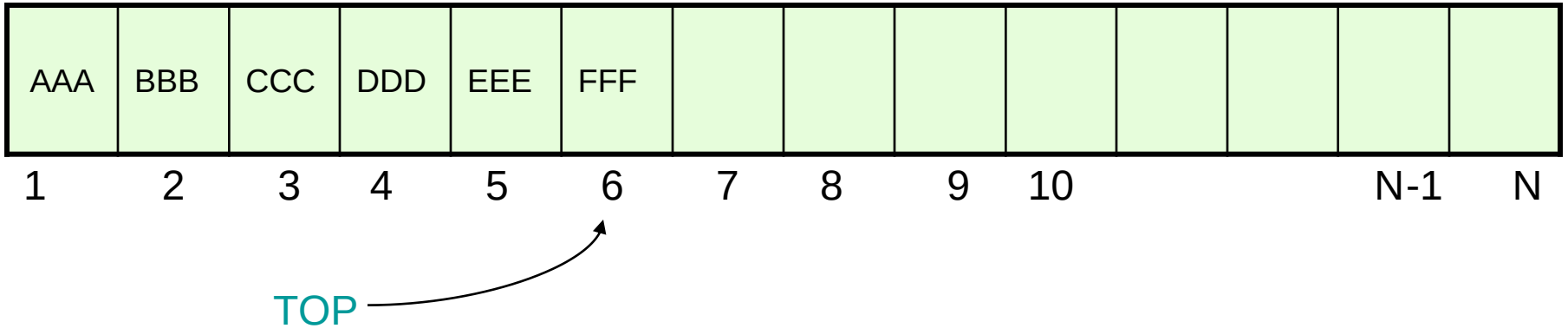
Diagram of stacks

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Frequently designate the stack by writing:

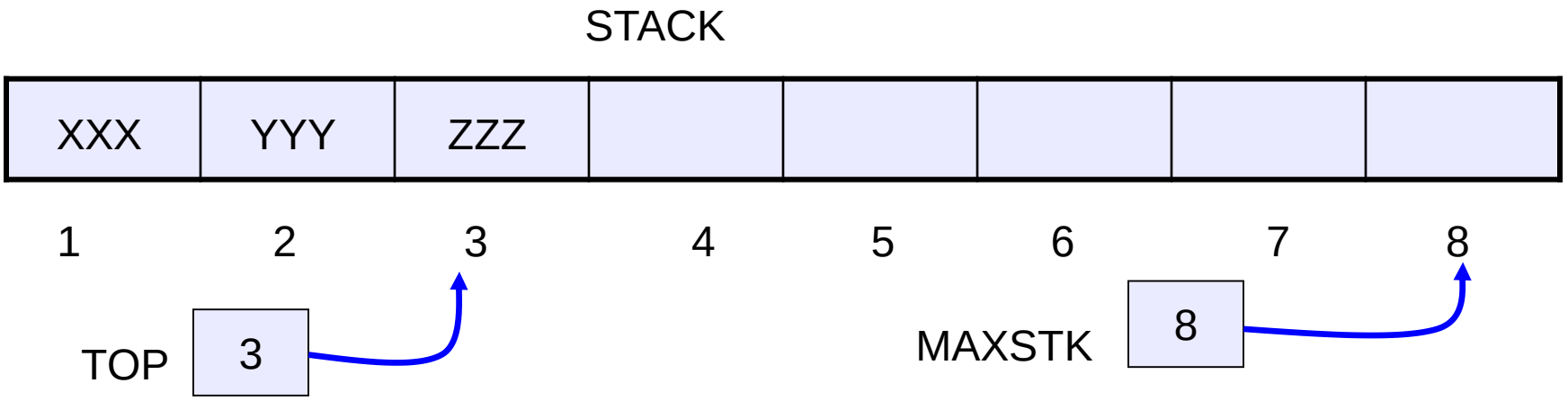
STACK : AAA, BBB, CCC, DDD, EEE, FFF



Array implementation of stacks

- To implement a stack, items are inserted and removed at the same end (called the top)
- To use an array to implement a stack, you need both the array itself and an integer
- The integer tells you either:
  - Which location is currently the top of the stack, or
  - How many elements are in the stack

Array Representation of Stacks



STACK : A linear array

TOP : A pointer variable, Which contains the location of the top element of the stack.

MAXSTK : Gives the maximum number of elements that can be held by the stack.

TOP = 0 or NULL will indicate that the stack is empty

Operations on a Stack

Maximum size of n.

Push : This operation adds or pushes another item onto the stack. The number of items on the stack is less than n.

Pop: This operation removes an item from the stack. The number of items on the stack must be greater than 0.

Top: This operation returns the value of the item at the top of the stack. Note: It does not remove that item.

Is Empty: This operation returns true if the stack is empty and false if it is not.

Is Full: This operation returns true if the stack is full and false if it is not.

These are the basic operations that can be performed on a stack.

Operations on a Stack

Push :

Push is the function used to add data items to the stack.

In order to understand how the Push function operates, we need to look at the algorithm in more detail.

Procedure 6.1 : PUSH(STACK, TOP, MAXSTK, ITEM)

1. If TOP = MAXSTK, then : Print :"OVERFLOW", and return.
2. Set : TOP := TOP+1.
3. STACK[TOP]) := ITEM.
4. Return.

In order to understand the algorithm, let's break it apart line by line.

PUSH(STACK, TOP, MAXSTK, ITEM)

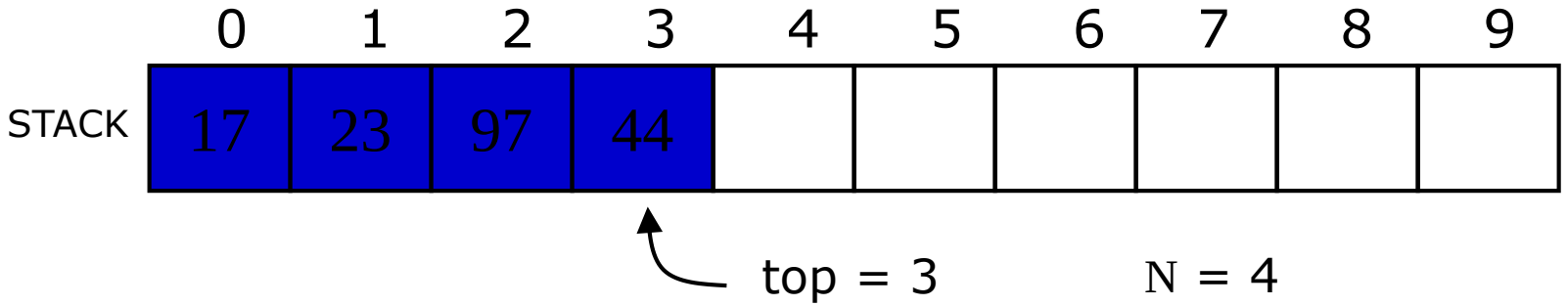
First, Push accepts a parameter - item. This parameter is of the same type as the rest of the stack. Item is the data to be added to the stack.

if **top = MAXSTK**, then stack is full.

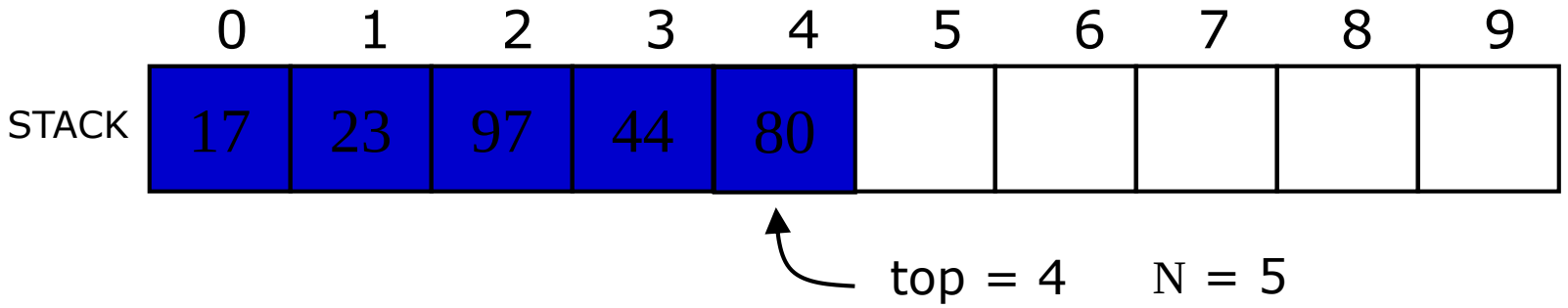
This line performs a check to see whether or not the stack is full.

top := top+1; If the stack is not full, top is increased by a value equal to the size of another item. This allocates room for the insertion.

Pushing



- If ITEM=80 is to be inserted, then TOP=TOP+1=4
- STACK[TOP]:= STACK[4]:= ITEM=80
- N=5
- MAXSTACK=10



Operations on a Stack

Pop :

Data is removed from a stack by using Pop. From a procedural perspective, pop is called with the line Pop(item), where item is the variable to store the popped item in. Once again, we will begin by looking at the algorithm.

Procedure 6.2 POP (STACK, TOP, ITEM);

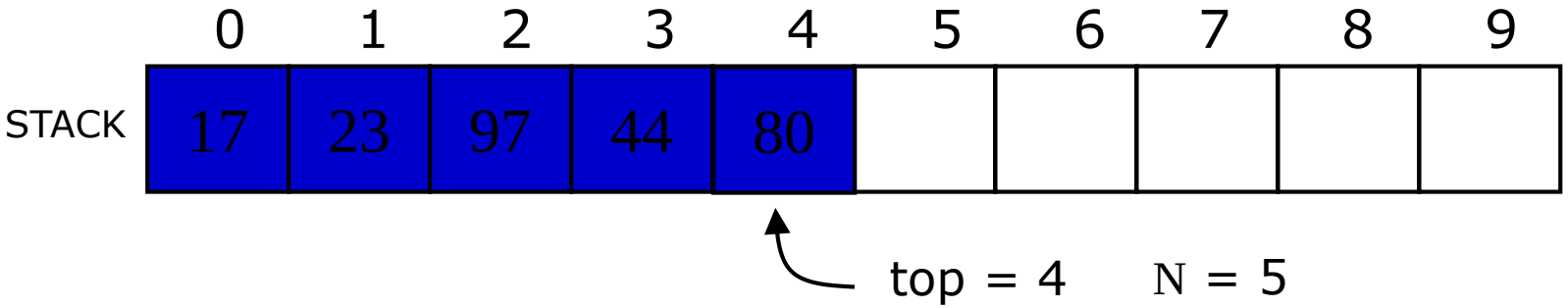
1. If TOP = 0 then : Print :"UNDERFLOW", and return.
2. ITEM : =STACK[TOP]) .
3. Set : TOP := TOP-1
4. Return.

If TOP = 0 then, stack is empty, there is no item to pop off the stack. Control can then be passed to an error handling routine.

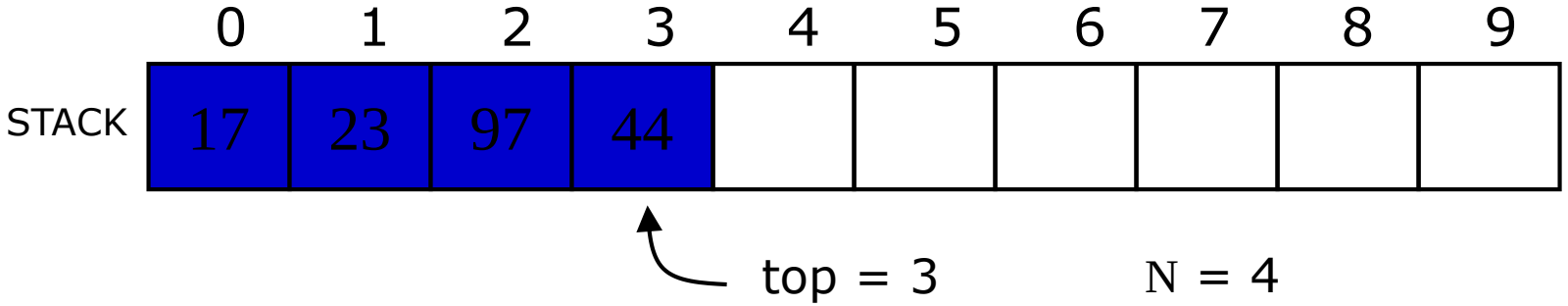
ITEM : =STACK[TOP]) Set item to be equal to the data in the top .

TOP := TOP-1; This statement removes the top item from the stack. Decrement top by 1 so that it now accesses the new top of the stack.

Poping



- If ITEM=80 is to be deleted, then ITEM:=STACK[TOP]= STACK[4]= 80
- TOP=TOP-1=3
- N=4
- MAXSTACK=10



Arithmetic expressions

- **Polish Notation (prefix**) : Polish notation, also known as prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands.
- Lacking parentheses or other brackets.

- Syntax : operator **operand1 operand2**
- Example : -*+ABC/D+EF

Infix notation

- Infix notation is the conventional notation for arithmetic expressions. It is called *infix* notation because
- each operator is placed between its operands,
- operands (as in the case with binary operators such as addition, subtraction, multiplication, division,……).
- When parsing expressions written in infix notation, you need parentheses and precedence rules to remove ambiguity.

- Syntax: **operand1** operator **operand2**
- Example: (A+B)*C-D/(E+F)

Arithmetic expressions

- **Postfix notation**
- In postfix notation, the operator comes after its operands. Postfix notation is also known as *reverse Polish*
- *notation* (RPN) and is commonly used because it enables easy evaluation of expressions.

- Syntax : **operand1 operand2** operator
- Example : AB+C*DEF+/-

Polish Notation

Q an arithmetic expression involving constants and operations
The binary operation in Q may have different levels of precedence :

Highest : Exponentiation(↑)
Next highest : Multiplication(*) and division(/)
Lowest : Addition (+) and subtraction (-)
Ex :
2 ↑ 3 + 5 *2 ↑ 2 – 12 / 6
Evaluating the Exponentiations :
8 + 5 * 4 -12 / 6
Evaluating the Multiplication and division
8 + 20 -2
Evaluating the Addition and subtraction
26

Polish notation

- **Infix expressions into polish notation :**

  (A+B)*C = [+AB]*C = *+ABC

  A+(B*C) = A +[*BC] = +A*BC

  (A+B)/(C-D) =[+AB]/[-CD] = /+AB-CD

  Fundamental property of polish notation is that the order in which the operation are to be performed is completely determined by the positions of the operators and operands in the expression.

Polish notation

An example shows the ease with which a complex statement in prefix notation can be deciphered through order of operations:

- * / 15 - 7 **+ 1 1** 3 + 2 + 1 1 =
- * / 15 **- 7 2** 3 + 2 + 1 1 =
- * **/ 15 5** 3 + 2 + 1 1 =
- *** 3 3** + 2 + 1 1 =
- 9 + 2 **+ 1 1** =
- 9 **+ 2 2** =
- **9 4** =
- **5**

An equivalent in-fix is as follows: ((15 / (7 - (1 + 1))) * 3) - (2 + 1 + 1) = 5

Arithmetic expressions(Reverse Polish notation)

Reverse Polish notation provided a straightforward solution for calculator or computer software mathematics because it treats the instructions (operators) and the data (operands) as "objects" and processes them in a last-in, first-out (LIFO) basis.

This is called a "stack method". (Think of a stack of plates. The last plate you put on the stack will be the first plate taken off the stack.)

The computer evaluates an arithmetic expression written in infix notation in two steps.

1. It converts the expression to postfix notation
2. Then it evaluates the postfix expression

In each step stack is main tool that is used to accomplish the given task

stacks are used to evaluate postfix expressions

Evaluation of a postfix expression

Let P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P

Algorithm 6.3 : This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis “)” at the end of P
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel “)” is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then :
   (a) Remove the two top element of STACK, where A is the top element and B is the next – to – top element.
   (b) Evaluate B $\otimes$ A.
   (C) Place the result of (b) back on STACK.

[End of if structure]

[End of step 2 loop]

5. Set VALUE equal to the top element on STACK.
6. Exit.

Example 6.5

Postfix notation P : 5, 6, 2, +, *, 12, 4, /, -)

Symbol Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, **6, 2**
(4) +	**5, 8**
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	**40**, 3
(9) -	37
(10))	

Infix expressions into postfix Expressions

Algorithm 6.4 :

Suppose Q is an arithmetic Expressions written in Infix notation. This algorithm finds the equivalent postfix expression P.

1. Push “(“ onto STACK, and add “)” to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then :
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗
   (b) Add ⊗ to STACK

[End of if structure]

6. If a right parenthesis is encountered, then :
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered
   (b) Remove the left parenthesis

[End of if structure]

[End of step 2 loop]

7. Exit.

After Step 20 is executed, the STACK is empty and

Infix expression Q: A+(B*C-(D/E↑F)*G)*H

P: A B C * D E F ↑ / G * - H * +

First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

A + (B * C - (D / E ↑ F) * G) * H)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20)

Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) -	(+ (-	A B C *
(8) (	(+ (- (	A B C *
(9) D	(+ (- (	A B C * D
(10) /	(+ (- (/	A B C * D
(11) E	(+ (- (/	A B C * D E
(12) ↑	(+ (- (/ ↑	A B C * D E
(13) F	(+ (- (/ ↑	A B C * D E F
(14))	(+ (-	A B C * D E F ↑ /
(15) *	(+ (- *	A B C * D E F ↑ /
(16) G	(+ (- *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * -
(18) *	(+ *	A B C * D E F ↑ / G * -
(19) H	(+ *	A B C * D E F ↑ / G * - H
(20))		A B C * D E F ↑ / G * - H * +

Infix into postfix expression

Infix	Postfix
2 + 3	2 3 +
2 + 3 * 6	3 6 * 2 +
(2 + 3) * 6	2 3 + 6 *
A / (B * C) + D * E - A * C	A B C * / D E * + A C * -

Quicksort

Quicksort is a divide-and-conquer method for sorting.

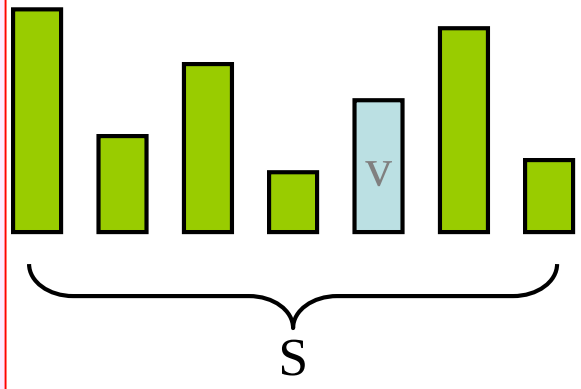
Divide and conquer **method:**

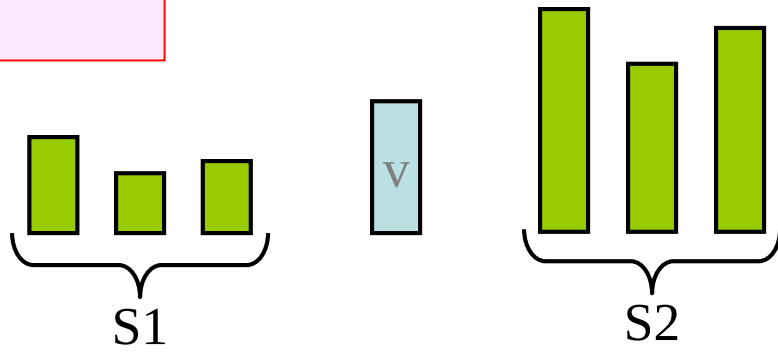
It works by partitioning an array into parts, then sorting each part independently.

Divide: Partition into sub arrays (sub-lists),
Select a splitting element (pivot)
Rearrange the array (sequence/list)

Conquer: Recursively sort 2 sub arrays

Combine : the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)



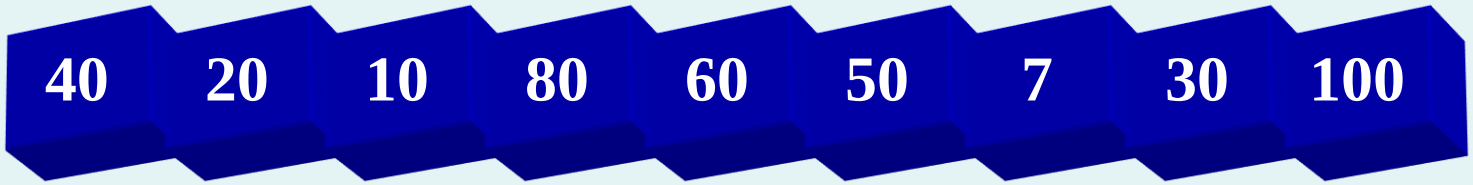


Quicksort

- How do we partition the array efficiently?
  - choose partition element to be leftmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

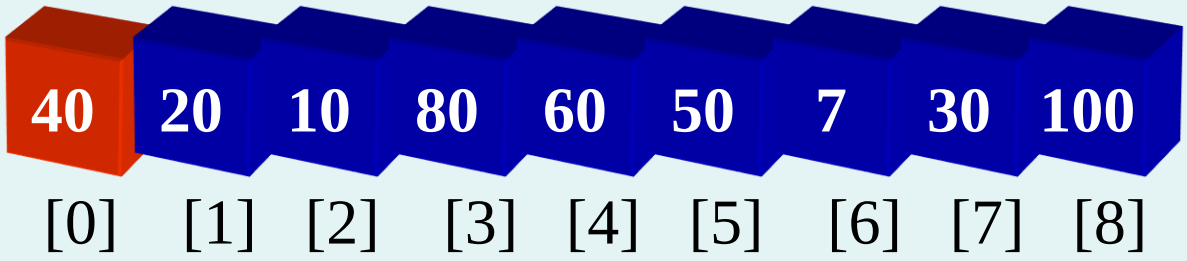
Example

We are given array of n integers to sort:



Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:



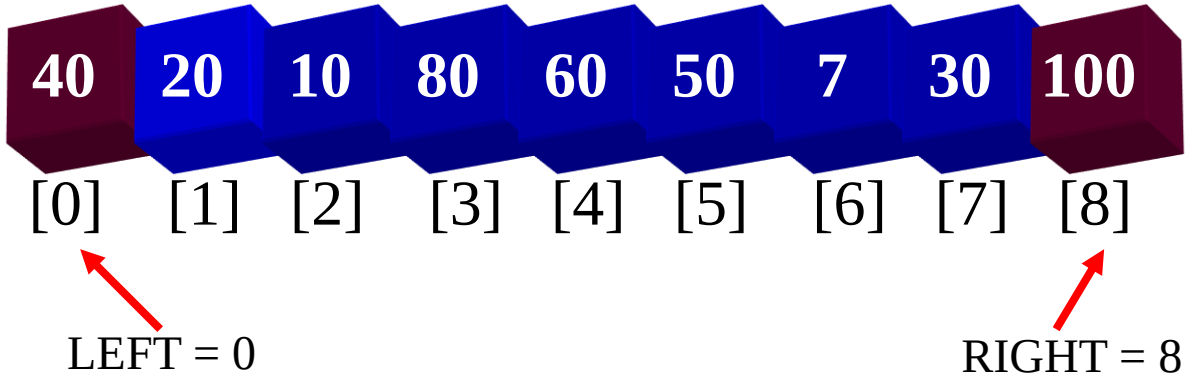
LOC=0 ; LEFT = 0 and RIGHT = 8

1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2

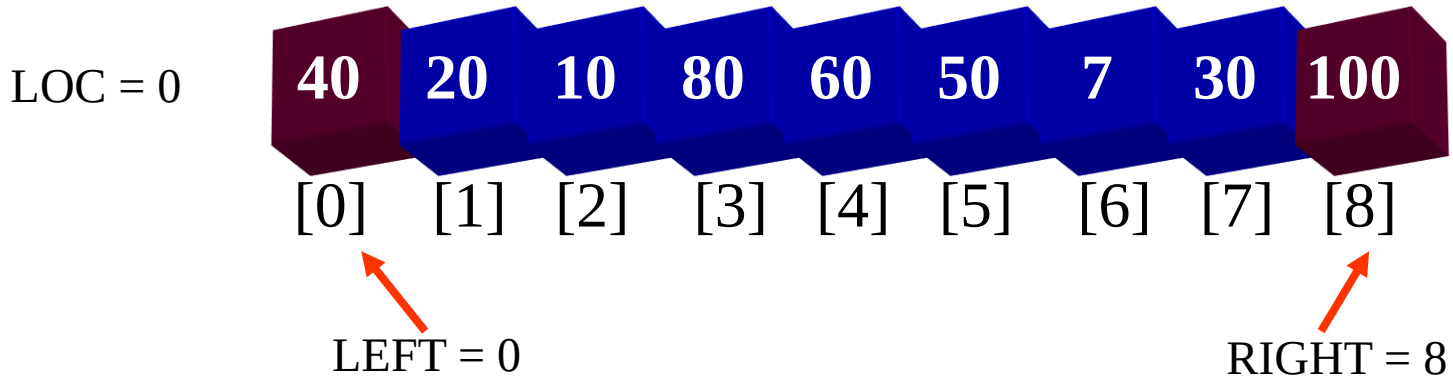
LOC = 0



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

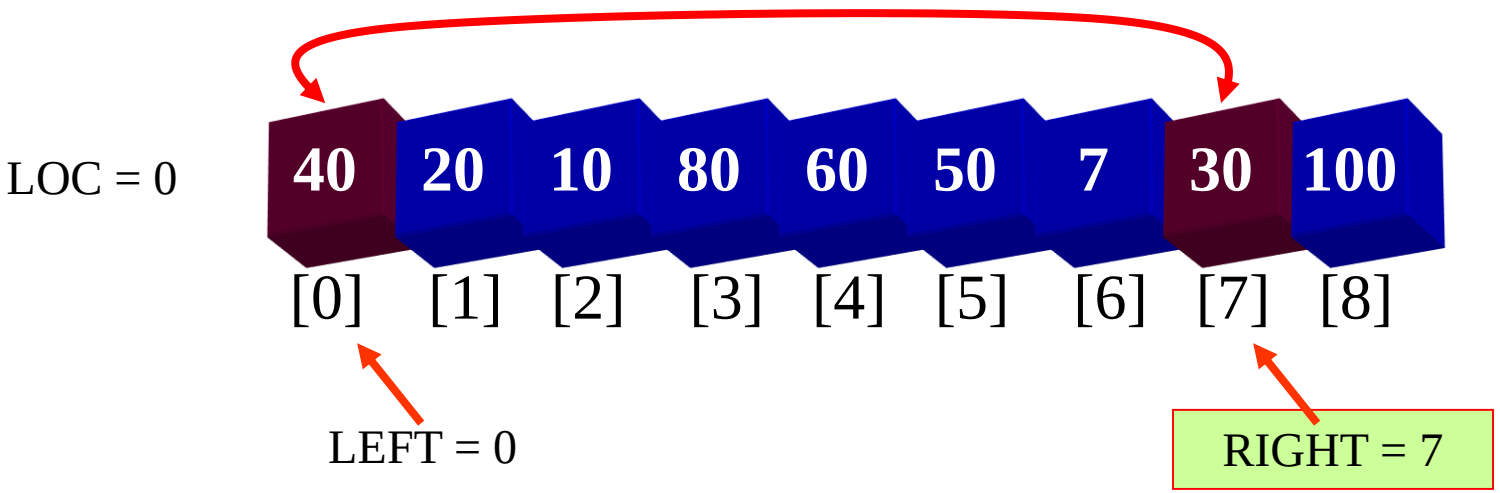
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

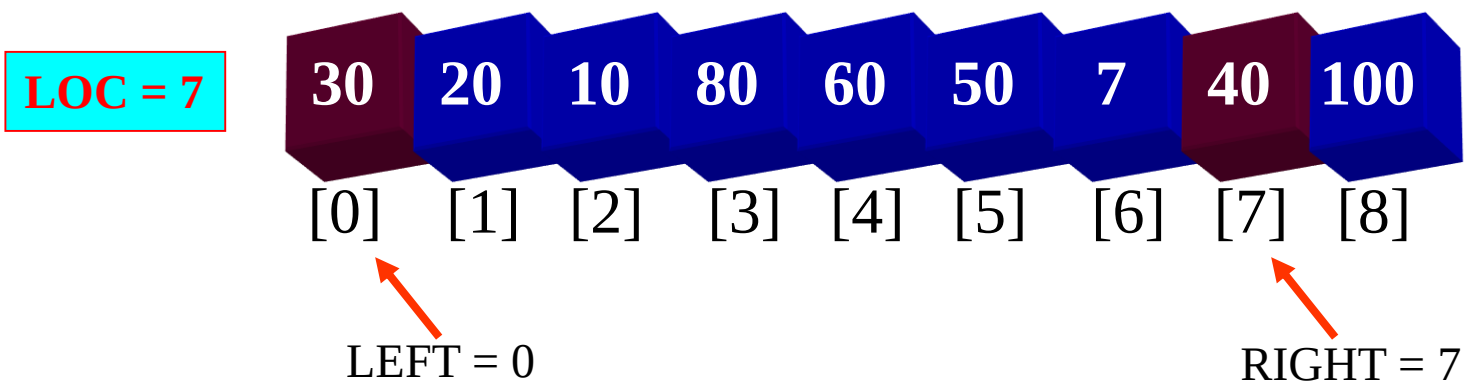
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

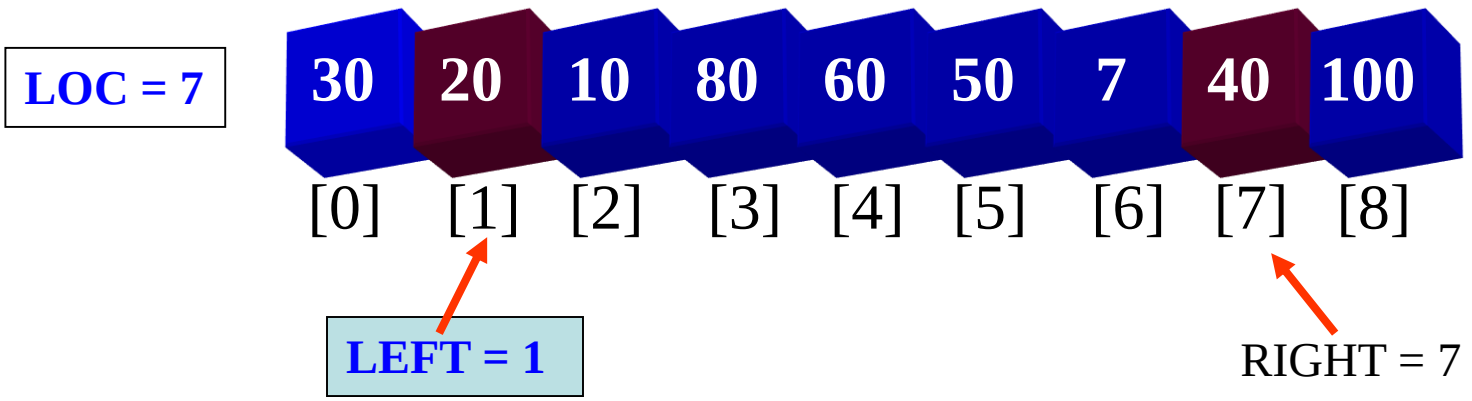
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

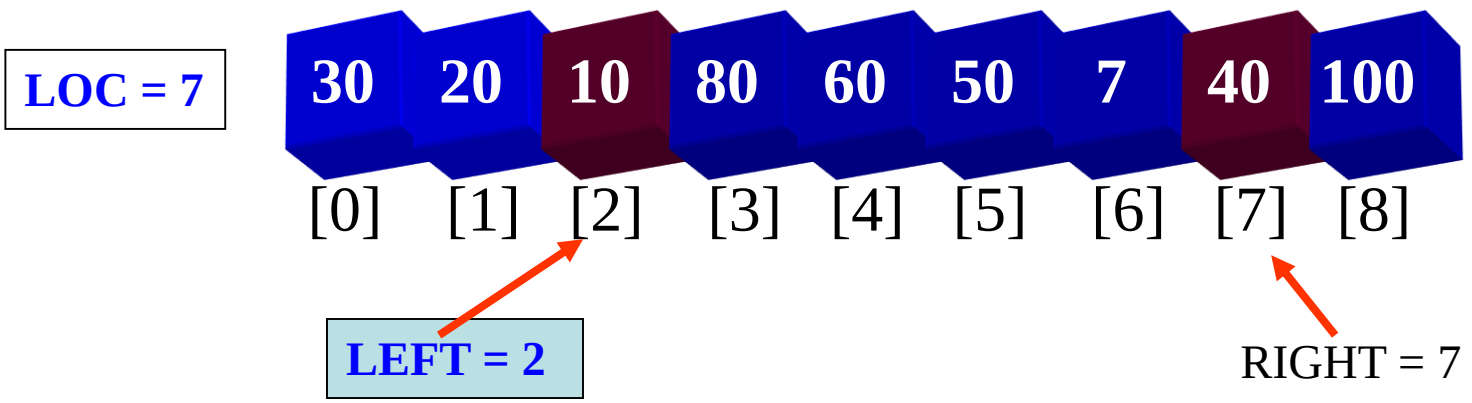
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

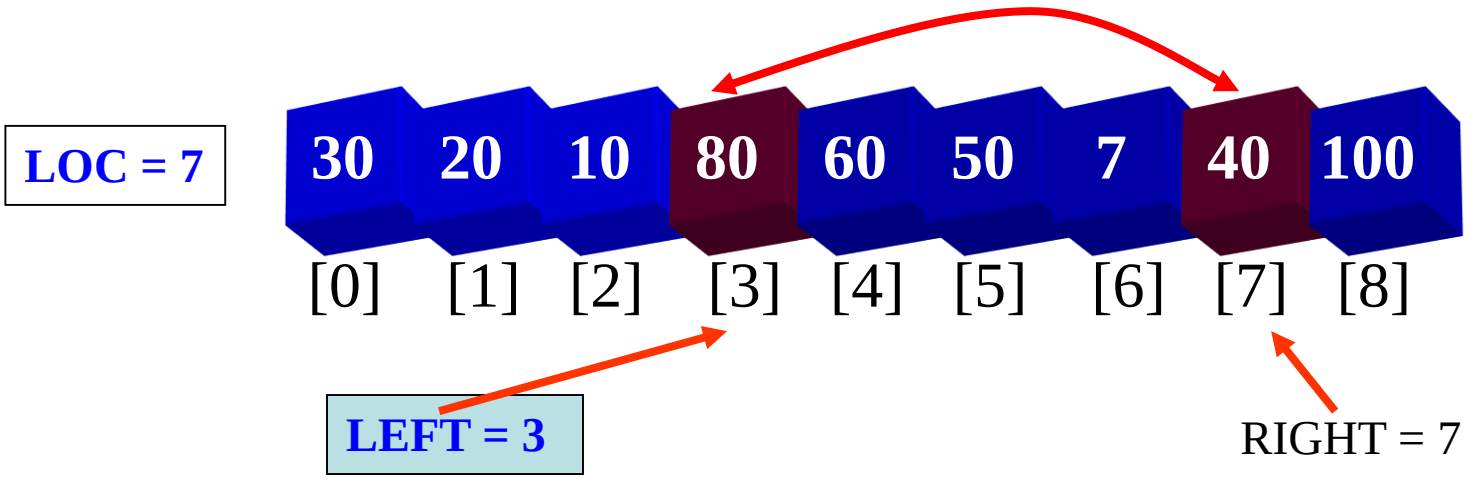
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

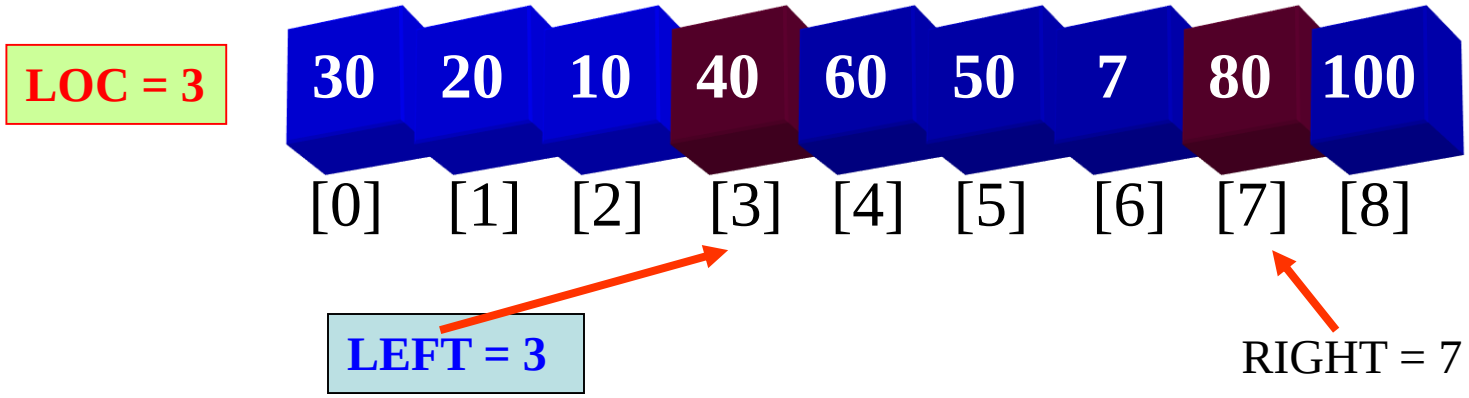
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2

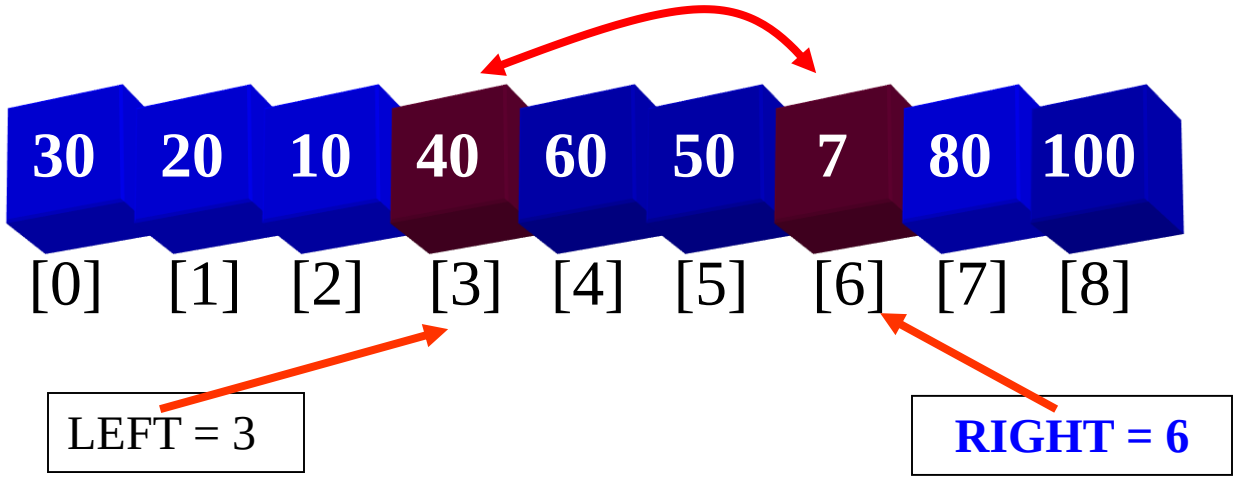


1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2

LOC = 3

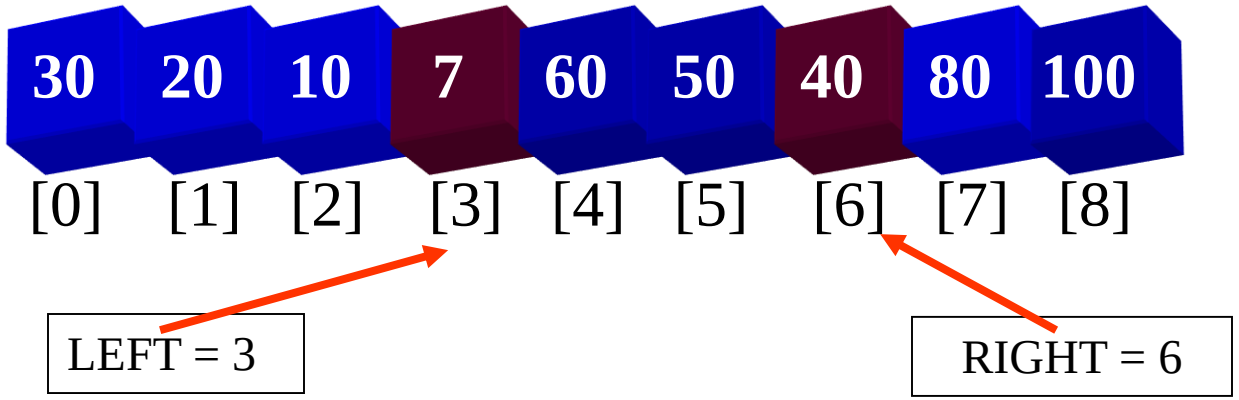


1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2

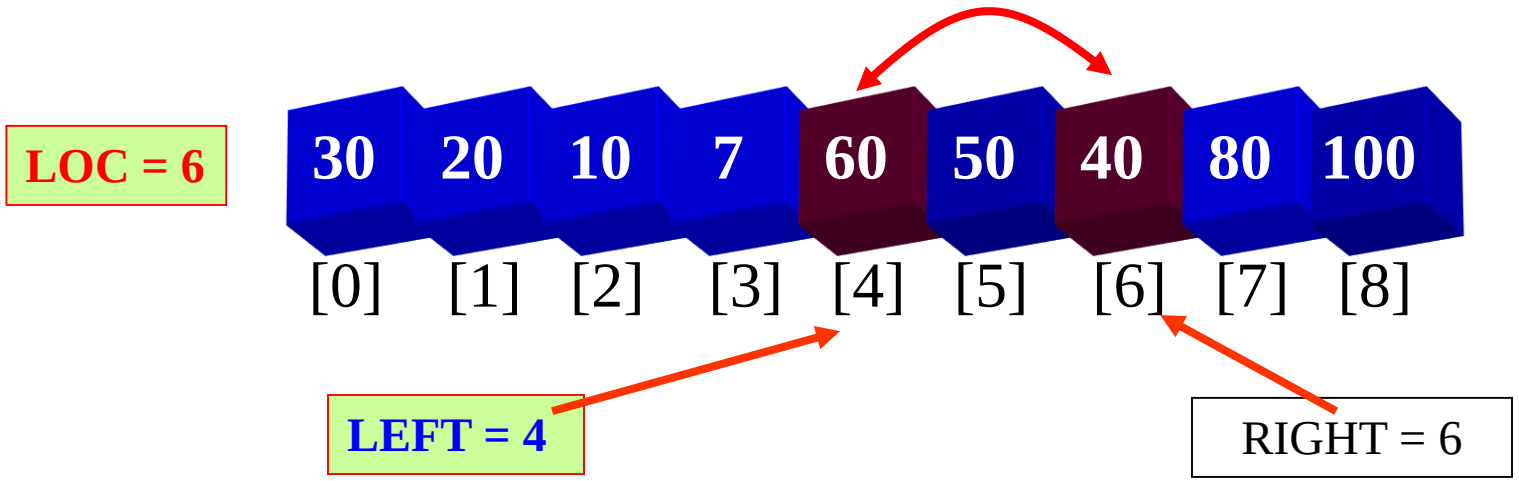
LOC = 6



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

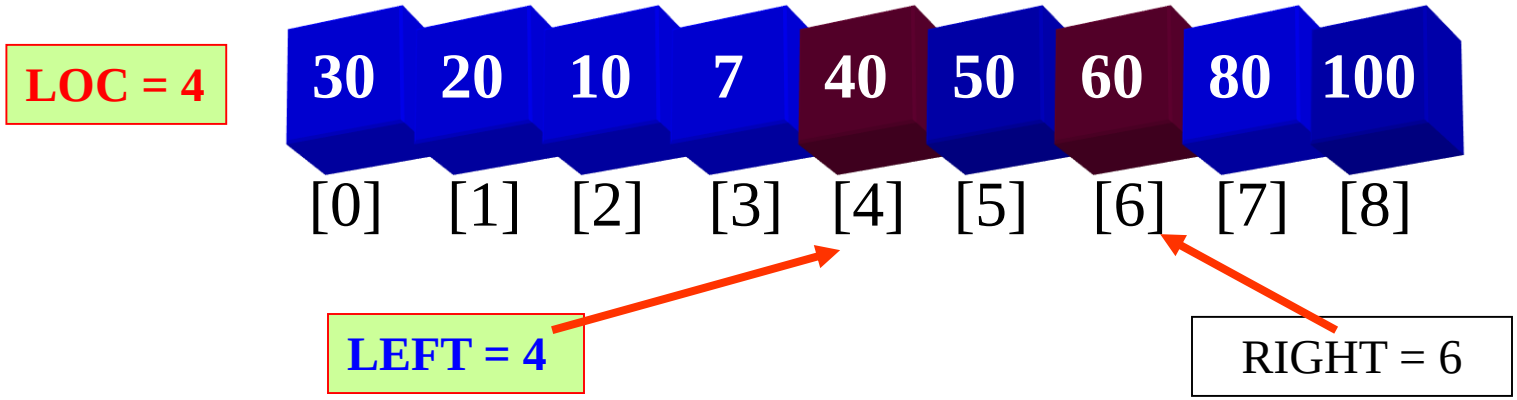
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

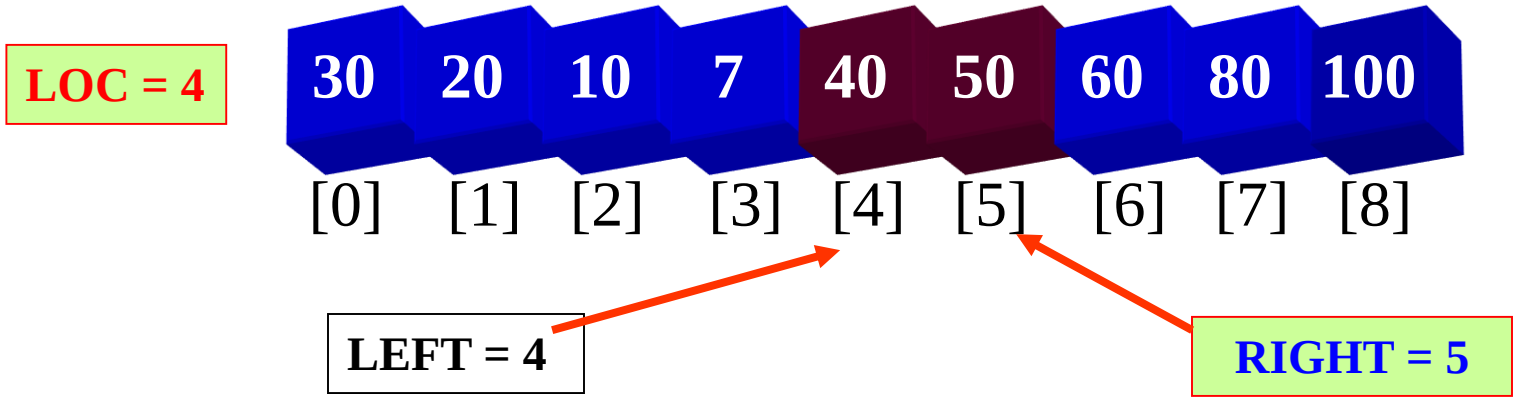
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2



1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

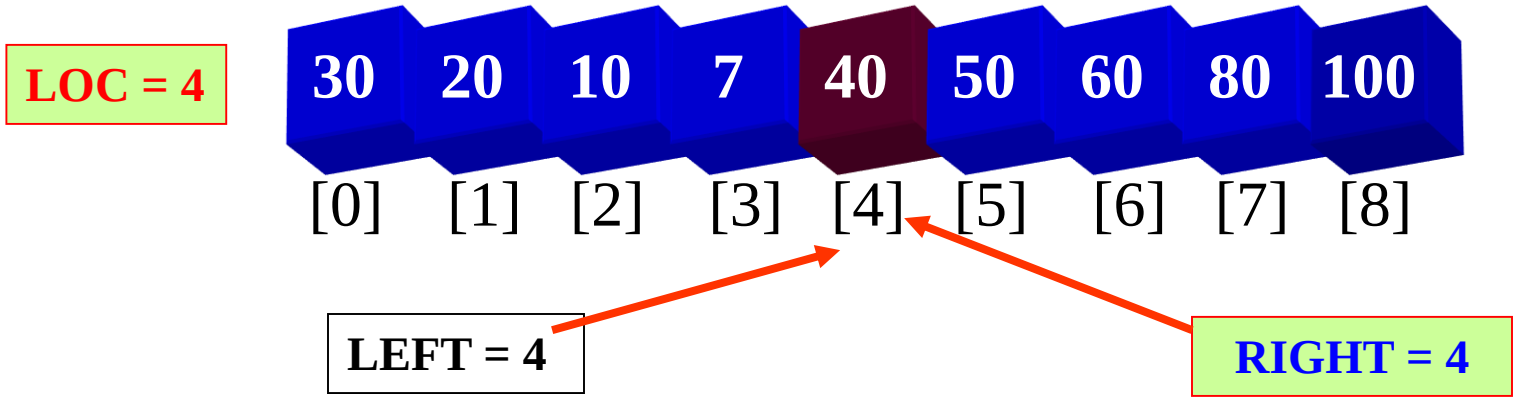
3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2

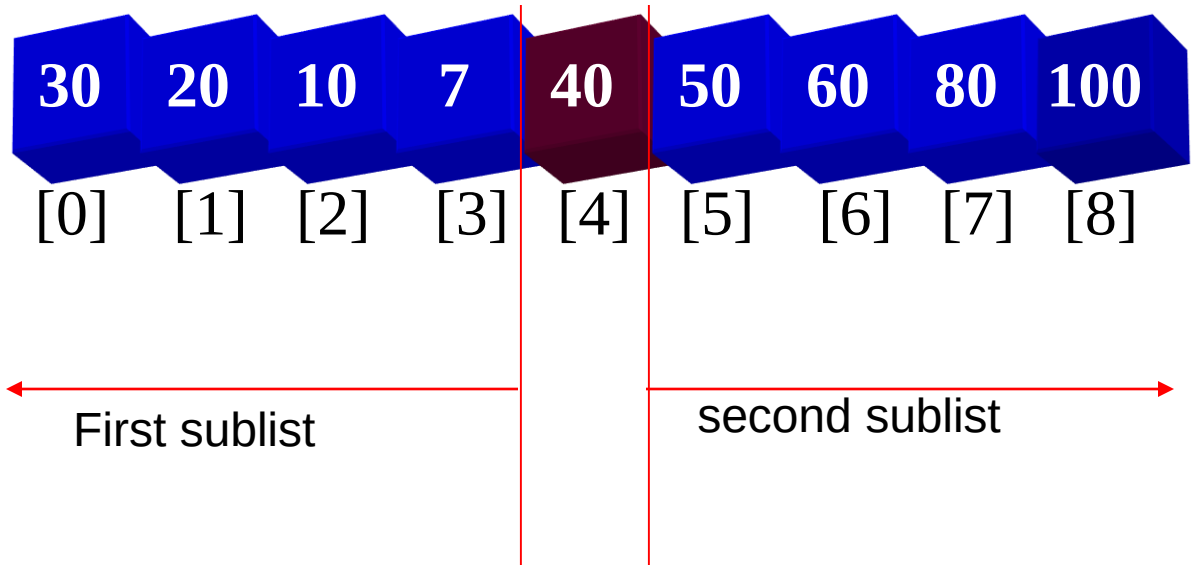


1. Set LEFT = BEG, RIGHT = END and LOC = LEFT

2. (a) Repeat A[LOC] <= A[RIGHT] and LOC ≠ RIGHT
RIGHT := RIGHT -1
(b) if LOC =RIGHT then return
(c) If A[LOC] > A[RIGHT]
(i) interchange each other
(ii) LOC =RIGHT
(iii) Goto Step 3

3. (a) Repeat A[LOC] > A[LEFT] and LOC ≠ LEFT
LEFT := LEFT +1
(b) if LOC = LEFT then return
(c) If A[LEFT] > A[LOC]
(i) interchange each other
(ii) LOC =LEFT
(iii) Goto Step 2





Apply the above procedure repetitively until each sub list contains one element

Example

- Suppose A is the following list of 12 numbers

  44,33,11,55,77,90,40,60,99,22,88,66
- The reduction step of quick sort algorithm finds the final position of one of the numbers; in this we use the first number 44. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. the number is 22. Interchange 44 and 22 to obtain the list.

  22,33,11,55,77,90,40,60,99,44,88,66
- Beginning with 22, next scan the list in opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list.

  22,33,11,44,77,90,40,60,99,55,88,66
- Beginning this time with 55, now scanning the list in original direction that is from right to left, until meeting the first number less than 44.It is 40. Interchange 44 and 40 to obtain the list.

  22,33,11,40,77,90,44,60,99,55,88,66

  22,33,11,40,44,90,77,60,99,55,88,66

Example

- Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such number before meeting 44. This mean all the numbers have been scanned and compared with 44. All the numbers less tan 44 now form the sub list to the left of 44, and all numbers greater than 44 form the sub list to the right of 44 as shown below

  22,33,11,40, 44, 90,77,60,99,55,88,66

  First sublist Second sublist

- Thus 44 is correctly placed in its final position, and the task of sorting the original list A has been reduced to the task of sorting each of the above sub list.
- The above reduction step is repeated with each sub list containing 2 or more elements. Since we can process only one sub list at a time, we must be able o keep track of some sub lists for future processing. This is accomplished by processing two stacks, called LOWER and UPPER, to temporarily old such sub list. That is the address of the first and last elements of each sub list, called its boundary values, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction set is applied to a sub list only after its boundary values are removed from the stack.