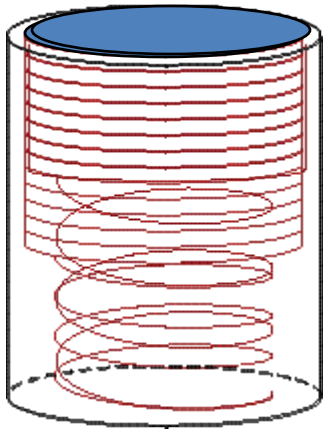
Stack and Queue

Introduction to Stacks

- A stack is a last-in-first-out (LIFO) data structure
- Adding an item
  - Referred to as pushing it onto the stack
- Removing an item
  - Referred to as popping it from the stack

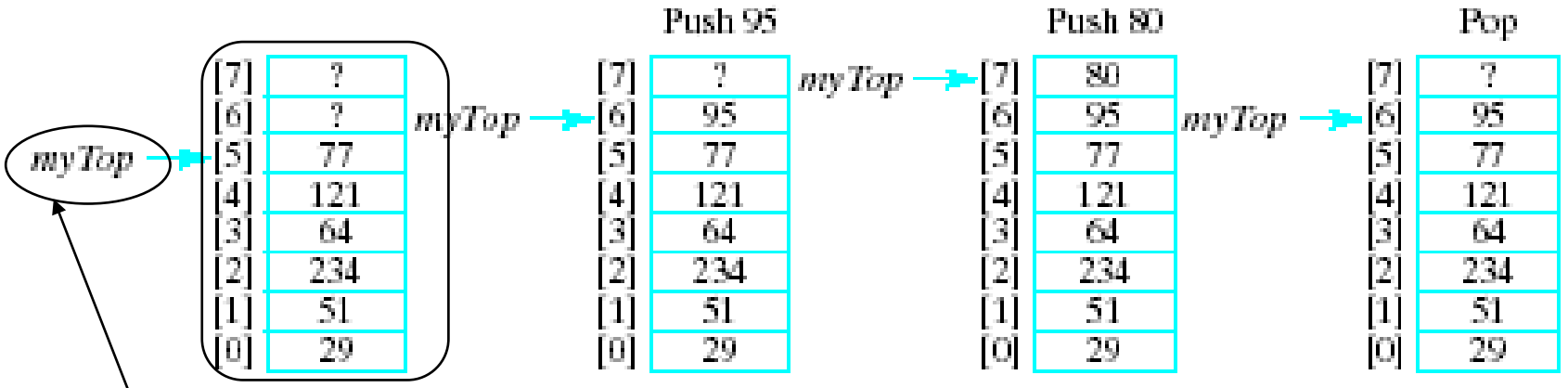


A Stack

- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)
- Operations:
  - construct a stack (usually empty)
  - check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element
  - Pop: remove the top element

Selecting Storage Structure

- Position 0 is the bottom of the stack



- Our design includes:
  - An array to hold the stack elements
  - An integer to indicate the top of the stack

Implementing Operations

- Empty
  - Check if value of `myTop == -1`
- Push (if `myArray` not full)
  - Increment `myTop` by 1
  - Store value in `myArray[myTop]`
- Top
  - If stack not empty, return `myArray[myTop]`
- Pop
  - If array not empty, decrement `myTop`
- Output routine added for testing

STACKS

Algorithm:-1

PUSH(STACK,TOP,MAXSTK,ITEM)

1.**[Stack already filled]**

If TOP=MAXSTK, then

Write: OVERFLOW and Return

[end of if structure]

2. Set TOP:=TOP+1 **[increases TOP by 1]**

3. Set STACK[TOP]:=ITEM **[insert ITEM in new TOP position]**

4. Return

Algorithm:-2

POP(STACK,TOP,ITEM)

1.**[Stack has an item to be removed]**

If TOP=0, then

Write: UNDERFLOW and Return

[end of if structure]

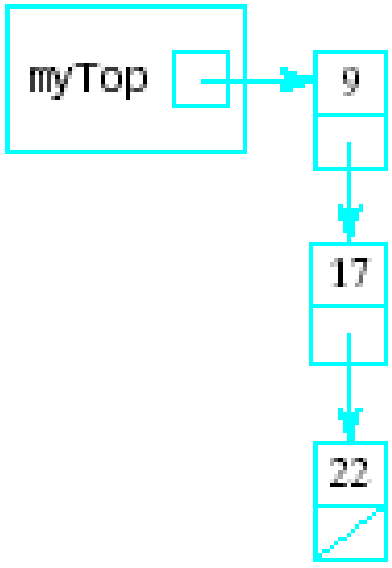
2. Set ITEM:=STACK[TOP] **[Assigns TOP element to ITEM]**

3. Set TOP:=TOP-1 **[Decreases TOP by 1]**

4. Return

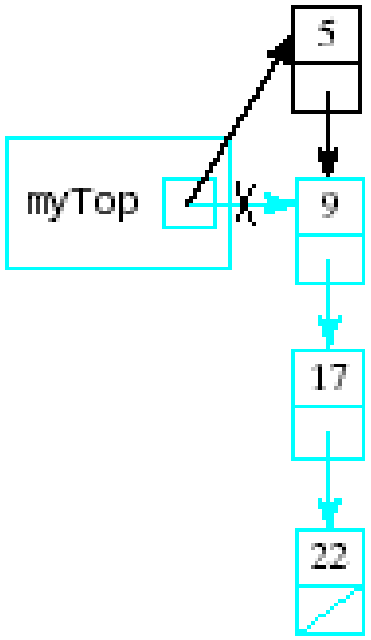
Linked Stacks

- Another alternative to allowing stacks to grow as needed
- Linked list stack needs only one data member
  - Pointer **`myTop`**
  - Nodes allocated (but not part of stack class)



Implementing Linked Stack Operations

- Empty
  - Check for `myTop == null`
- Push
  - Insertion at beginning of list
- Top
  - Return data to which `myTop` points

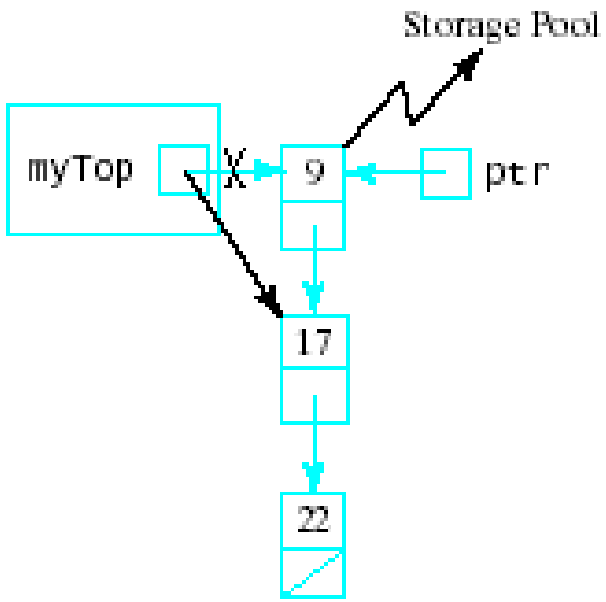


Implementing Linked Stack Operations

- Pop
  - Delete first node in the linked list

```
ptr = myTop;
myTop = myTop->next;
delete ptr;
```

- Output
  - Traverse the list




```
for (ptr = myTop; ptr != 0; ptr = ptr->next)
   cout << ptr->data << endl;
```

Linked Representation of stack

Algorithm:-3

PUSH_LINKSTACK(INFO,LINK,TOP,AVAIL,ITEM)

This procedure pushes an ITEM into a linked stack

1. **[Available Space?]**
   if AVAIL==NULL , then
      Write: OVERFLOW and EXIT
   [end of if structure]
2. **[Remove first node from the avail list]**
   Set NEW = AVAIL and AVAIL = AVAIL→link
3. Set NEW→info = ITEM **[copies ITEM into new node]**
4. Set NEW→link = TOP **[new node points to the original top node in stack]**
5. Set TOP=NEW **[reset TOP to point to new node at the top of stack]**
6. Exit

Algorithm:-4

POP_LINKSTACK(INFO,LINK,TOP,AVAIL,ITEM)

This procedure deletes the top element an ITEM into a linked stack and assign it to the variable ITEM

1. **[stack has an item to be removed?]**
   If TOP==NULL , then
   Write: UNDERFLOW and EXIT
   [end of if structure]
2. Set ITEM = TOP→info **[copies the top element of stack into ITEM]**
3. Set TEMP = TOP and TOP = TOP→link
   [remember the old value of the TOP pointer in TEMP and reset TOP to point to the next element in the stack]
4. Set TEMP→link = AVAIL **[Return deleted node to the AVAIL list]**
5. Set Avail = TEMP
6. Exit

Arithmetic operations; polish notation

There are three notations to represent an expression

- Infix
- Prefix/Polish Notation
- Suffix/Postfix/Reverse Polish Notation

Polish Notation

- In infix notation, the operator symbol is placed between its two operands. For example,

A + B C - D E * F G / H

- In postfix notation, the operator symbol is placed after its two operands. For example,

AB+

- In Prefix notation, the operator symbol is placed before its two operands. For example,

+AB

Arithmetic evaluation

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

- First it converts the expression to postfix notation.
- And then it evaluates the postfix expression.

Priority of binary operations

Highest :	Exponentiation (↑)
Next highest :	Multiplication (*) and division (/)
Lowest :	Addition (+) and subtraction (-)

EVALUATION OF POSTFIX EXPRESSION

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 until the right parenthesis “)” is encountered.
3. If an operand → add to STACK
4. If an operator (x) is encountered then,
   a) Remove the top two elements of STACK, where A is the top element and B is the next-to-top element.
   b) Evaluate B (x) A
   c) Place the result of (b) back on STACK
5. Set VALUE equal to the top element on STACK.
6. Exit.

example

Consider the following arithmetic expression P written in postfix notation:

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

Q: 5 * (6 + 2) - 12 / 4

We evaluate P using *algorithm 1.3*. First we add a sentinel right parenthesis at the end of P to obtain

P:	5,	6,	2,	+,	*,	12,	4,	/,	-,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

Figure 1.15 shows the contents of STACK as each element of P is scanned. The final number in STACK, 37, which is assigned to VALUE when the sentinel ")" is scanned, is the value of P.

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)	)	

Figure 1.15

Evaluate the postfix expression

P: 20, 2, *,9, +, 14, 7, /, -, 5, 3, *, +

Ans: 62

STEPS FOR CONVERTING INFIX TO POSTFIX

Suppose Q → is an infix notation.

P→ is an equivalent postfix notation.

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 → add to P
4. If an operator(x) ,then,
   i. Repeatedly pop from STACK and add to P each operator which has the same precedence as or higher precedence than (X)
   ii. Add (x) to STACK
5. If a left parenthesis ‘(’→ push it onto STACK.
6. If a right parenthesis ‘)‘ then
   i. Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered.
   ii. Remove the left parenthesis ‘(’.

Infix to postfix conversion

Q= A + (B * C - (D / E ↑ F) * G) * H

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 * +

Examples

INFIX	POSTFIX
A+B-C	AB+C-
A*B+C/D	AB*CD/+
A*B+C	AB*C+
A-B/(C*D^E)	ABCDE^*/-
(A+B)*C	AB+C*
(A-B)*(C/D)+E	AB-CD/*E+
A-B/C+D^E	ABC/DE^+

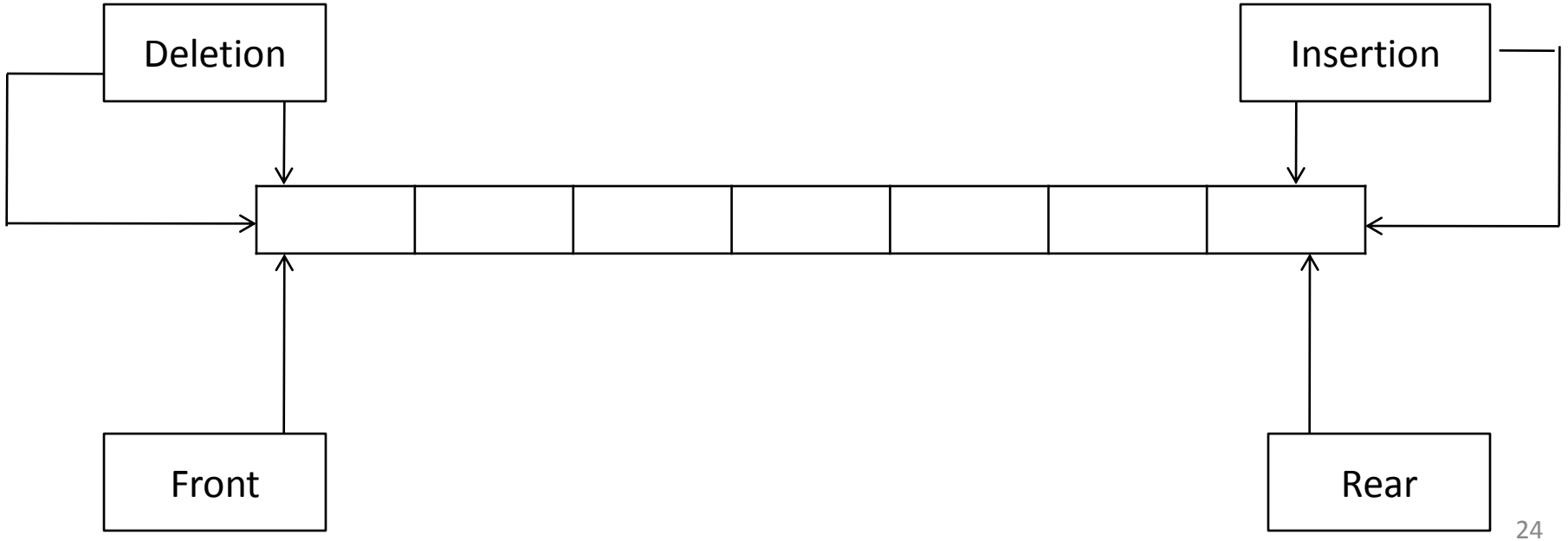
Infix Expression	Equivalent Postfix Expression
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c\ *\ +$
$a * b + c$	$a\ b\ *\ c\ +$
$(a + b) * c$	$a\ b\ +\ c\ *$
$(a - b) * (c + d)$	$a\ b\ -\ c\ d\ +\ *$
$(a + b) * (c - d / e) + f$	$a\ b\ +\ c\ d\ e\ /\ -\ *\ f\ +$

Applications of stack

1. To reverse a sequence or string.
2. To check whether a string is a palindrome or not.
3. To convert an infix expression to its postfix equivalent.
4. To convert an infix expression to its prefix equivalent.
5. To evaluate one postfix expression.

Queues

A queue is a linear list of elements in which deletions can take place only at one end called the FRONT, and insertions can take place only other end called the REAR.

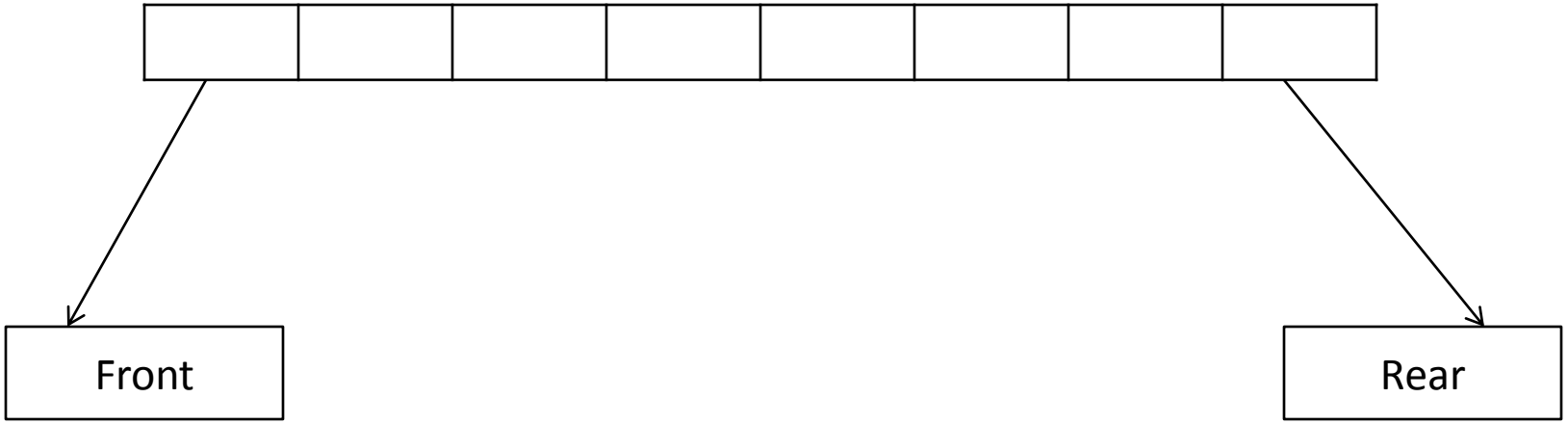


Applications and characteristics of Queues

- **Characteristics:**
  - Like waiting in line
  - Nodes are added to the rear
  - Nodes are removed from the front
  - First-in, first-out (FIFO) data structure
  - Insertion is called **ENQUEUE**
  - Removal is called **DEQUEUE**
- **Applications**
  - Print spooling
    - Documents wait in queue until printer available
  - Packets on network
  - File requests from server

Queue Implementation

- Static Implementation – done with the array
- Dynamic Implementation – done using linked list. (pointers)



Using Arrays

Declaration of a Queue

```
# define MAXQUEUE 50 /* size of the queue
   items*/
typedef struct {
   int front;
   int rear;
   int items[MAXQUEUE];
}QUEUE;
```

Queue Implementation As Array

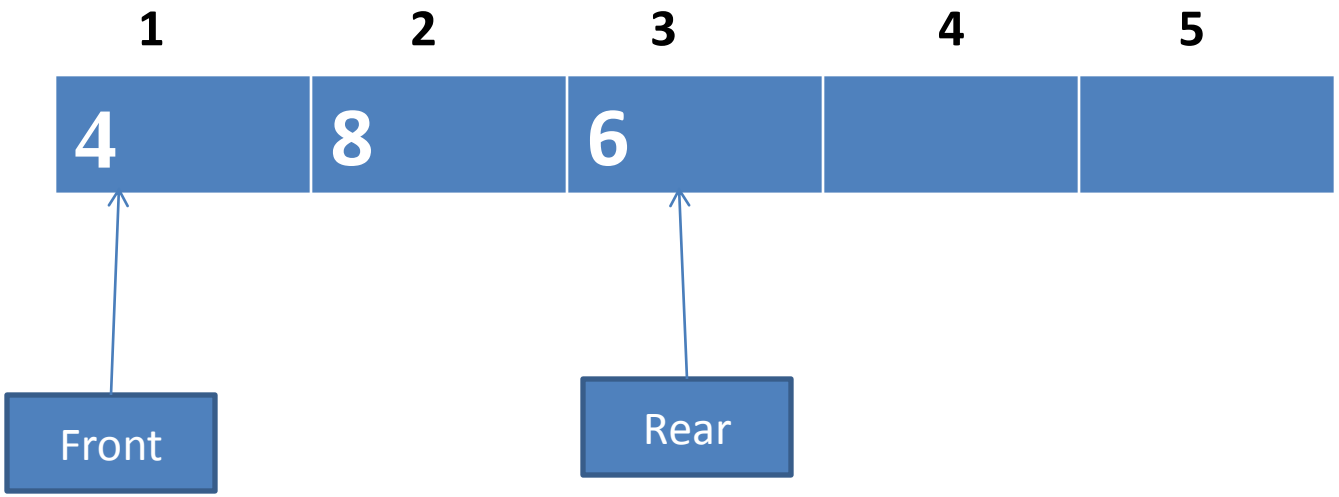
- There is the need of one array called as **Queue** and two pointer variables **Front** and **Rear,** containing the location of the Front element and Rear element of the queue.
- Whenever an element is deleted from the queue the value of Front is incremented by 1

  Front=Front+1
- Similarly, Whenever an element is added to the queue the value of rear is incremented by 1

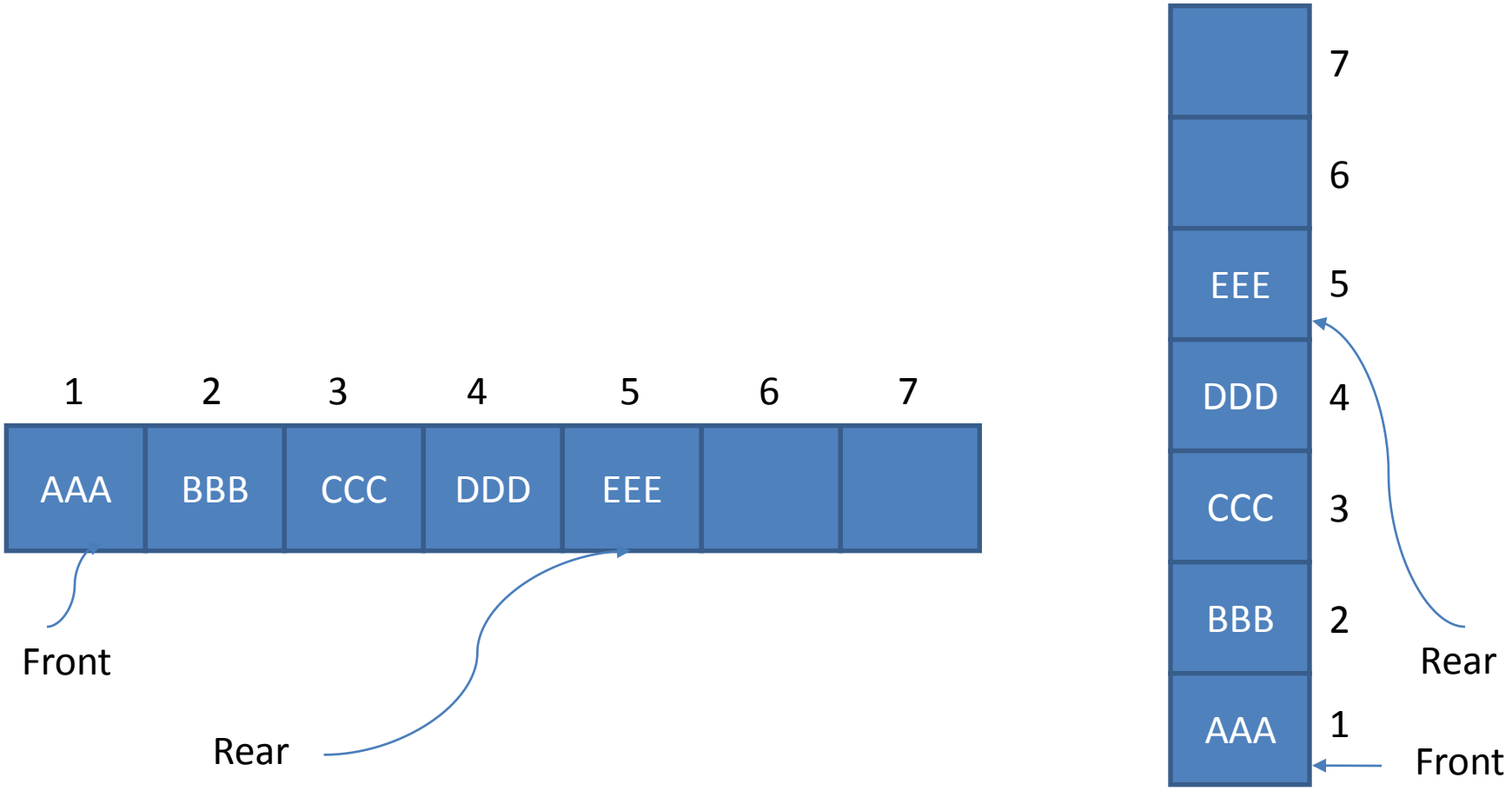
  Rear =Rear+1
- Set FRONT = 0 and REAR = 0, queue is empty.

Queue Implementation As Array

- A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



Queue Implementation As Array



Operations On Linear Queues

- Insertion – enqueue
- Deletion – dequeue

Insertion into linear queue

PROCEDURE ENQUEUE(QUEUE, Front, Rear, N, item)

1. [Overflow ?]
if (Rear = = N) then write Queue is full and return.

2. If (Front==0 and Rear == 0) Then
    Set Front = 1 and Set Rear = 1
  Else
    Set Rear = Rear + 1 [Increment Rear by 1]
 [End of Step 2 If]

3. QUEUE[Rear] = ITEM

4. Return

Deletion from linear queue

PROCEDURE DEQUEUE(QUEUE, Front, Rear, item)

1. [Underflow ?]
   if (Front = = 0) then write Queue is empty and return.

2. ITEM = QUEUE[Front]

3. If (Front == Rear) Then
   Set Front = 0 and Set Rear = 0
   Else
   Set Front = Front + 1
   [End of Step 3 If]

4. Return

Example: Consider the following queue (linear queue).

Rear = 4 and Front = 1 and N = 7

10	50	30	40			
1	2	3	4	5	6	7

(1) Insert 20. Now Rear = 5 and Front = 1

10	50	30	40	20		
1	2	3	4	5	6	7

(2) Delete Front Element. Now Rear = 5 and Front = 2

	50	30	40	20		
1	2	3	4	5	6	7

(3) Delete Front Element. Now Rear = 5 and Front = 3

		30	40	20		
1	2	3	4	5	6	7

(4) Insert 60. Now Rear = 6 and Front = 3

		30	40	20	60	
1	2	3	4	5	6	7

Drawback of Linear Queue

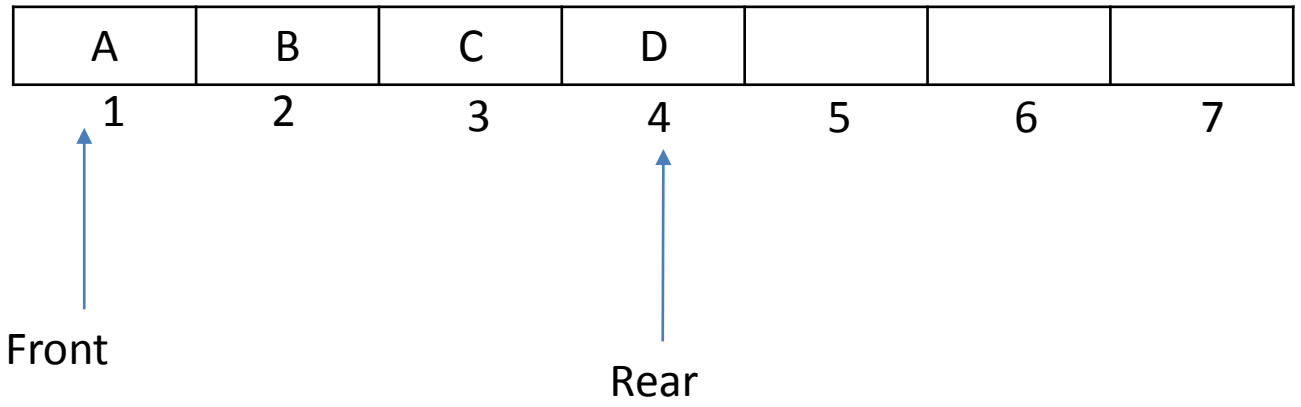
Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the Queue's rear most position.

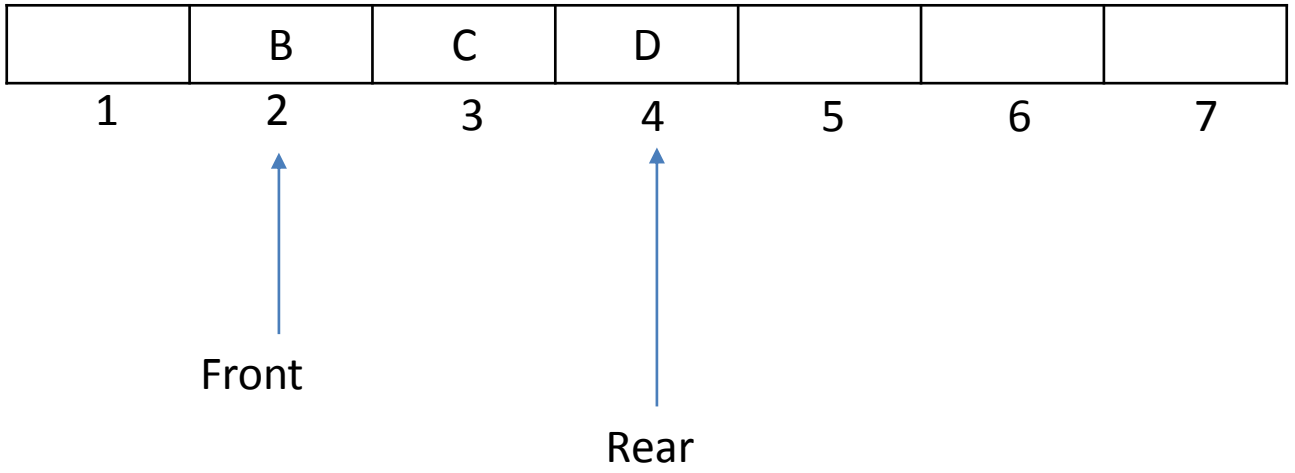
Insertion [Linked list representation]

1. If AVAIL==NULL then
   Write: OVERFLOW and EXIT
   [end of if structure]
2. Set NEW = AVAIL and AVAIL = AVAIL→link
3. Set NEW→info = ITEM and NEW→link = NULL
4. If FRONT==NULL then **[then queue initially empty]**
   FRONT=REAR=NEW
   else
   Set REAR→link = NEW and REAR=NEW
   [end of if structure]
5. Exit

Delete A

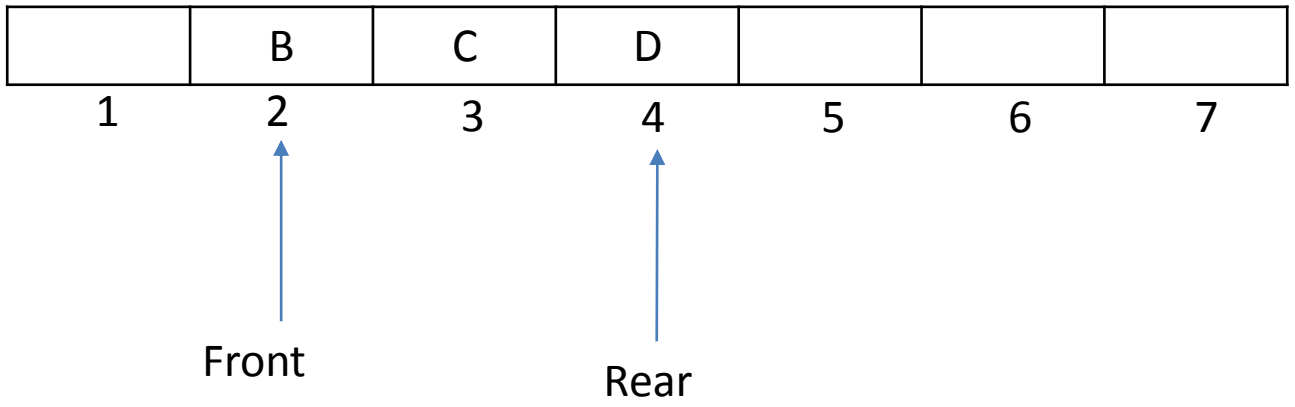
- **An item (*A*) is deleted from the *Front* of the queue**

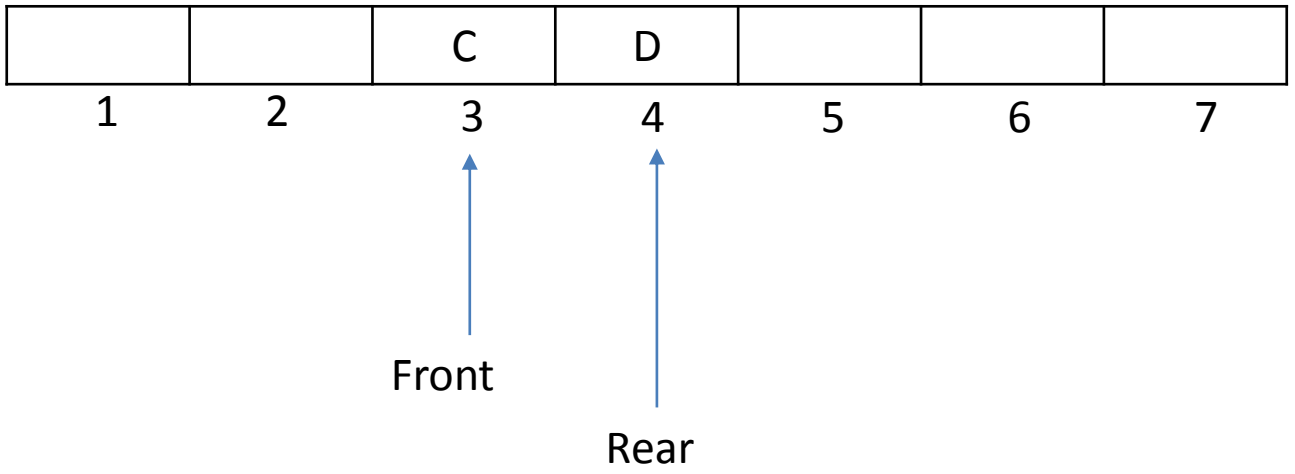




Delete B

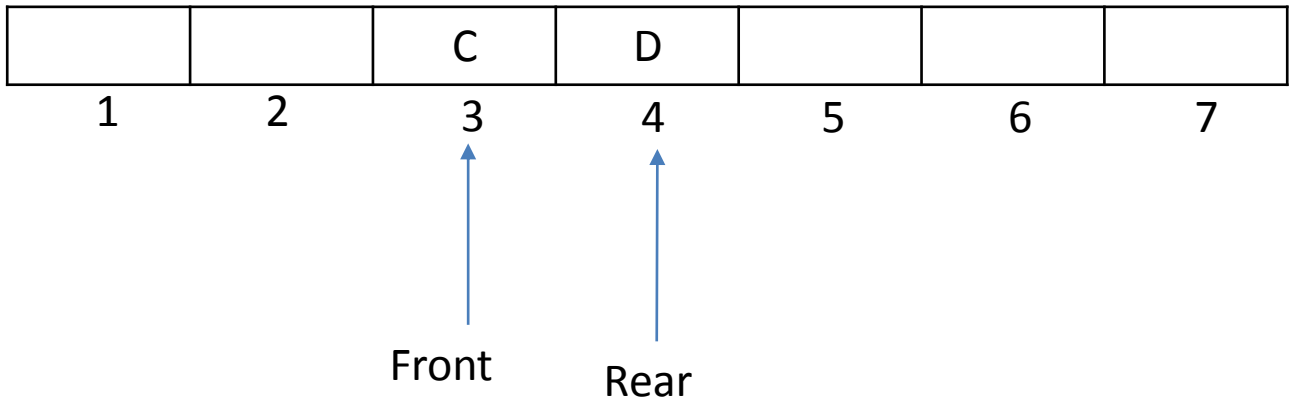
- **An item (*B*) is deleted from the *Front* of the queue.**

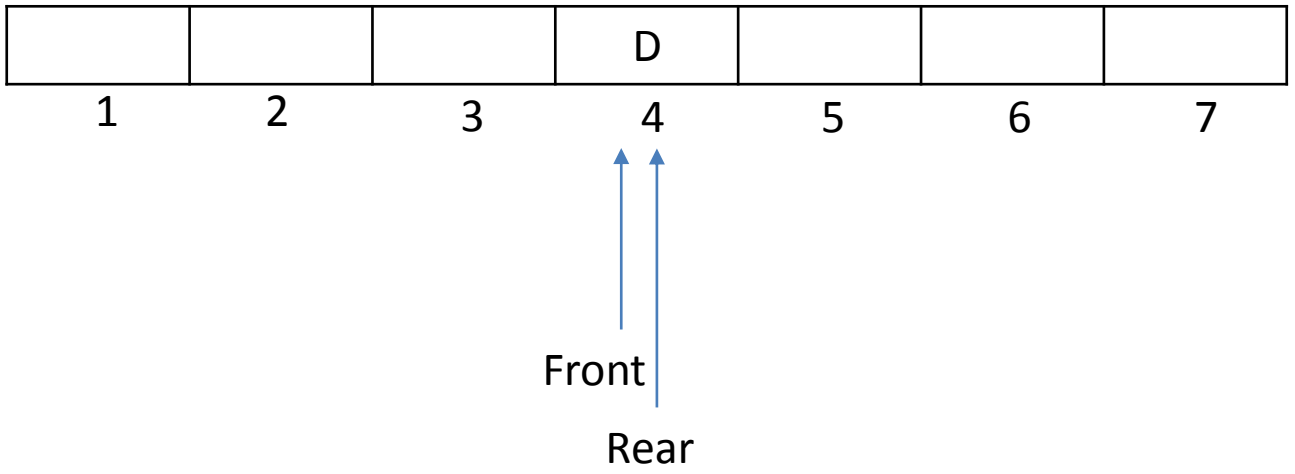




Delete B

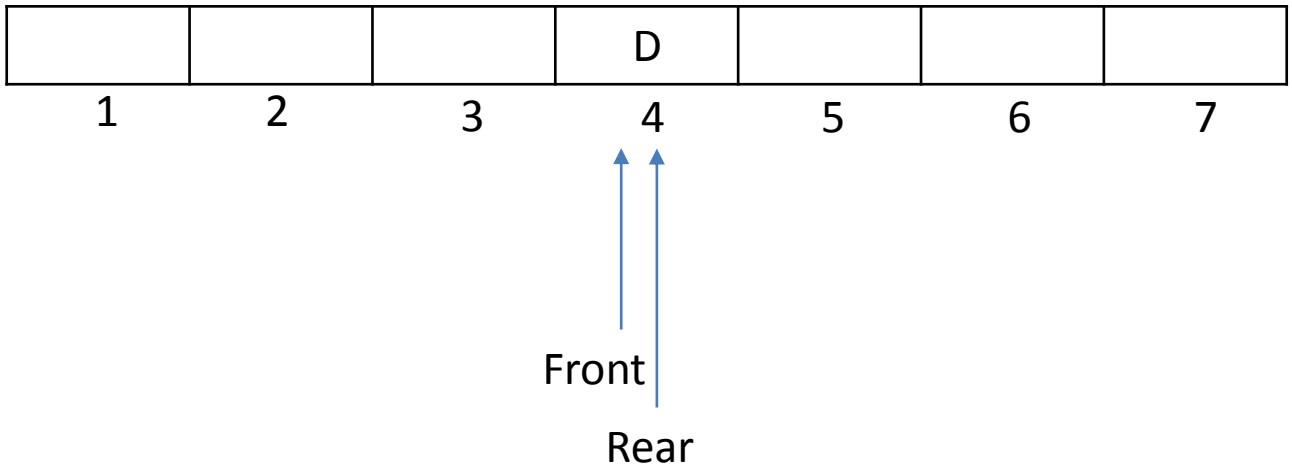
- **An item (*C*) is deleted from the *Front* of the queue.**

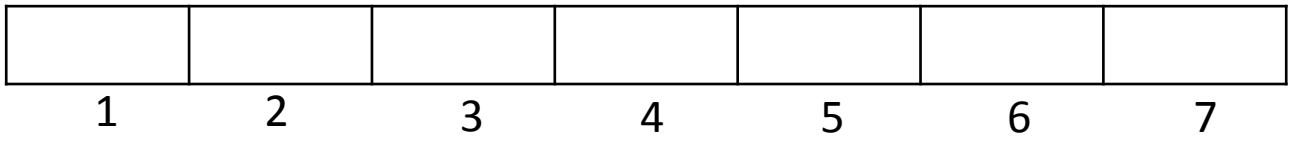




Delete B

- **An item (*D*) is deleted from the *Front* of the queue.**





Front = 0

Rear = 0

QDELETE(QUEUE,N,FRONT,REAR,ITEM)

This procedure deletes an element from a queue and assign it to the variable item

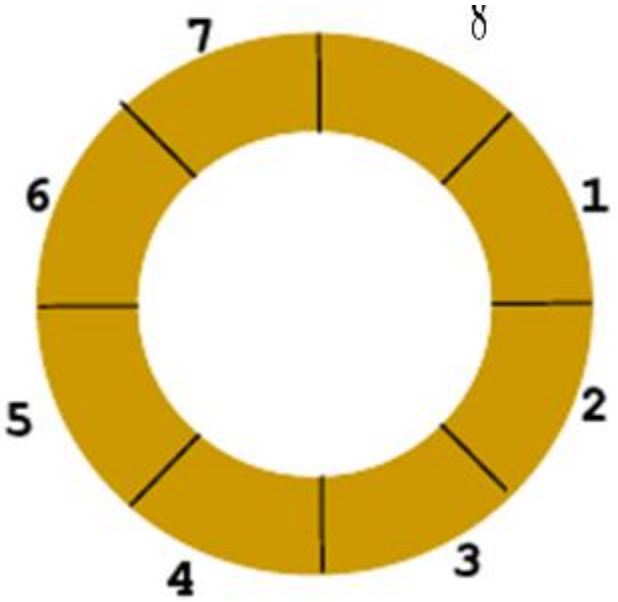
1) **[queue already empty?]**
    If FRONT == 0 Then
        Write: UNDERFLOW and return
   [end of if structure]
2) Set ITEM = QUEUE[FRONT]
3) **[find new value of FRONT]**
    if FRONT==REAR , then: **[queue has only one element]**
        Set FRONT = 0 and REAR = 0
    else if FRONT==N, then **[if Front is pointing at end of array]**
        Set FRONT = FRONT+1
    [end of if structure]
4) RETURN

Deletion [Linked list representation]

1. If FRONT==NULL then
   Write: UNDERFLOW and EXIT
   [end of if structure]
2. Set TEMP = FRONT
3. ITEM = TEMP→info
4. If FRONT == REAR **[queue has only one element]**
   Front = NULL and Rear = NULL
   else
   Front = Front→link
   [end of if structure]
5. TEMP→link = AVAIL and AVAIL = TEMP
6. Exit

Circular Queue

- The idea of the circular array is that the end of the array “wraps around” to the start of the array.

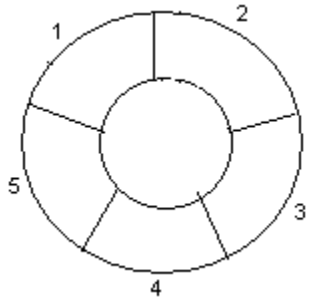


Circular Queue

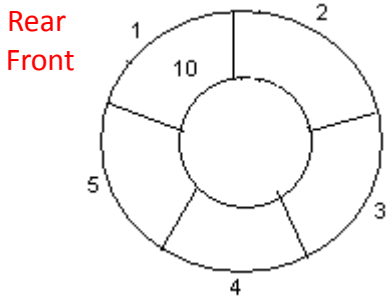
- In circular queue, once the Queue is full the "First" element of the Queue becomes the "Rear" most element, if and only if the "Front“ has moved forward, otherwise it will again be a "Queue overflow" state.
- Initially Rear = 0 and Front = 0.

Example: Consider the following circular queue with N = 5.

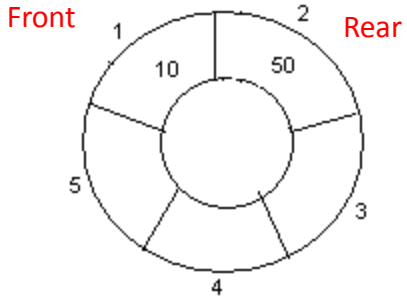
1. Initially, Rear = 0, Front = 0.



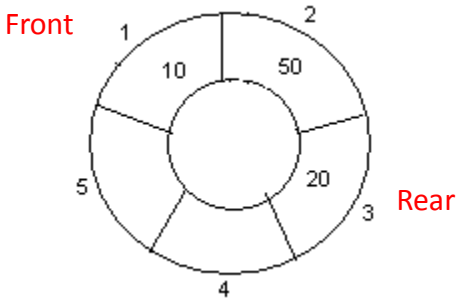
2. Insert 10, Rear = 1, Front = 1.



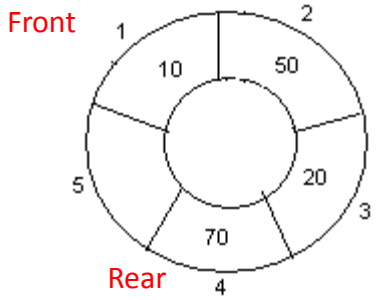
3. Insert 50, Rear = 2, Front = 1.



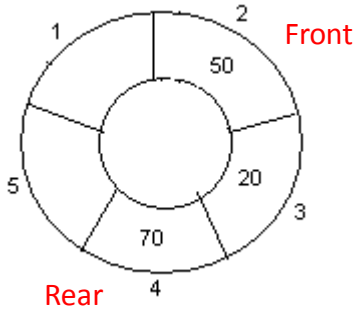
4. Insert 20, Rear = 3, Front = 1.



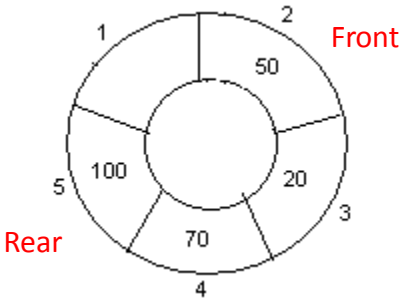
5. Insert 70, Rear = 4, Front = 1.



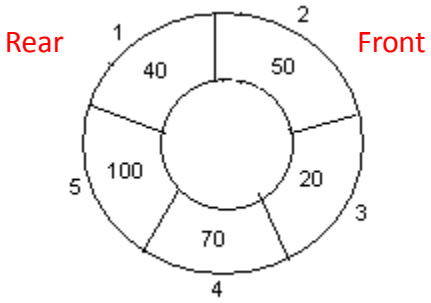
6. Delete front, Rear = 4, Front = 2.



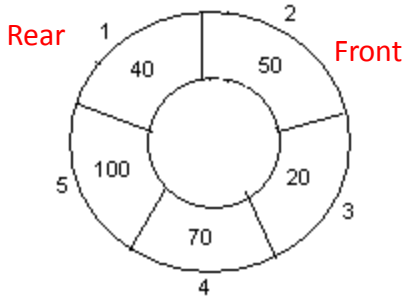
7. Insert 100, Rear = 5, Front = 2.



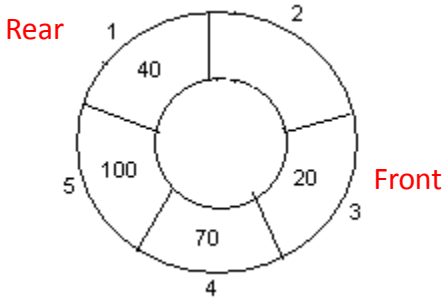
8. Insert 40, Rear = 1, Front = 2.



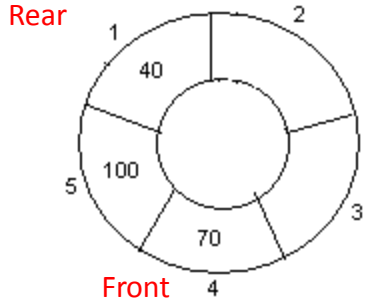
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.

Rear

Front

Insertion into circular queue

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

1. If (Front =1 and Rear = N) or Front = Rear + 1 then Print: “Circular Queue Overflow” and Return.

2. If (Rear == 0) Then [Check if QUEUE is empty]
   Set Front = 1 and Set Rear = 1
   Else If (Rear == N) Then [If Rear reaches end if QUEUE]
   Set Rear = 1
   Else
   Set Rear = Rear + 1 [Increment Rear by 1]
   [End of Step 2 If]

3. Set QUEUE[Rear] = Item

4. Return

DELETION FROM CIRCULAR QUEUE

Delete-Circular-Q(CQueue, Front, Rear, Item)

1. If Front = 0 then Print: “Circular Queue Underflow” and Return.

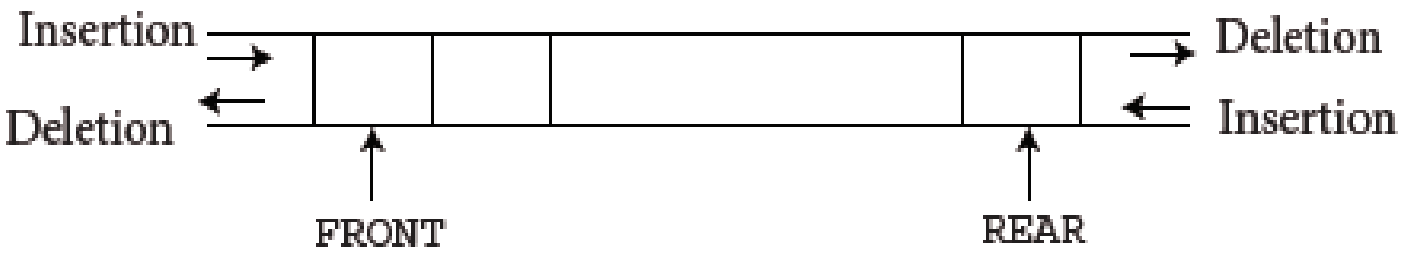
2. Item = QUEUE[Front]

3. If (Front == Rear) Then
   Set Front = 0 and Set Rear = 0
   Else If (Front == N) Then
   Set Front = 1
   Else
   Set Front = Front + 1
   [End of Step 3 If]

4. Return

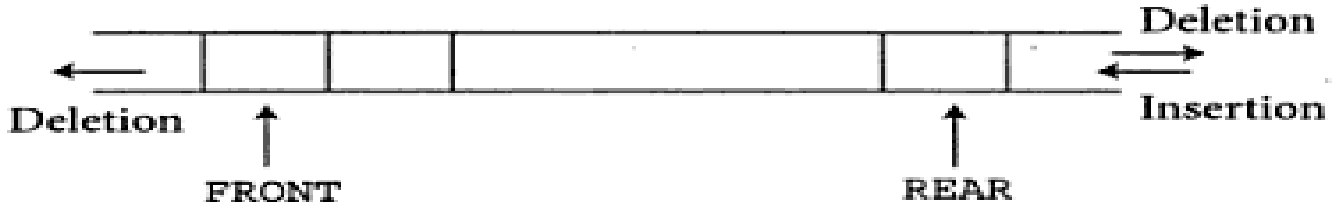
Deque

- A **deque** is a double-ended queue.
- A deque is a linear list in which elements can be added or removed at either end but not in the middle.
- The deque is maintained by a circular array deque with pointers left and right, which point to the two ends of the deque.

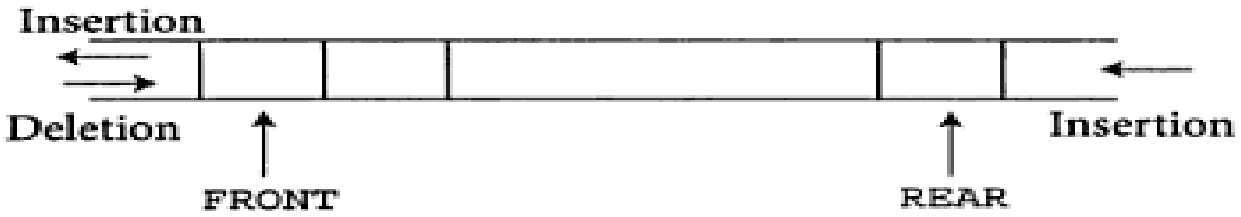


Types of deque

- There are two variations of a deque
  - Input-restricted deque: An input restricted deque is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list
  - Output restricted deque: An output-restricted deque is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.



(a) Input restricted deque



(b) Output restricted deque

Priority Queues

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Priority queues characteristics

- Like an ordinary queue
- Has a front and a rear
- Elements are removed from front only *(dequeue)*
- When an item is inserted (*enqueued)* the order must be maintained
- Elements are ordered so that the item with highest priority is always in front
- Said to be an **ascending-priority queue** if the item with smallest key has the highest priority
- Said to be a **descending-priority queue** if the item with largest key has the highest priority

Applications of priority queue

- Multiuser system
- Bandwidth management
- *discrete event simulation*
- Dijkstra's algorithm

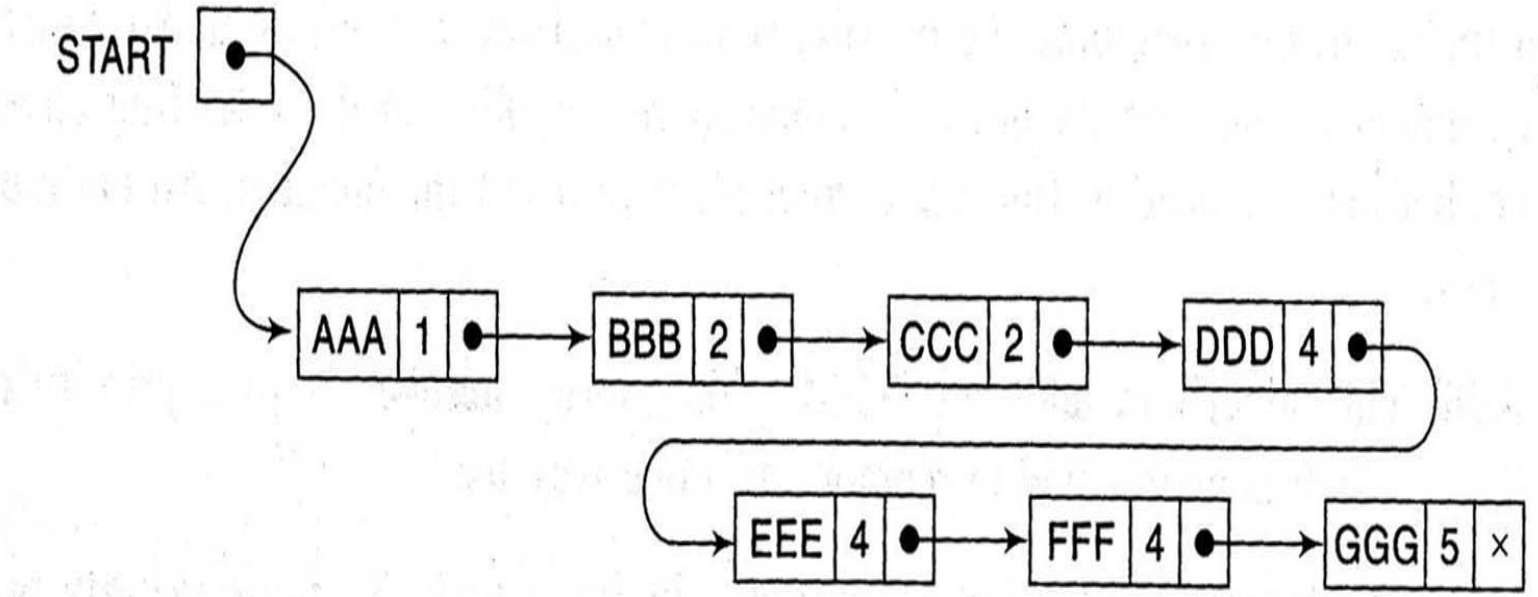
Maintaining a priority queue

- There are various ways of maintaining a priority queue in memory.
  - One-way list (LINKED LIST)
  - Multiple queues (ARRAY REPRESENTATION OF PRORITY QUEUE)

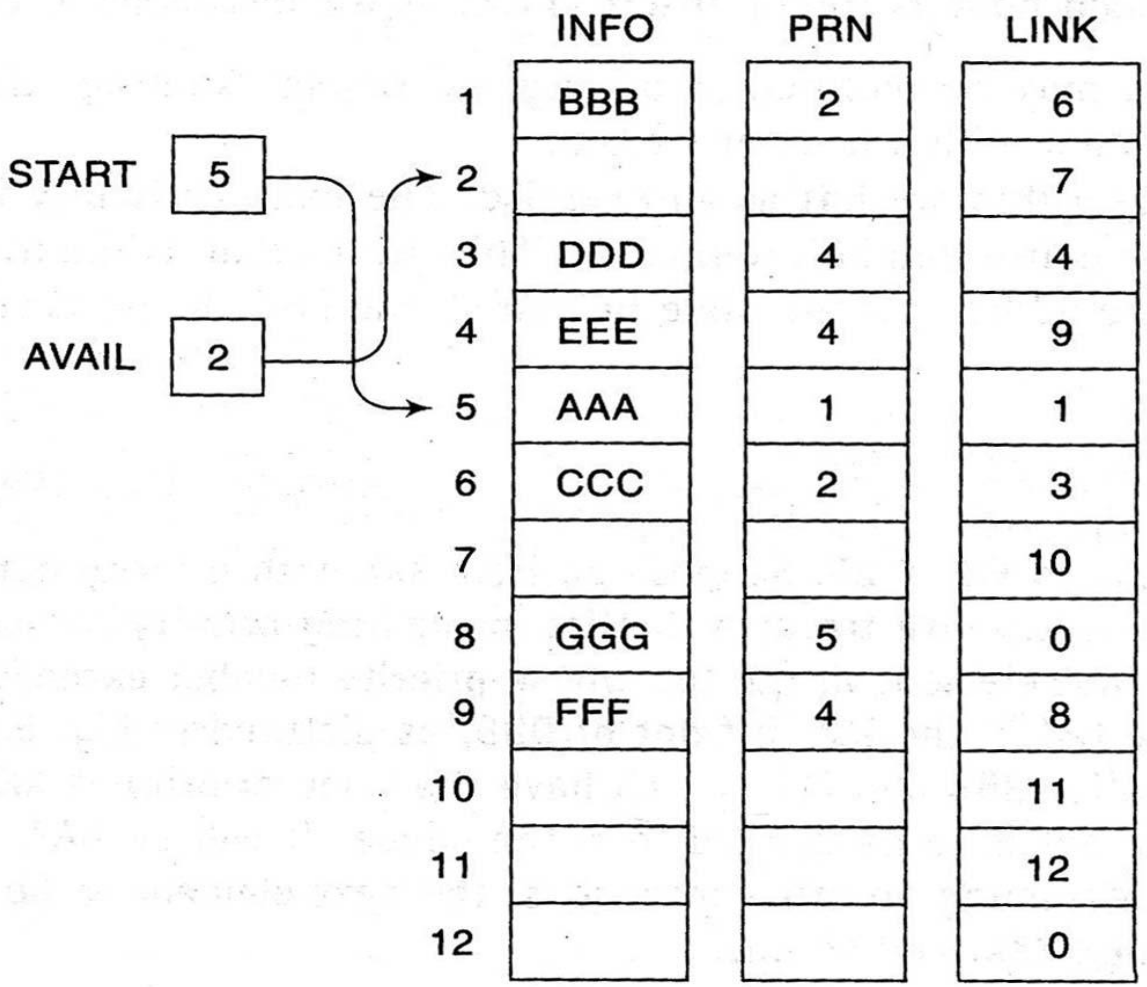
One-way list

- Each node will represented by three information
  - INFO-information field
  - PRN-priority number
  - LINK-link address
- A node X precedes a node Y in the list
  - When X has higher priority than Y or
  - When both have the same priority but X was added to the list before Y.

One-way list



One-way list


INFO
PRN
LINK
START
5
AVAIL
2
1 BBB 2 6
2 7
3 DDD 4 4
4 EEE 4 9
5 AAA 1 1
6 CCC 2 3
7 10
8 GGG 5 0
9 FFF 4 8
10 11
11 12
12 0

Insertion to one-way list queue

This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

a) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

b) If no such node is found, insert ITEM as the last element of the list.

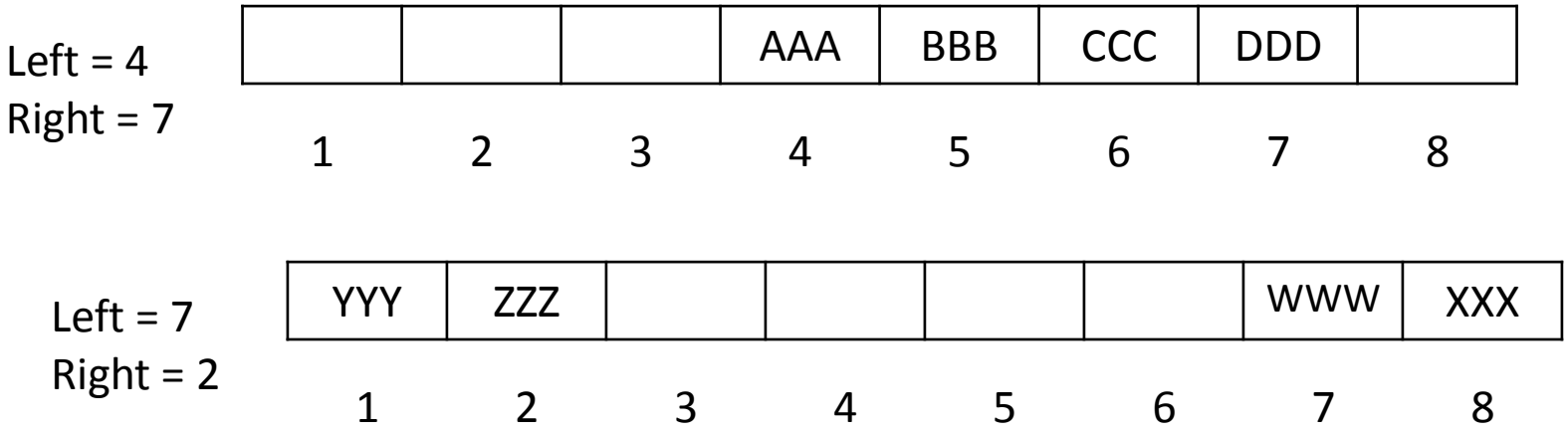
Deletion from one-way list queue

This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM=INFO[START]
2. Delete first node from the list.
3. Process ITEM
4. Exit

Deque (Double-ended queue)

- It is a double-ended queue.
- Items can be inserted and deleted from either ends.
- More versatile data structure than stack or queue.
- E.g. policy-based application (e.g. low priority go to the end, high go to the front)



Priority Queue

- More specialized data structure.
- Similar to Queue, having front and rear.
- Items are removed from the front.
- Items are ordered by key value so that the item with the lowest key (or highest) is always at the front.
- Two elements with same priority are processed according to order in which they were added to queue.
- Items are inserted in proper position to maintain the order.

- It can be maintained in memory by means of one-way list as:
  - Each node will contain 3 items → **info**, **priority_no** & **link**
  - A node X precedes a node Y in list when,
    - X has higher priority than Y
    - Both have same priority but X was added to list before Y

Polish (Prefix) Notation

Arithmetic Expressions

- Arithmetic Expressions involve *constants* and *operations*.

- Binary operations have different levels of precedence.
  - First : Exponentiation (^)
  - Second: Multiplication (*) and Division (/)
  - Third : Addition (+) and Subtraction (-)

Example

- Evaluate the following Arithmetic Expression:

  5 ^ 2 + 3 * 5 – 6 * 2 / 3 + 24 / 3 + 3

- First:

  25 + 3 * 5 – 6 * 2 / 3 + 24 / 3 + 3

- Second:

  25 + 15 – 4 + 8 + 3

- Third:

  47

Infix Notation

- Infix Notation: Operator symbol is placed between the two operands.

  Example:

  (5 * 3) + 2 & 5 * (3 + 2)

Polish (Prefix) Notation

Prefix / Polish Notation

- Polish Notation: The Operator Symbol is placed before its two operands.

  Example: + A B, * C D, / P Q etc.

- In prefix notation you put the operator first followed by the things it acts on and enclose the whole lot in brackets.

- So for example if you want to write 3+4, in scheme you say: (+ 3 4)

Examples

- (A + B) * C =

  * + ABC
- A + (B * C) =

  + A *BC
- (A + B) / (C - D) =

  / +AB –CD

- Also Known as Prefix Notation.

Advantage of Prefix notation

- The advantage of the prefix notation is that **no operator precedence is required**; the bracketing shows what the operator acts on.
- So in scheme if you want 3*4+2, you say : (+ (* 3 4) 2)
- And if you want 3*(4+2), you write: (* 3 (+ 4 2))

- **So in scheme, you don't need operator precedence, but you do need lots of brackets.**

Reverse-Polish (Postfix) Notation

Reverse-Polish / Postfix Notation

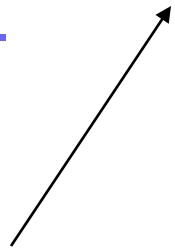
- Reverse Polish Notation: The Operator Symbol is placed after its two operands.

  Example: A B+, C D*, P Q/ etc.

- Also known as Postfix Notation.

Postfix and Prefix Examples

INFIX	POSTFIX	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D---	-A-B-C D
A - B - C - D	A B-C-D-	---A B C D



Prefix : Operators come before the operands

Advantage of Postfix notation

- You can easily evaluate a postfix expression in a single scan from left to right with the help of a stack (unlike evaluating infix expressions).

- There is no need of the concept of parentheses and precedence rules etc. in a postfix expression.

Transforming Infix into Postfix

Transforming Infix into Postfix

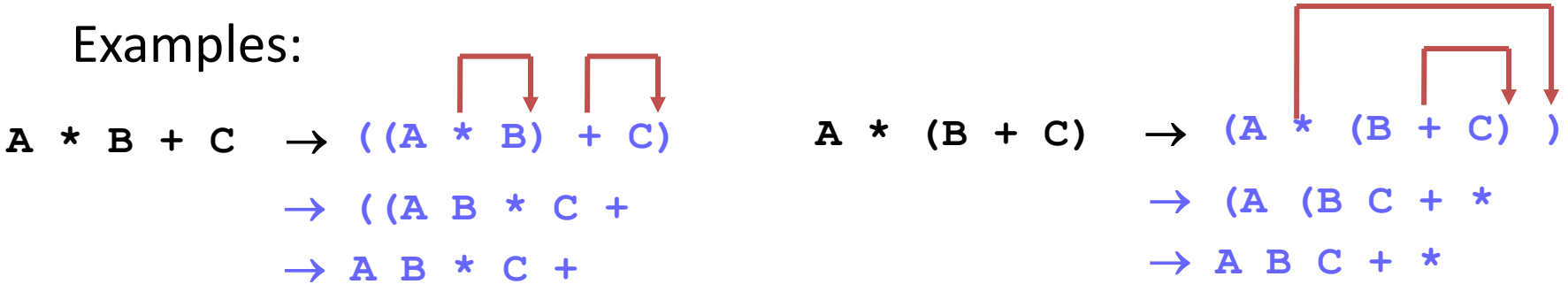
By hand: "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.

2. Replace each right parenthesis by the corresponding operator.

3. Erase all left parentheses.

Examples:



Infix to Postfix

A + (B * C – (D / E ↑ F) * G) * H

Symbol Scanned	Stack	Expression P
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (-	A B C *
(	(+ (- (	A B C *
D	(+ (- (	A B C * D
/	(+ (- (/	A B C * D
E	(+ (- (/	A B C * D E
↑	(+ (- (/ ↑	A B C * D E

A + (B * C – (D / E ↑ F) * G) * H

Symbol Scanned	Stack	Expression P
F	(+ (- (/ ↑	A B C * D E F
)	(+ (-	A B C * D E F ↑/
*	(+ (- *	A B C * D E F ↑/
G	(+ (- *	A B C * D E F ↑/ G
)	(+	A B C * D E F ↑/ G * -
*	(+ *	A B C * D E F ↑/ G * -
H	(+ *	A B C * D E F ↑/ G * - H
)		A B C * D E F ↑/ G * - H * +

Infix to Postfix Transformation (Algo.)

POLISH (Q, P)

1. PUSH "(" on to 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. **[Don't add the left parenthesis to P.]**

   [End of If Structure.]

   [End of step 2 Loop.]
7. Exit.

Evaluating Postfix (RPN) Expressions

Evaluating Postfix (RPN) Expressions

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands
   and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

```
    2 3 4 + 5 6 - - *
→   2 3 4 + 5 6 - - *
→   2 7 5 6 - - *
→   2 7 5 6 - - *
→   2 7 -1 - *
→   2 7 -1 - *   →  2 8 *  →  2 8 *  → 16
```

Evaluation of Postfix (RPN) Expression (Algo.)

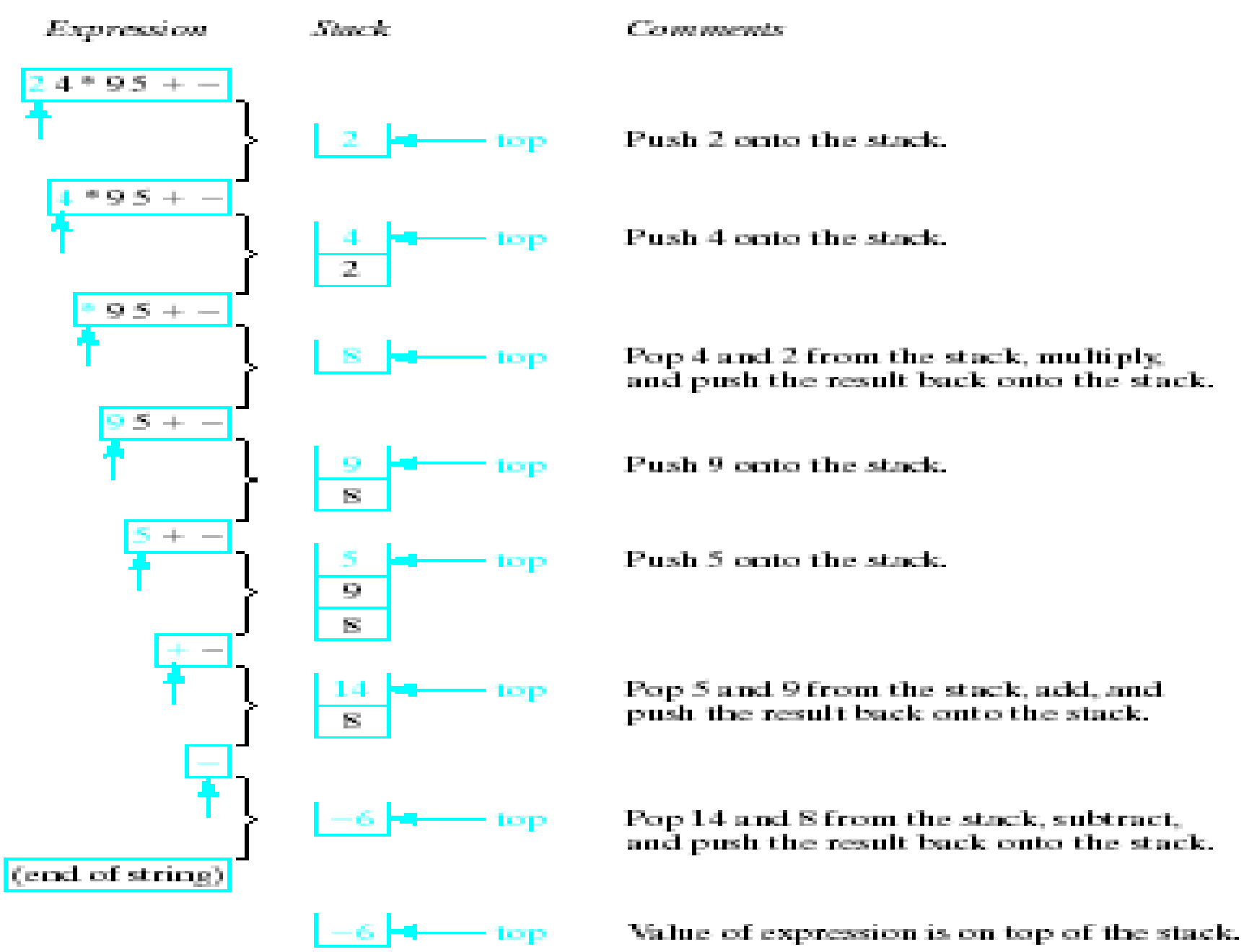
- P is an arithmetic expression in Postfix Notation.

1. Add a right parenthesis “)” at the end of P.

2. Scan P from left to right and Repeat Step 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 @ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next to top element.
   (b) Evaluate B @ 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.


Expression
Stack
Comments
2 4 * 9 5 + −
4 * 9 5 + −
* 9 5 + −
9 5 + −
5 + −
+ −
−
(end of string)
2
top
Push 2 onto the stack.
4
2
top
Push 4 onto the stack.
8
top
Pop 4 and 2 from the stack, multiply, and push the result back onto the stack.
9
8
top
Push 9 onto the stack.
5
9
8
top
Push 5 onto the stack.
14
8
top
Pop 5 and 9 from the stack, add, and push the result back onto the stack.
−6
top
Pop 14 and 8 from the stack, subtract, and push the result back onto the stack.
−6
top
Value of expression is on top of the stack.

Quick Sort

Quick Sort

- It is an algorithm of divide and conquer.

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

- After reduction step, final position of no. is:

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

First sublist Second sublist

Iterative procedure

- The algorithm begins by pushing boundary values 1 and 12 onto stack to yield

Lower: 1 Upper: 12

- To apply reduction step, top values are removed from stack, leaving

Lower: (empty) Upper: (empty)

- And then reduction step is applied to list from A[1] to A[12]. Finally 44 reaches its final location A[5].
- Accordingly, algo. Pushes boundary values 1 and 4 to 1st sublist and 6 and 12 to 2nd sublist onto stack, leaving

Lower: 1, 6 Upper: 4, 12

- Now, apply reduction step again by removing top values from stack, leaving

Lower: 1 Upper: 4

- Then reduction step is applied to sublist A[6] to A[12], and so on.

Lower: 1, 6 Upper: 4, 10

Quick sort algorithm

QUICK(A, N, BEG, END, LOC)

1. **[Initialize]** Set LEFT = BEG, RIGHT=END, LOC=BEG
2. **[Scan from right to left]**
   a. Repeat while A[LOC] <=A[RIGHT] and LOC != RIGHT
      i. RIGHT = RIGHT-1
   b. If LOC = RIGHT then return
   c. If A[LOC] > A[RIGHT] then
      i. [Interchange A[LOC] and A[RIGHT]]
      Temp = A[LOC], A[LOC] = A[RIGHT], A[RIGHT] = Temp
      ii. Set LOC = RIGHT
      iii. Go to step 3.
3. **[Scan from left to right]**
   a. Repeat while A[LEFT] <=A[LOC] and LEFT != LOC
      i. LEFT = LEFT+1
   b. If LOC = LEFT then return
   c. If A[LEFT] > A[LOC] then
      i. [Interchange A[LEFT] and A[LOC]]
      Temp = A[LOC], A[LOC] = A[LEFT], A[LEFT] = Temp
      ii. Set LOC = LEFT
      iii. Go to step 2.

Quick sort algorithm

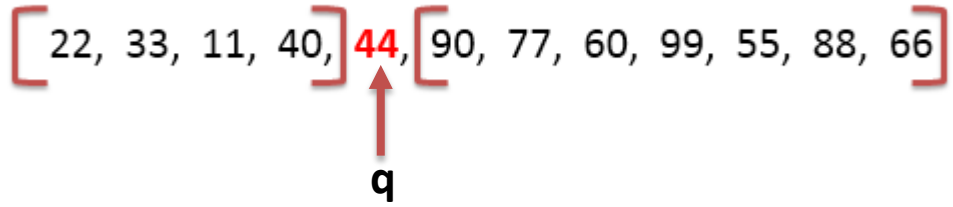
QUICKSORT()

1. **[Initialize]** TOP = 0
2. **[Push boundary value of A onto stacks when A has 2 or more elements]**
   If N >1 then TOP = TOP + 1, LOWER[1] = 1, UPPER[1] = N
3. Repeat steps 4 to 7 while TOP != 0
4. **[Pop sublist from stacks]**
   Set BEG = LOWER[TOP] , END = UPPER[TOP]
   TOP = TOP – 1
5. Call QUICK(A, N, BEG, END, LOC)
6. **[Push left sublist onto stacks when it has 2 or more elements]**
   If BEG < LOC-1 then
   TOP = TOP + 1, LOWER[TOP] = BEG, UPPER[TOP] = LOC-1
7. **[Push right sublist onto stacks when it has 2 or more elements]**
   If LOC+1 < END then
   TOP = TOP+1, LOWER[TOP] = LOC + 1, UPPER[TOP] = END
8. Exit.

Recursive procedure

Recursive Algorithm

QUICKSORT(A, p, r)
p is left and r is right



1. **if** p < r
2. q = PARTITION(A, p, r)
3. QUICKSORT(A, p, q-1)
4. QUICKSORT(A, q +1, r)

[End of if]

5. Exit

http://me.dt.in.th/page/Quicksort/

Recursive Algorithm(partitioning)

PARTITION(A, p, r)

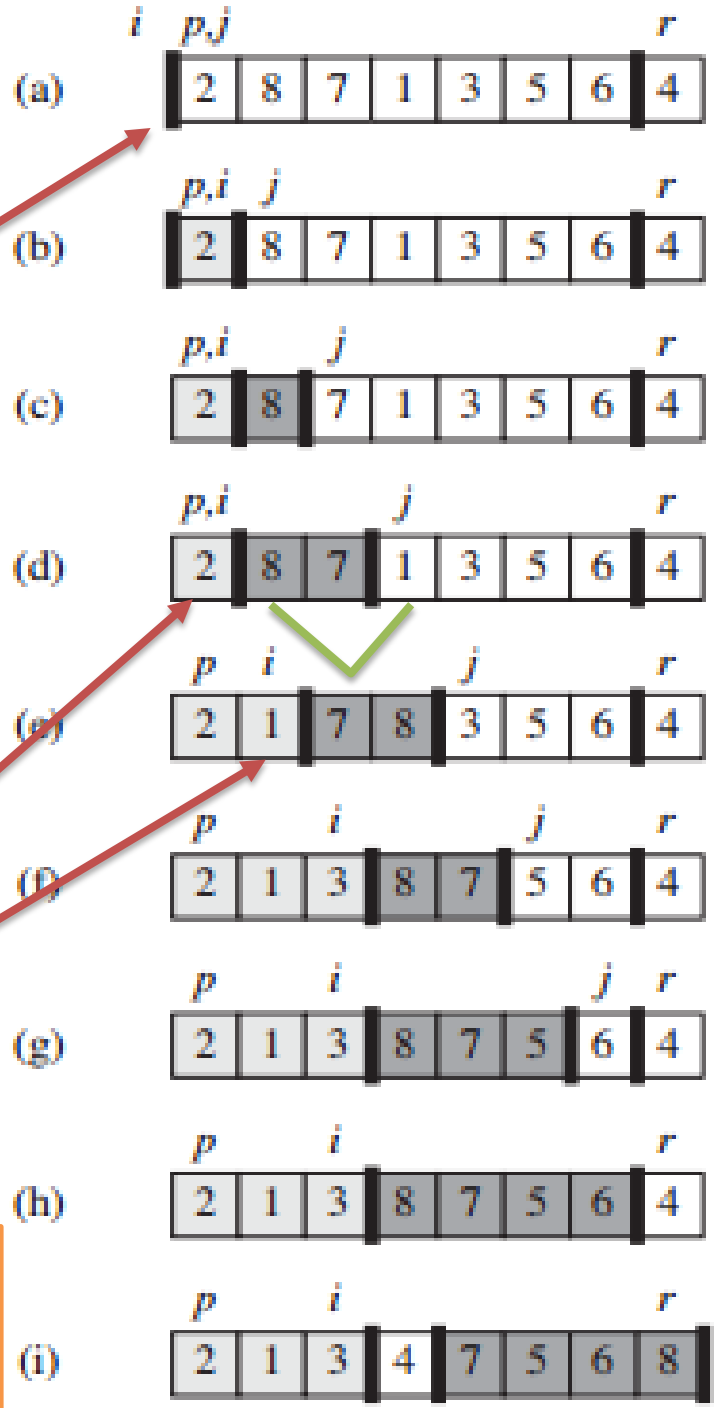
1. x = A[r]
2. i = p-1
3. **for** j = p **to** r – 1
4. **if** A[j] <= x
5. i = i+1
6. exchange A[i] with A[j]

[End of if statement]

[End of for loop]

7. exchange A[i+1] with A[r]
8. **return** i

Partitioning steps by assuming rightmost element as pivot



Purpose of “i” is to interchange first scanned value in window with the value at “j” if value at “j” < pivot value

Complexity of Quick Sort

- Worst case
  - $O(n^2)$
- Average case
  - $O(n \log n)$

Merge sort

Divide and Conquer

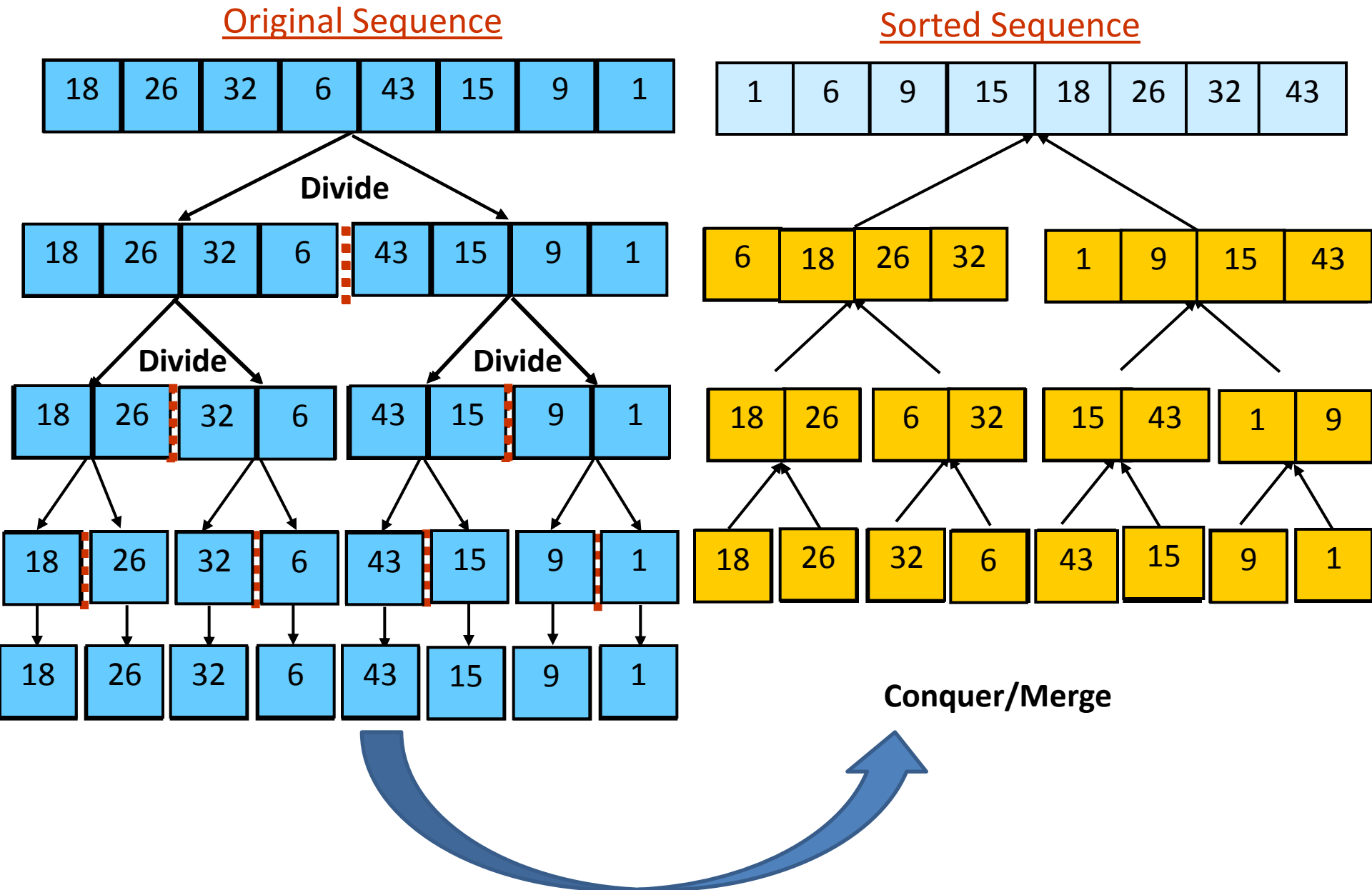
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

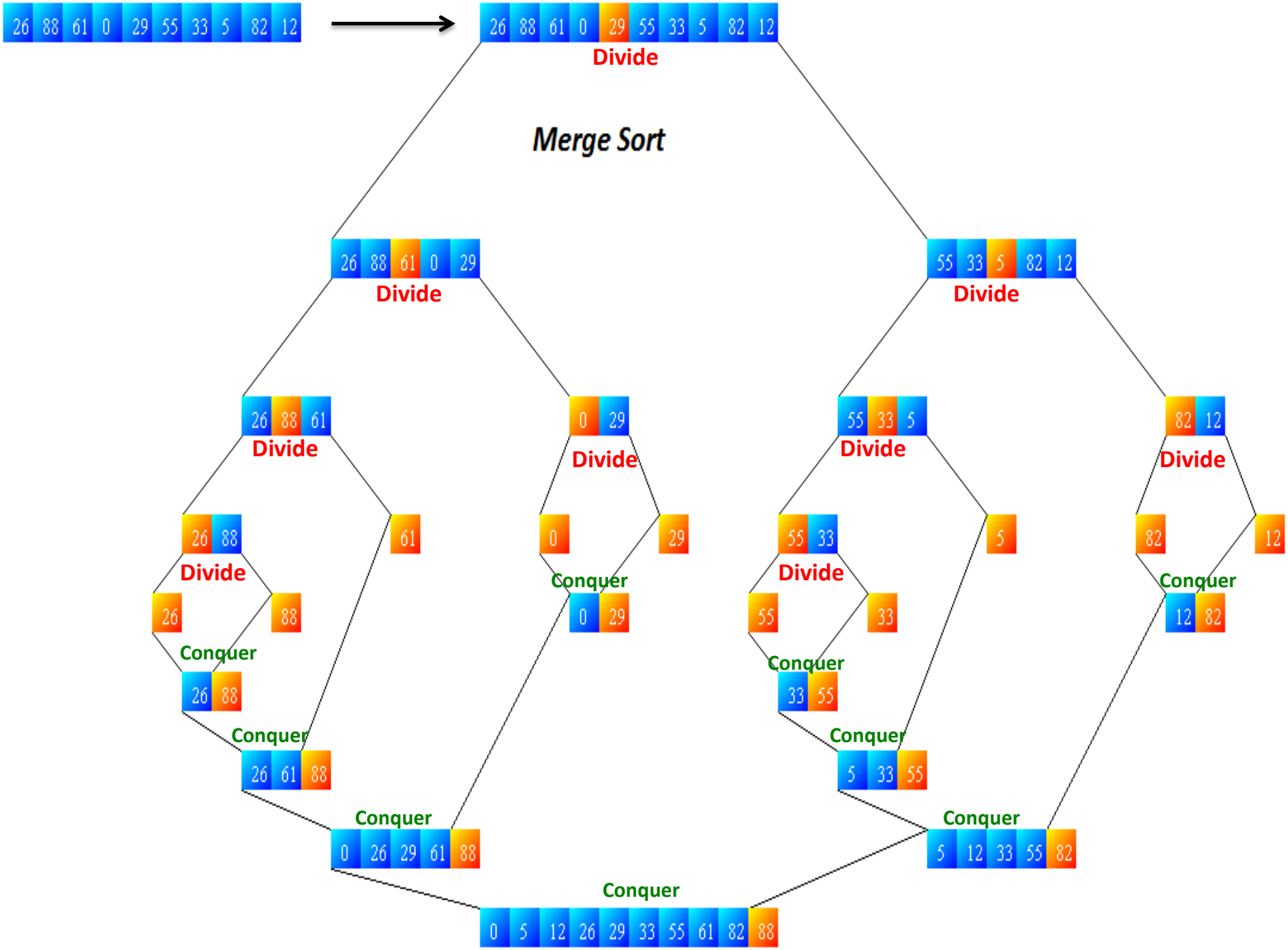
An Example: Merge Sort

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine*:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example




26 88 61 0 29 55 33 5 82 12
26 88 61 0 29 55 33 5 82 12
Divide
Merge Sort
26 88 61 0 29
Divide
55 33 5 82 12
Divide
26 88 61
Divide
0 29
Divide
55 33 5
Divide
82 12
Divide
26 88
Divide
61
0
29
55 33
Divide
5
82
12
Conquer
26
88
0 29
55
33
Conquer
Conquer
26 88
33 55
Conquer
Conquer
26 61 88
5 33 55
Conquer
Conquer
0 26 29 61 88
5 12 33 55 82
Conquer
0 5 12 26 29 33 55 61 82 88

Merge-Sort (A, fst, lst)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, fst, lst)   // sort A[fst...lst] by divide & conquer
1     if fst< lst
2        then mid ← ⌊(fst+lst)/2⌋
3             MergeSort (A, fst, mid)
4             MergeSort (A, mid+1, lst)
5             Merge (A, fst, mid, lst)        // merges  A[fst..mid]  with  A[mid+1..lst]
```

Initial Call: MergeSort(*A*, 1, *n*)

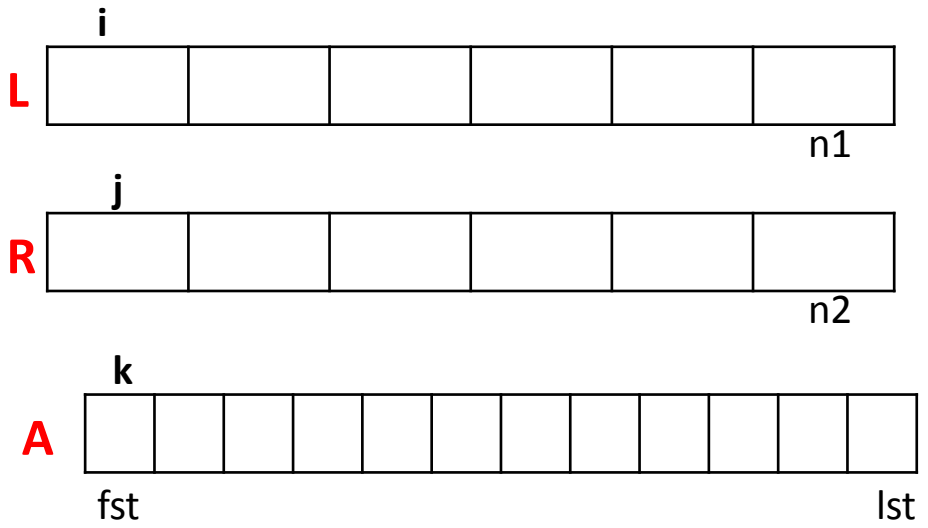
Procedure Merge

```
Merge(A, fst, mid, lst)
1  n1 ← (mid – fst + 1)
2  n2 ← (lst – mid)

3       for i ← 1 to n1
4           do L[i] ← A[fst+ i – 1]
5       for j ← 1 to n2
6           do R[j] ← A[mid + j]

7       L[n1+1] ← ∞
8       R[n2+1] ← ∞
9       i ← 1
10      j ← 1
11      for k ←fst to lst
12          do if L[i] ≤ R[j]
13             then A[k] ← L[i]
14                  i ← i + 1
15             else A[k] ← R[j]
16                  j ← j + 1
```

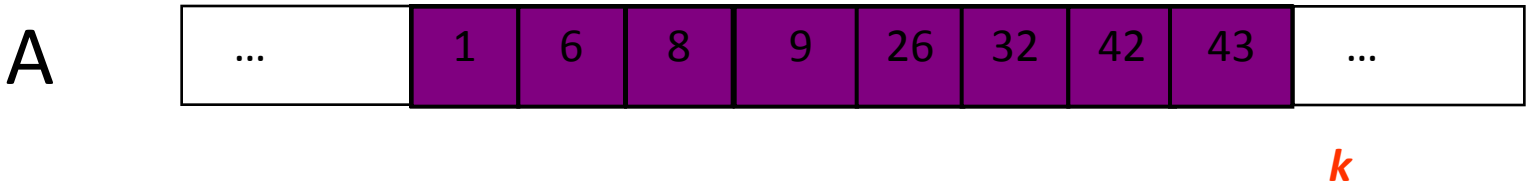
Input → Array containing sorted subarrays ***A[fst...mid]*** and ***A[mid+1…lst]***

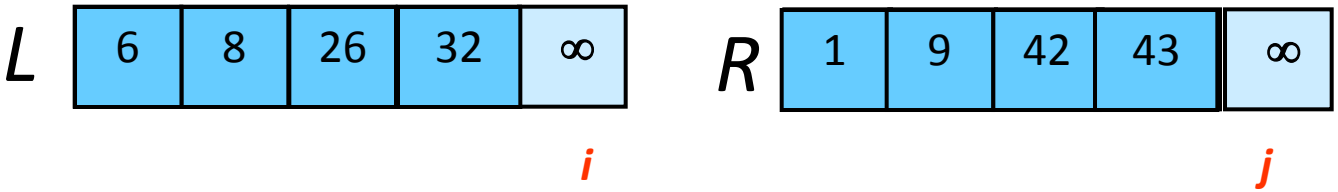
Output → Merged sorted subarray in ***A[fst...lst]***

Sentinels, to avoid having to check if either subarray is fully copied at each step.



Merge – Example





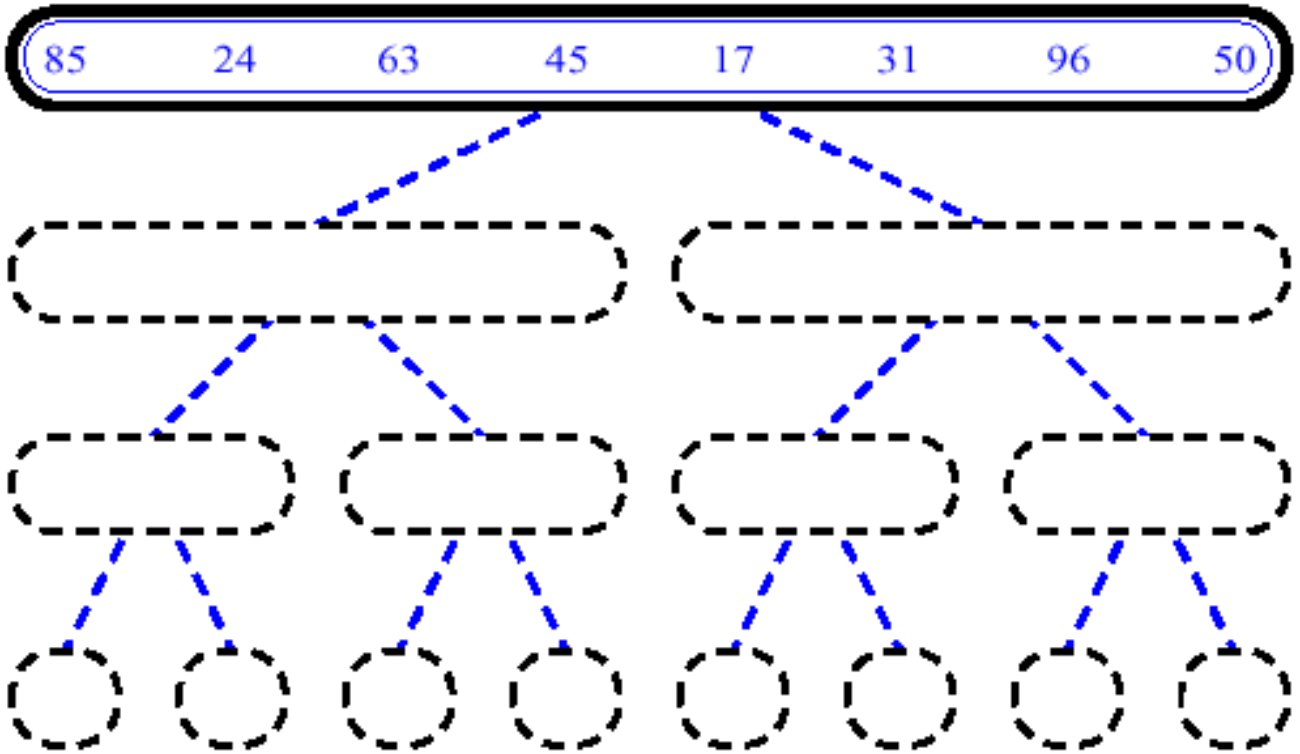
Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

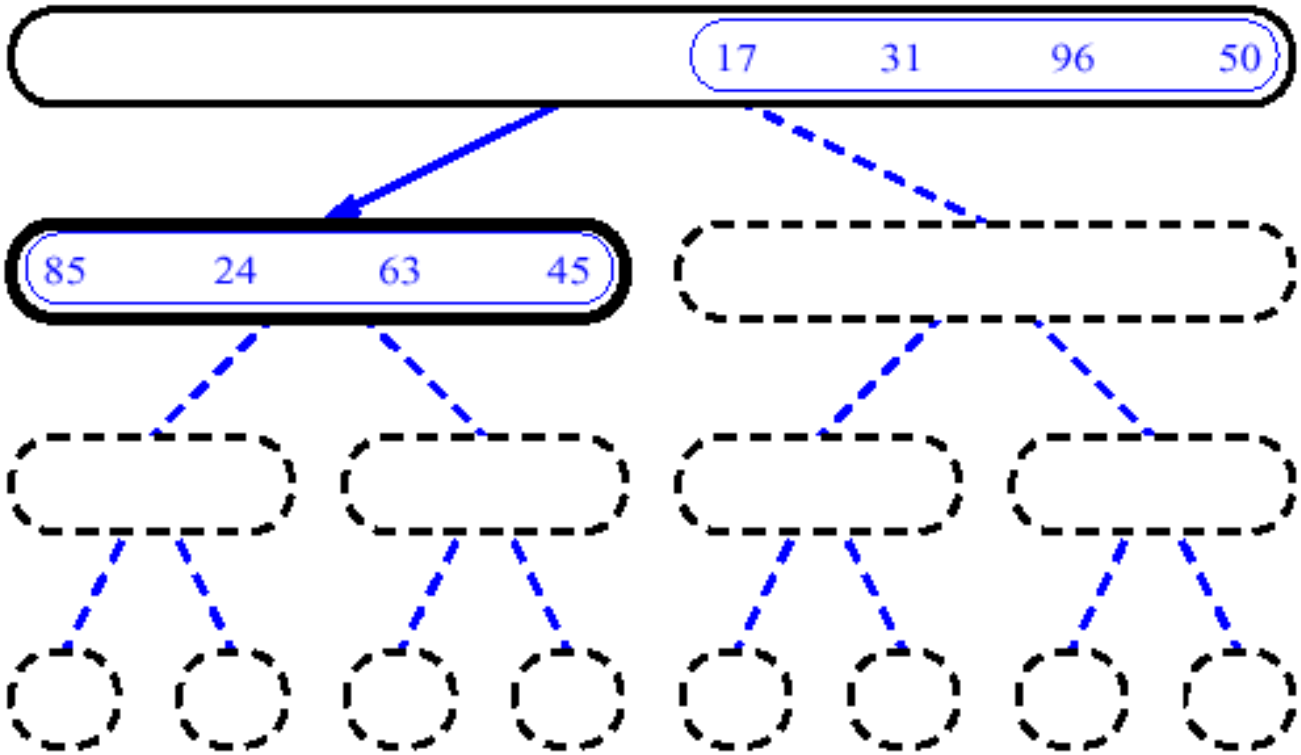
$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS, Chapter 4)

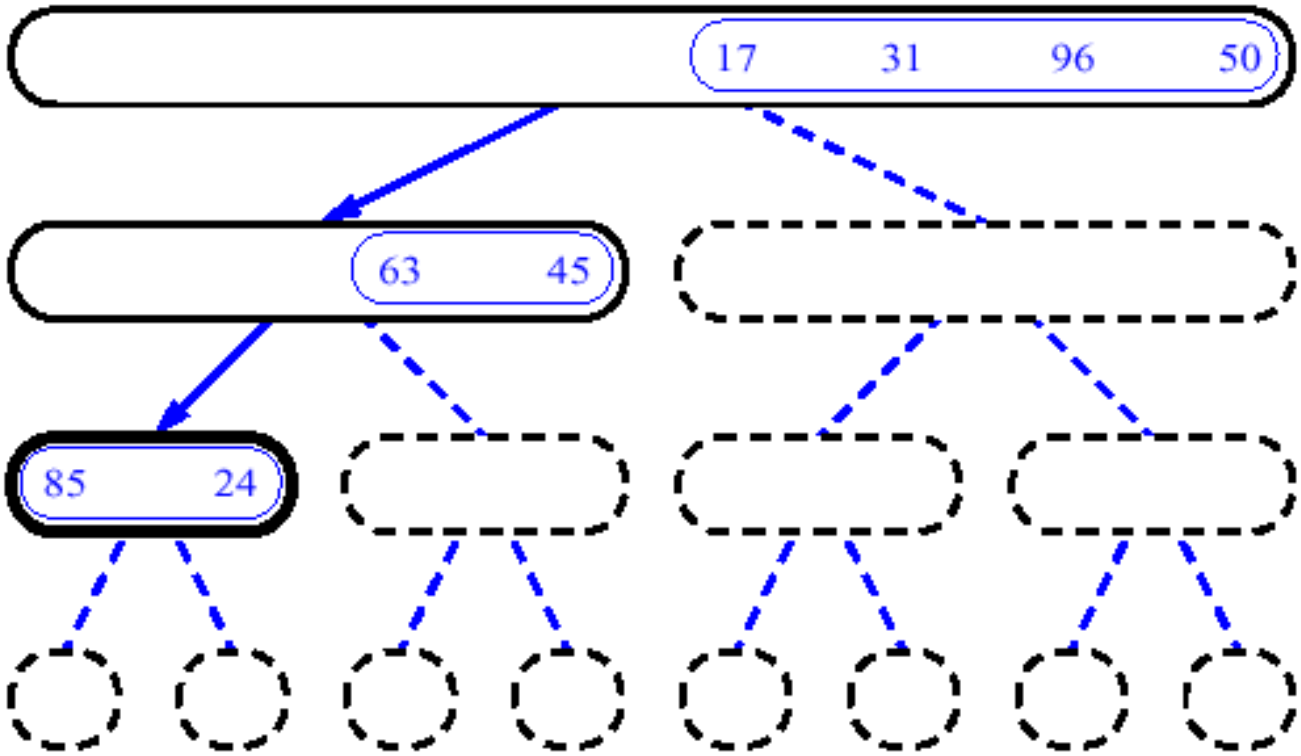
MergeSort (Example) - 1



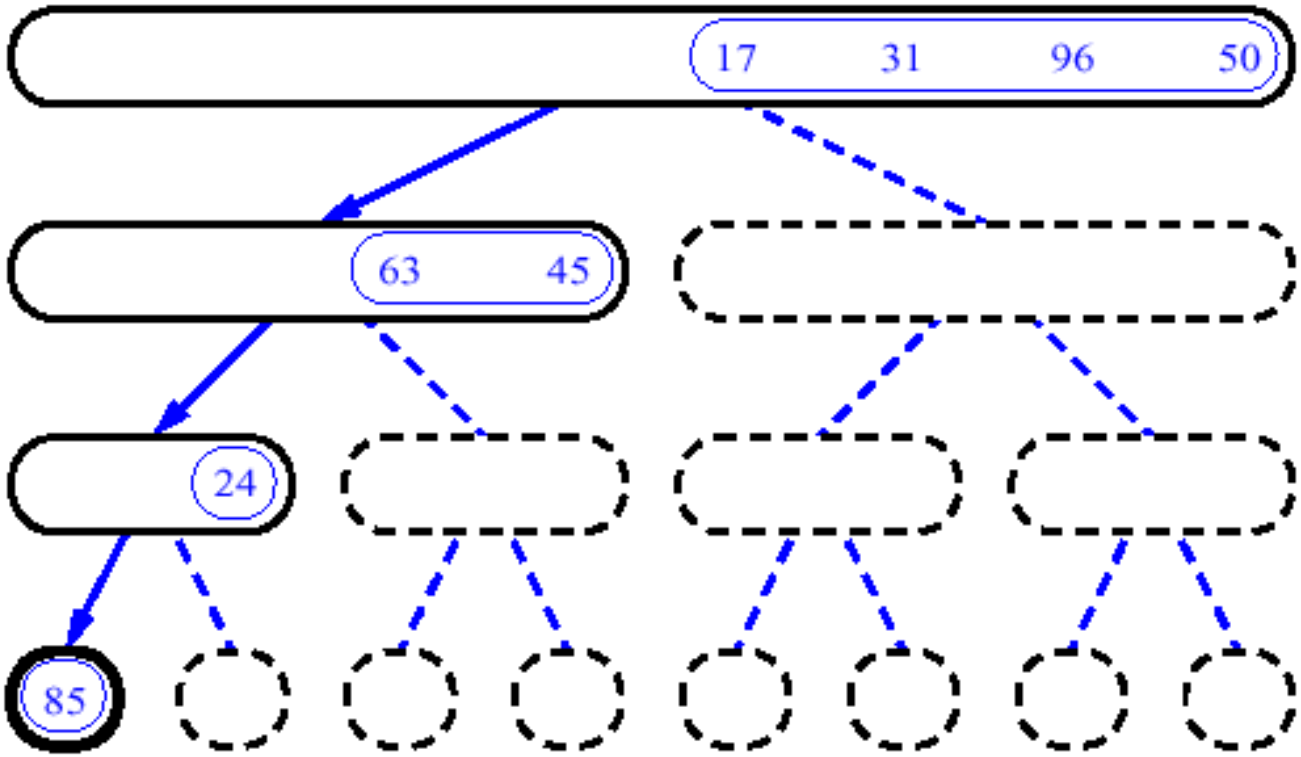
MergeSort (Example) - 2



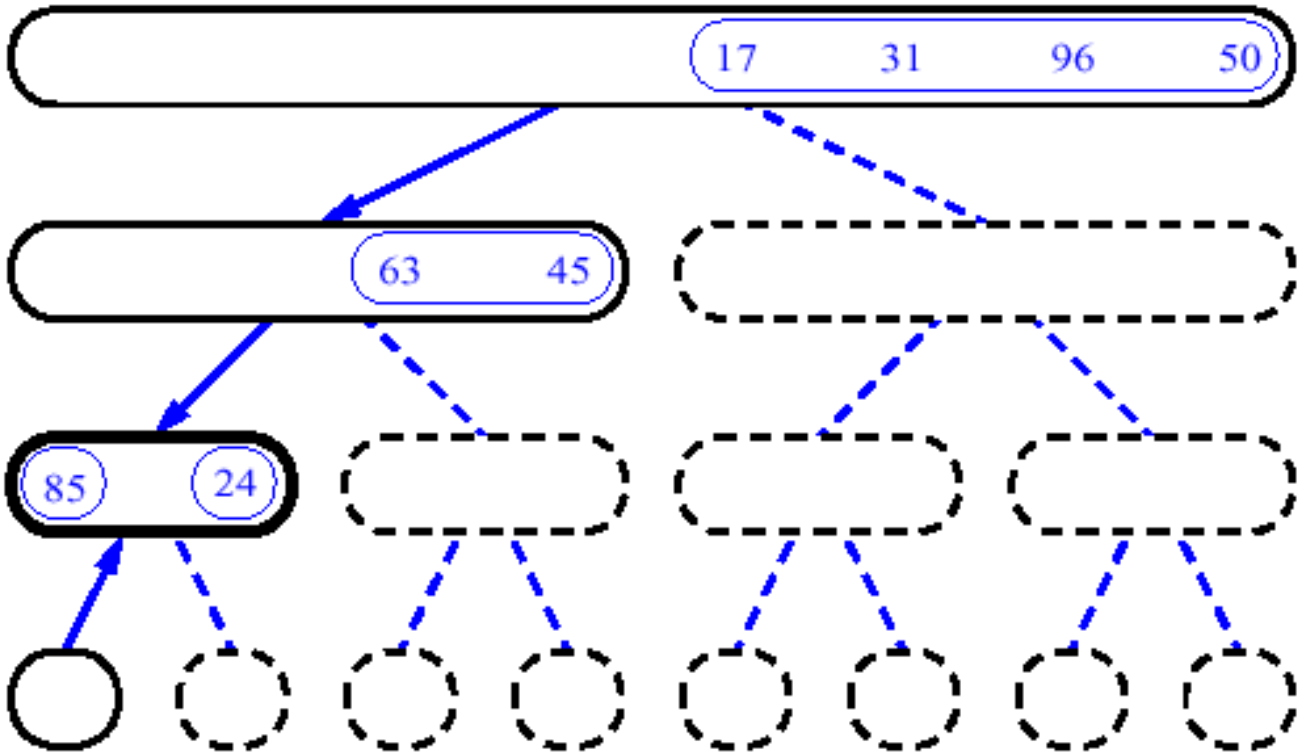
MergeSort (Example) - 3



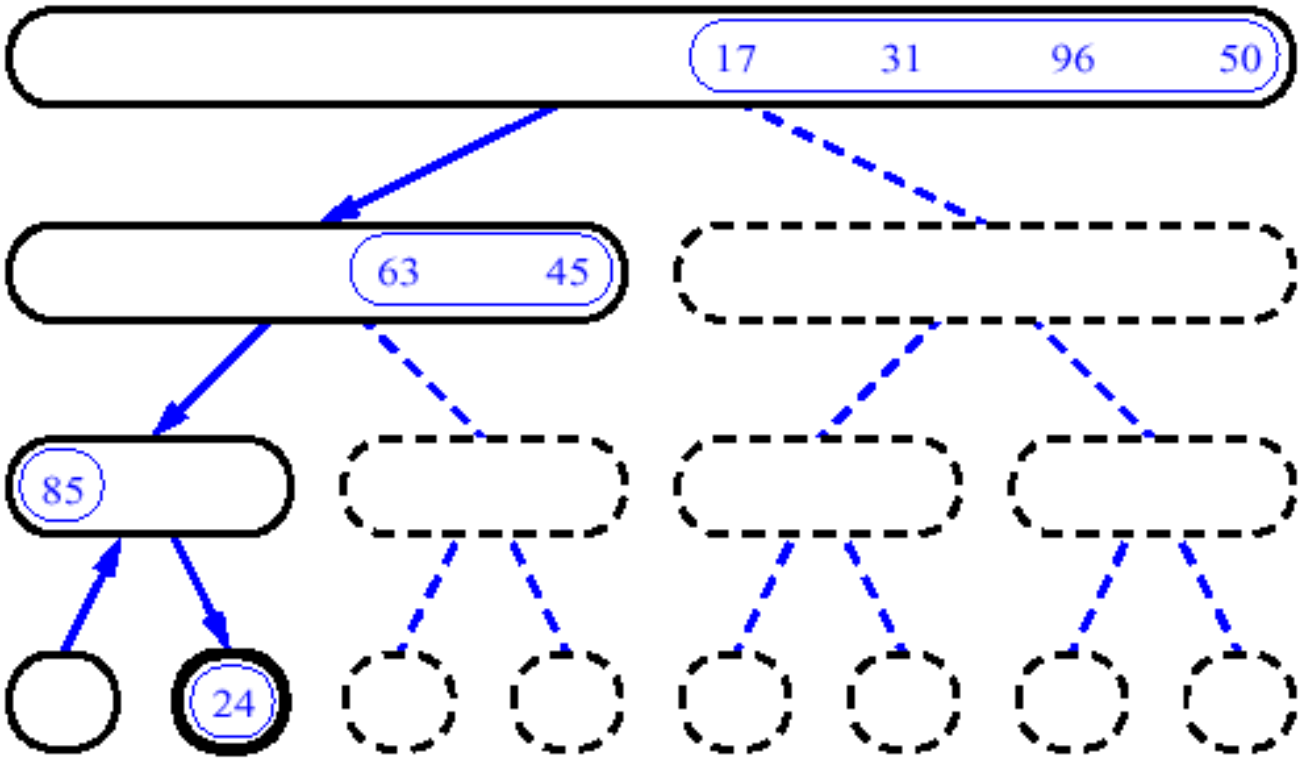
MergeSort (Example) - 4



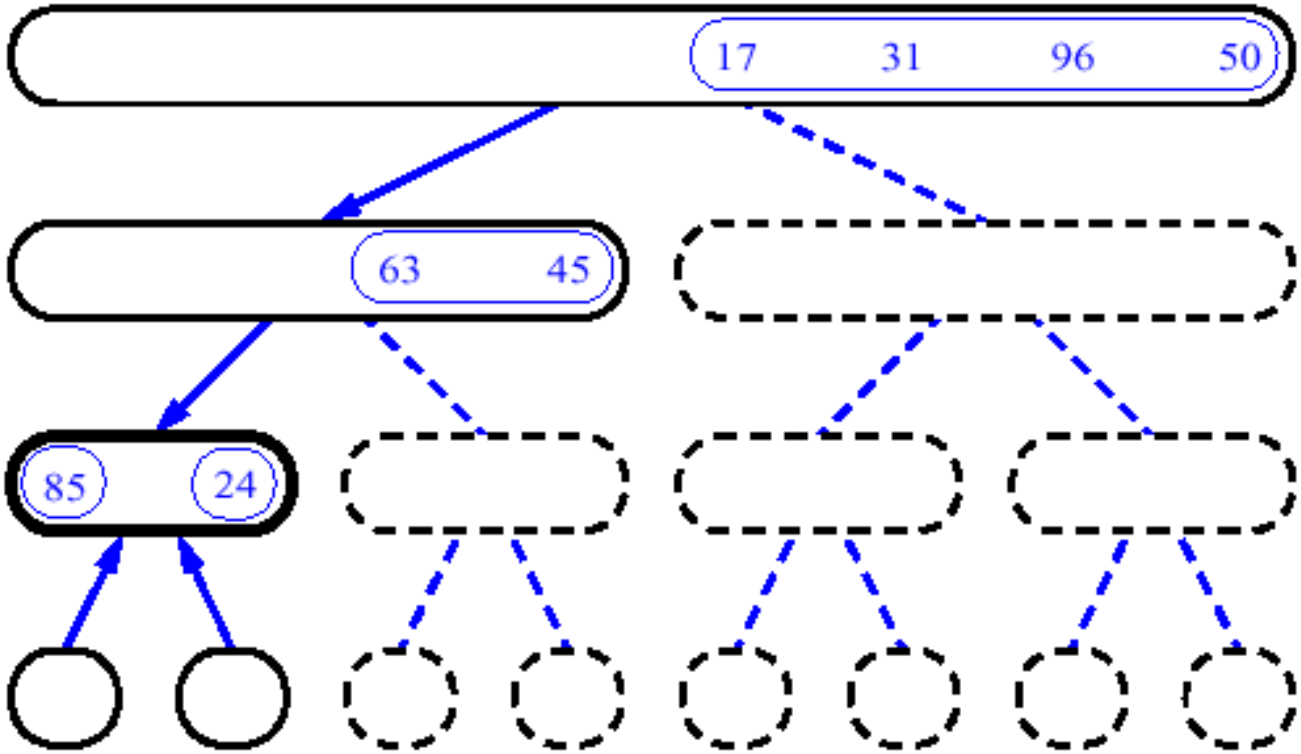
MergeSort (Example) - 5



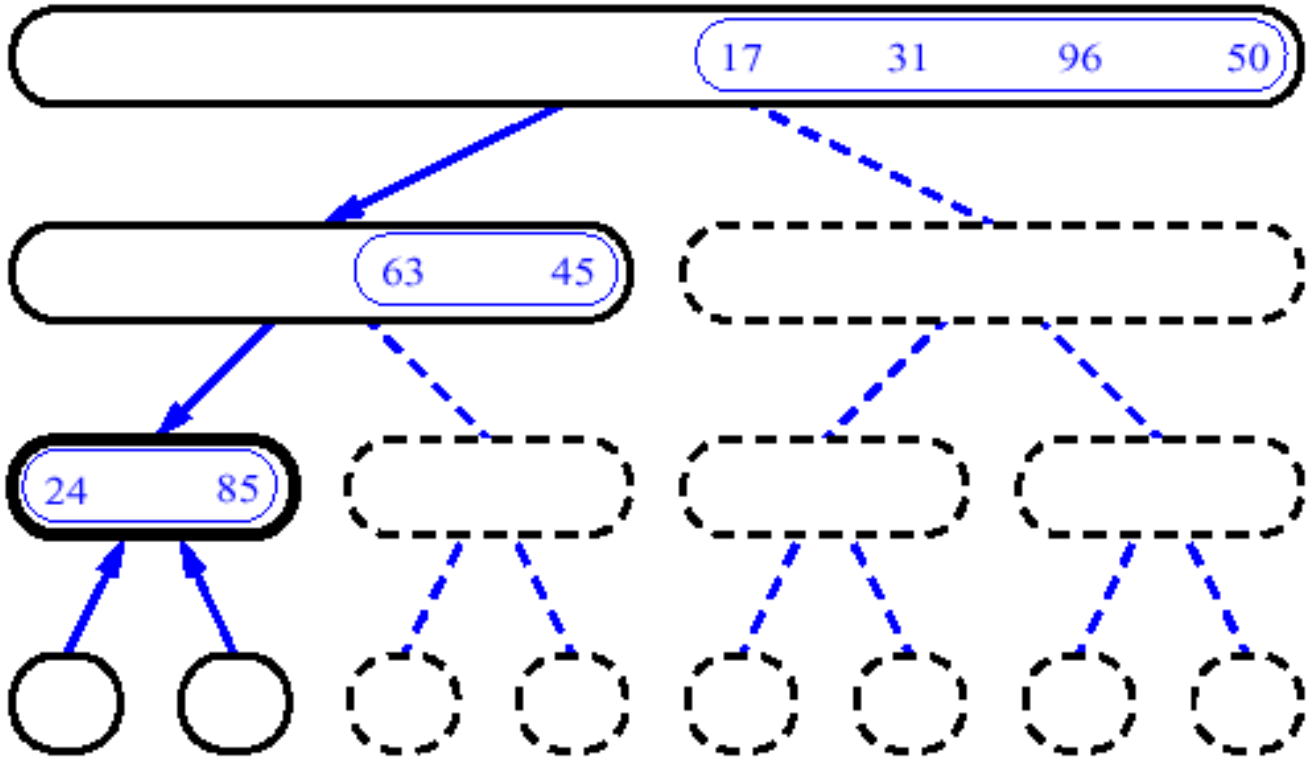
MergeSort (Example) - 6



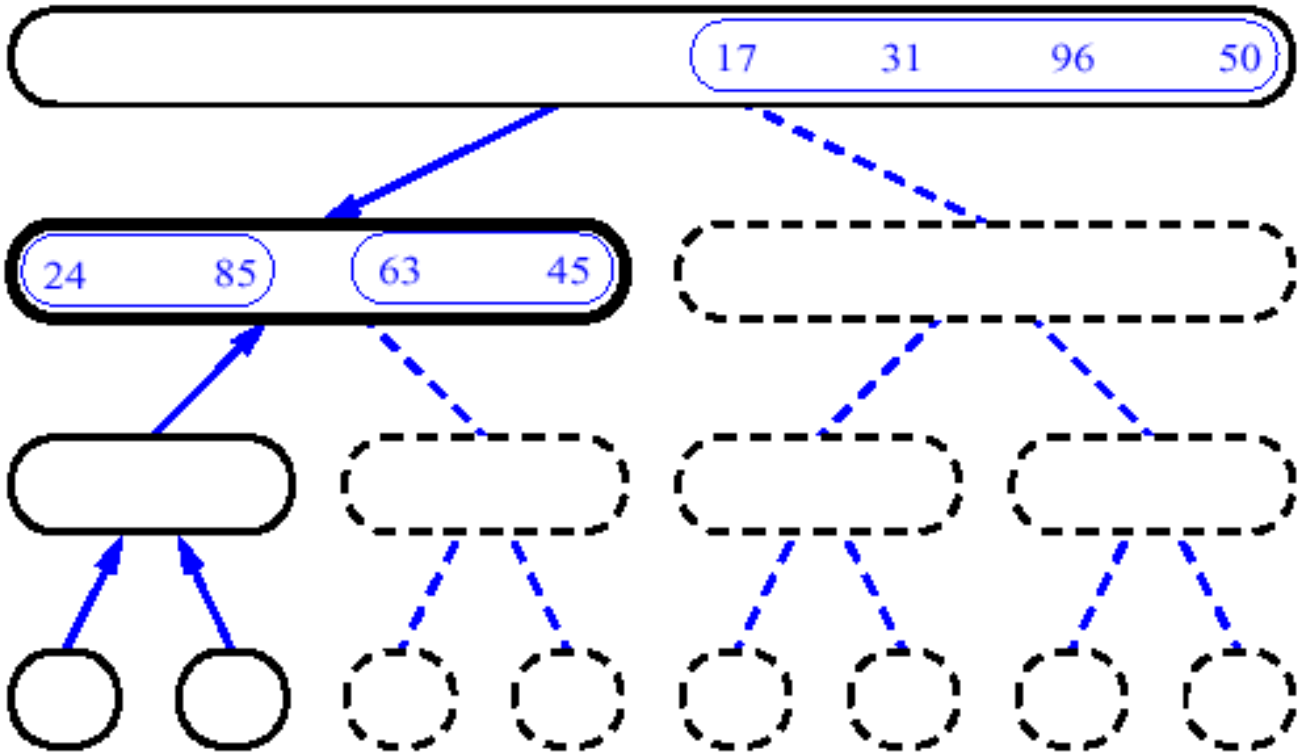
MergeSort (Example) - 7



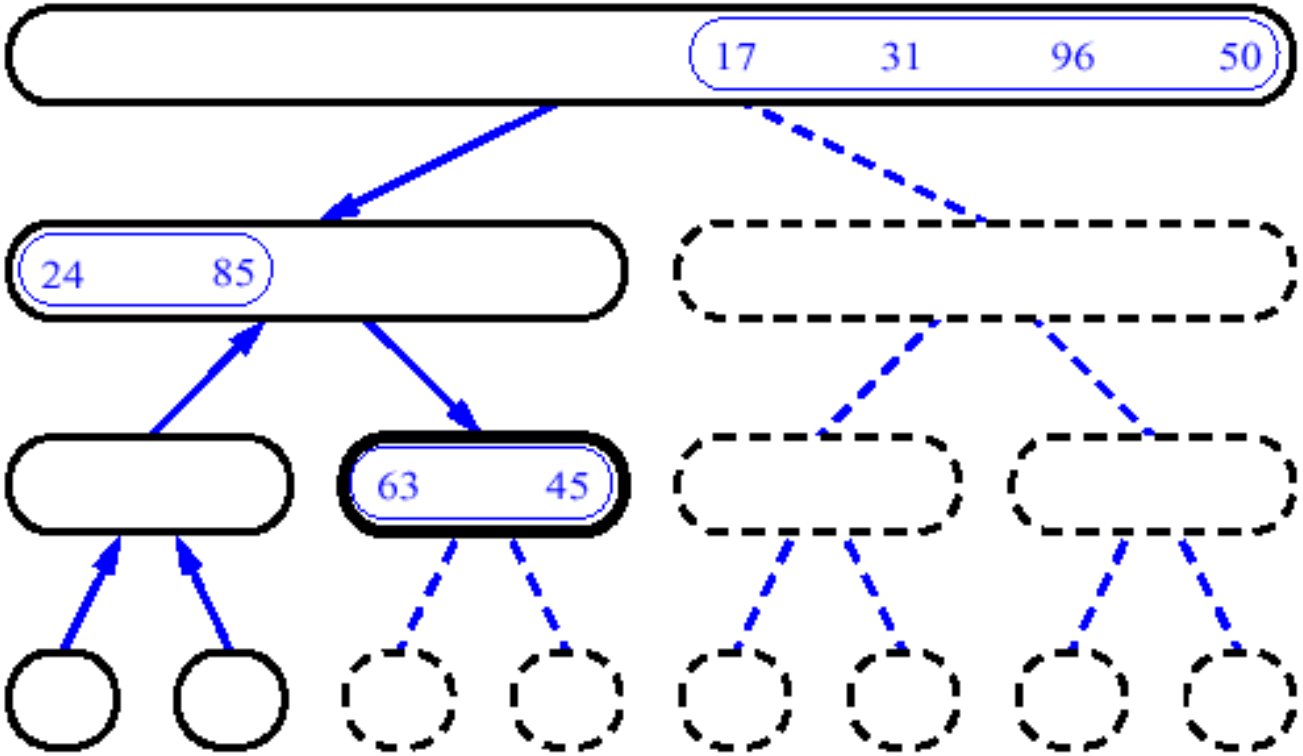
MergeSort (Example) - 8



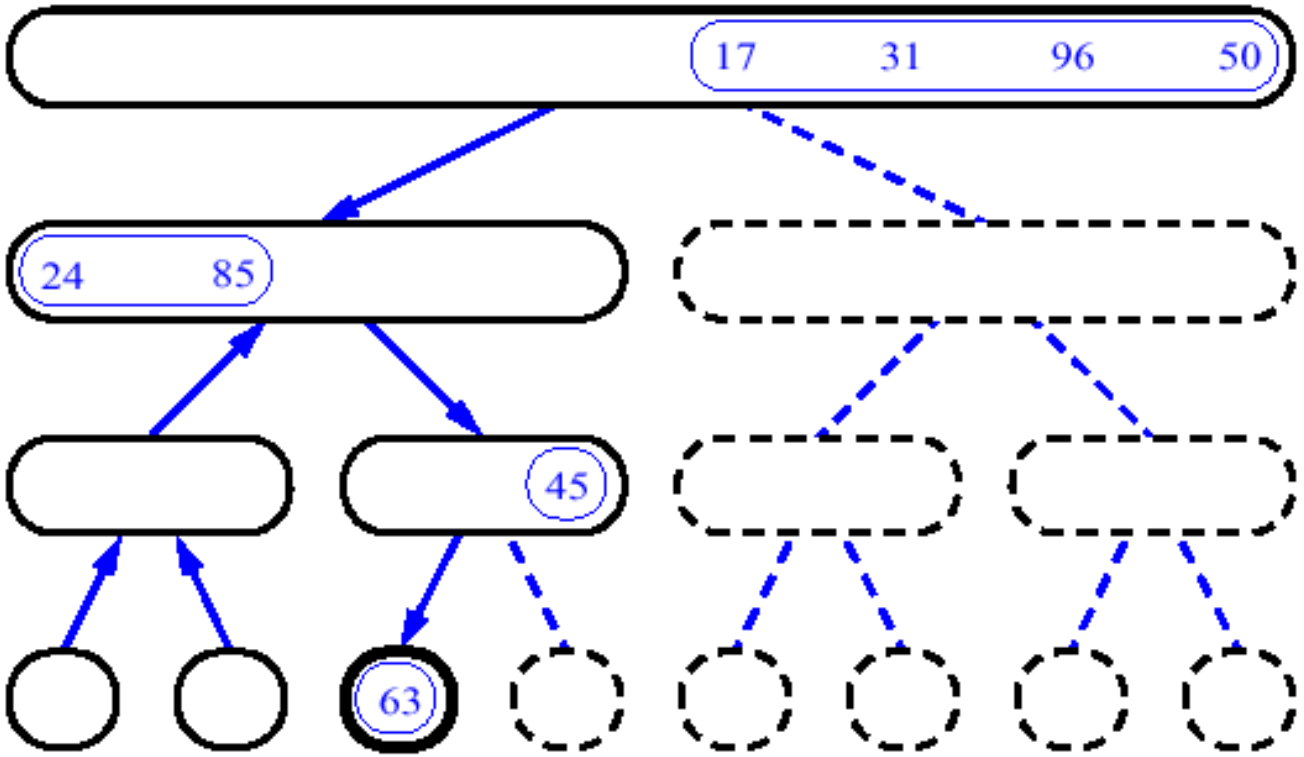
MergeSort (Example) - 9



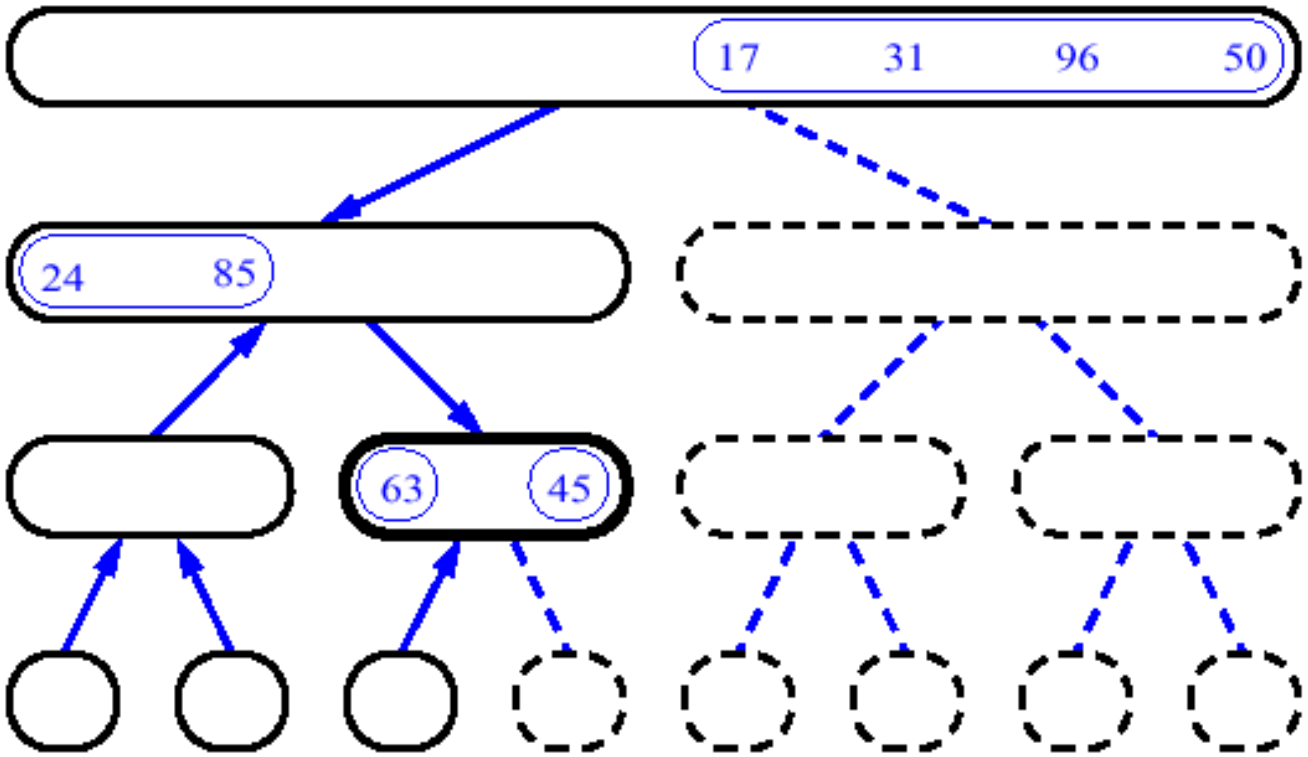
MergeSort (Example) - 10



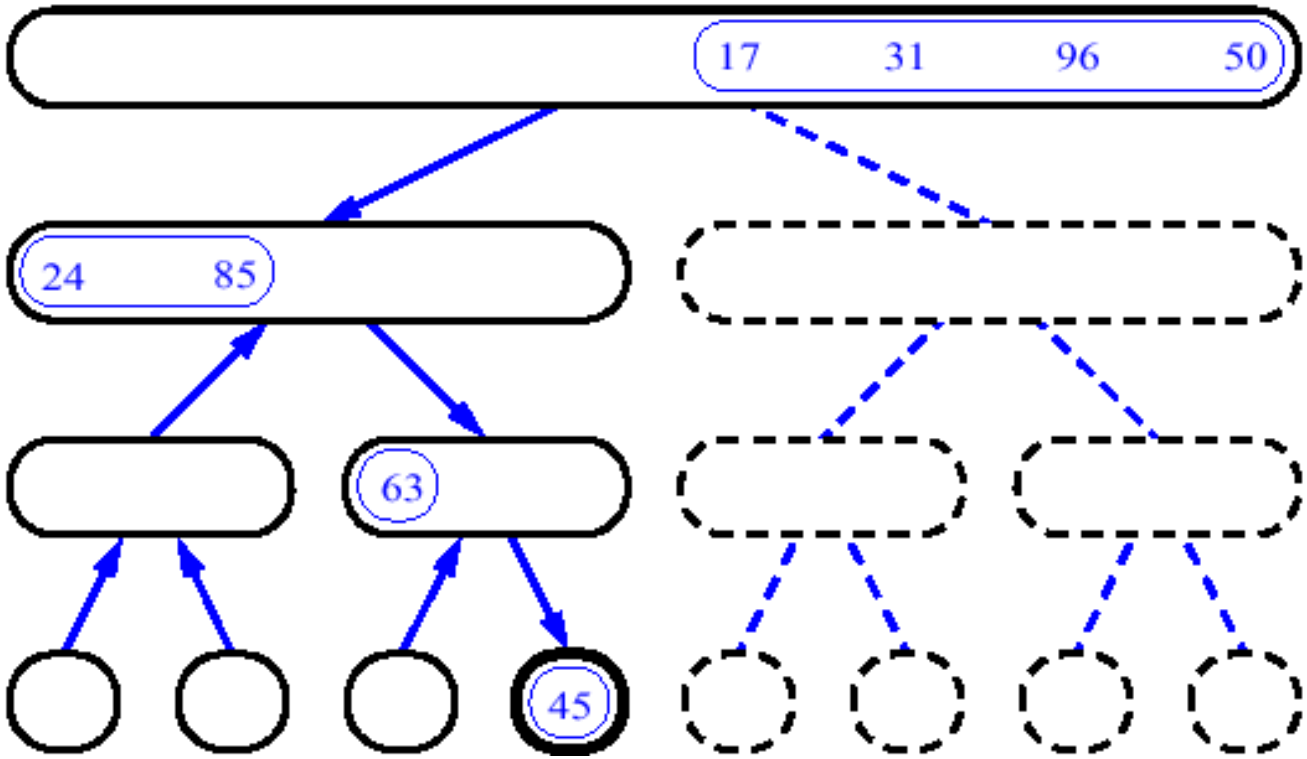
MergeSort (Example) - 11



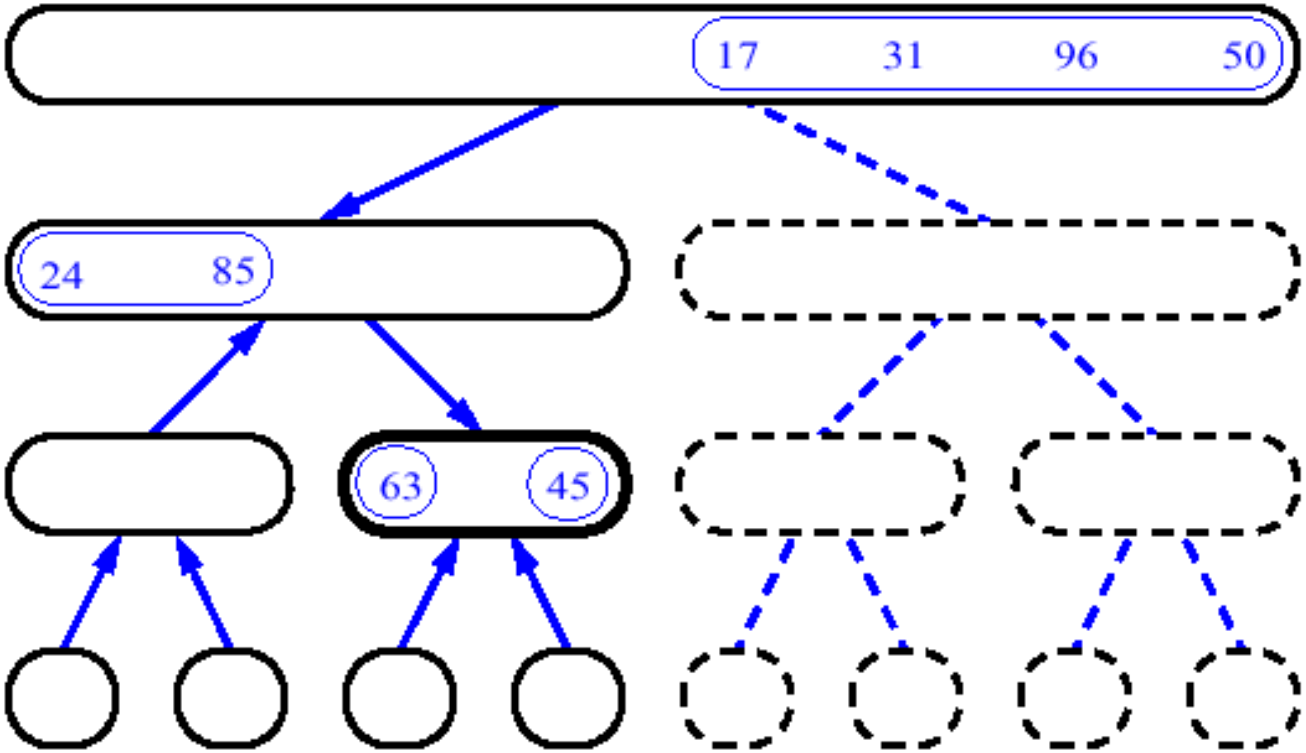
MergeSort (Example) - 12



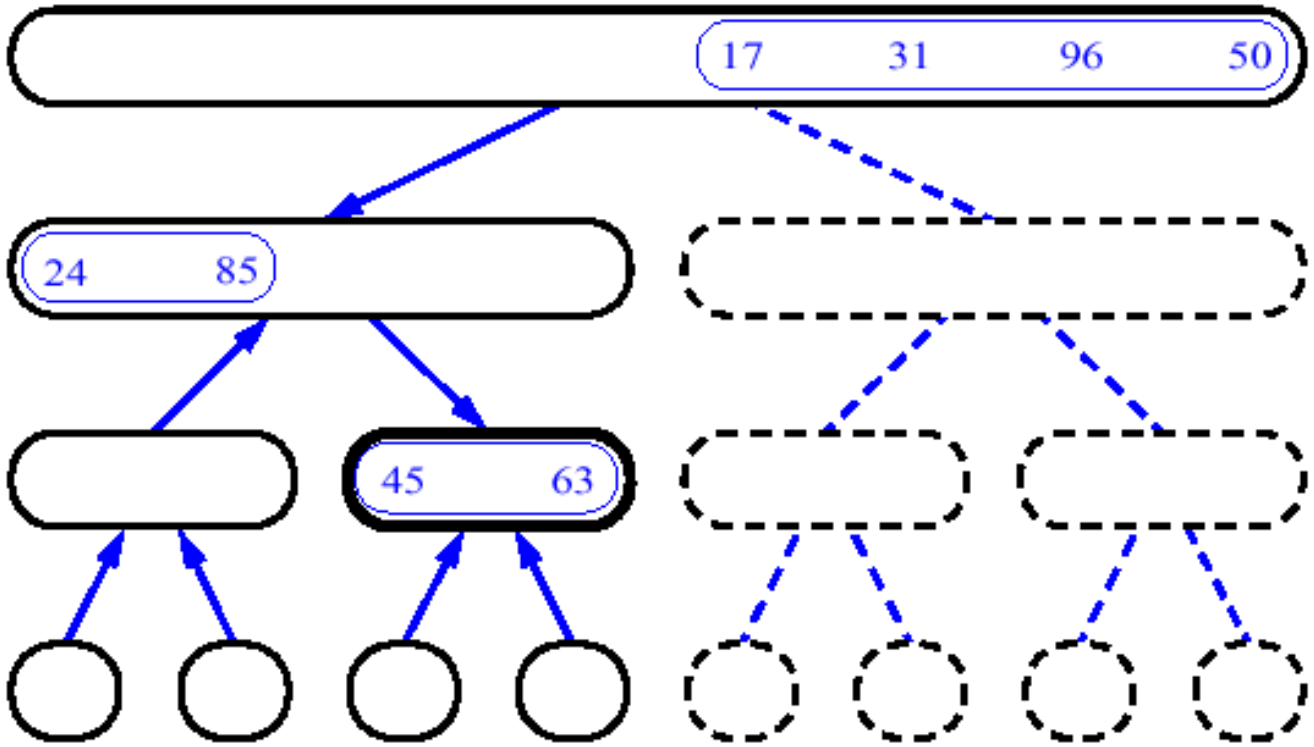
MergeSort (Example) - 13



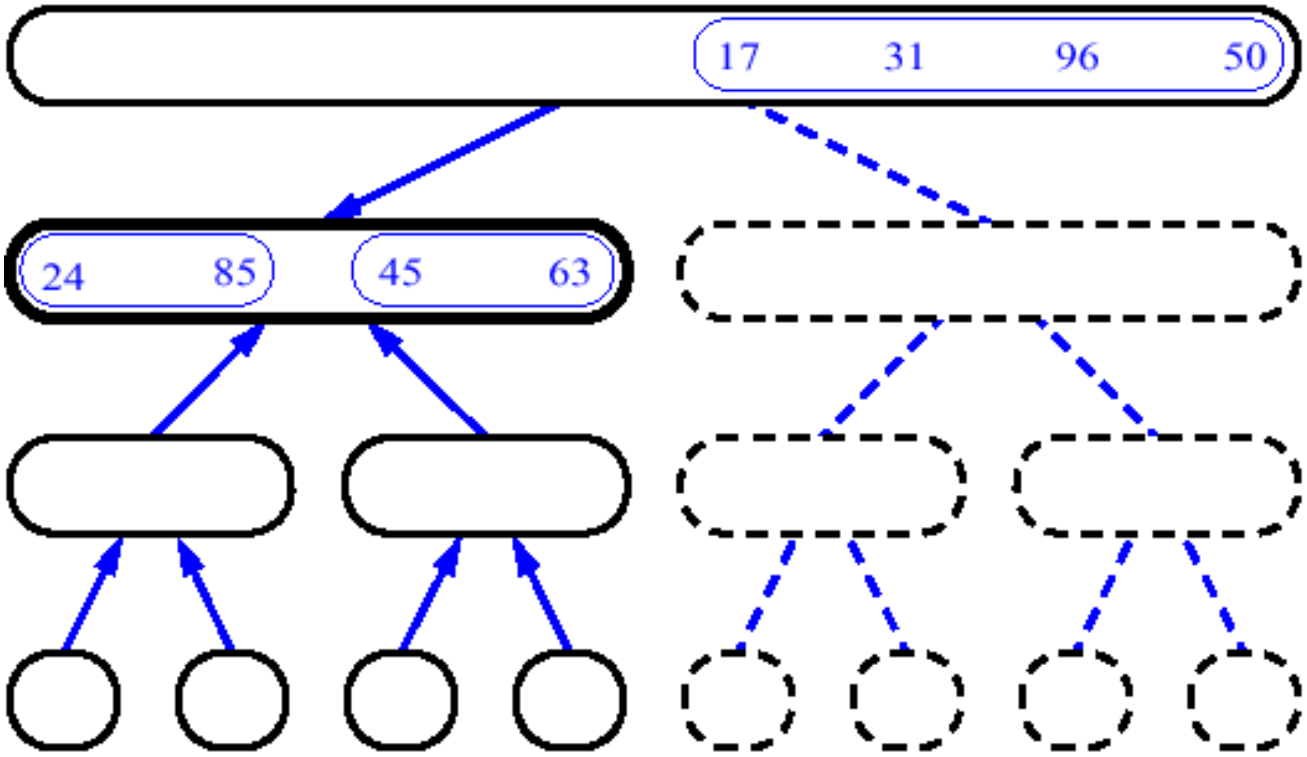
MergeSort (Example) - 14



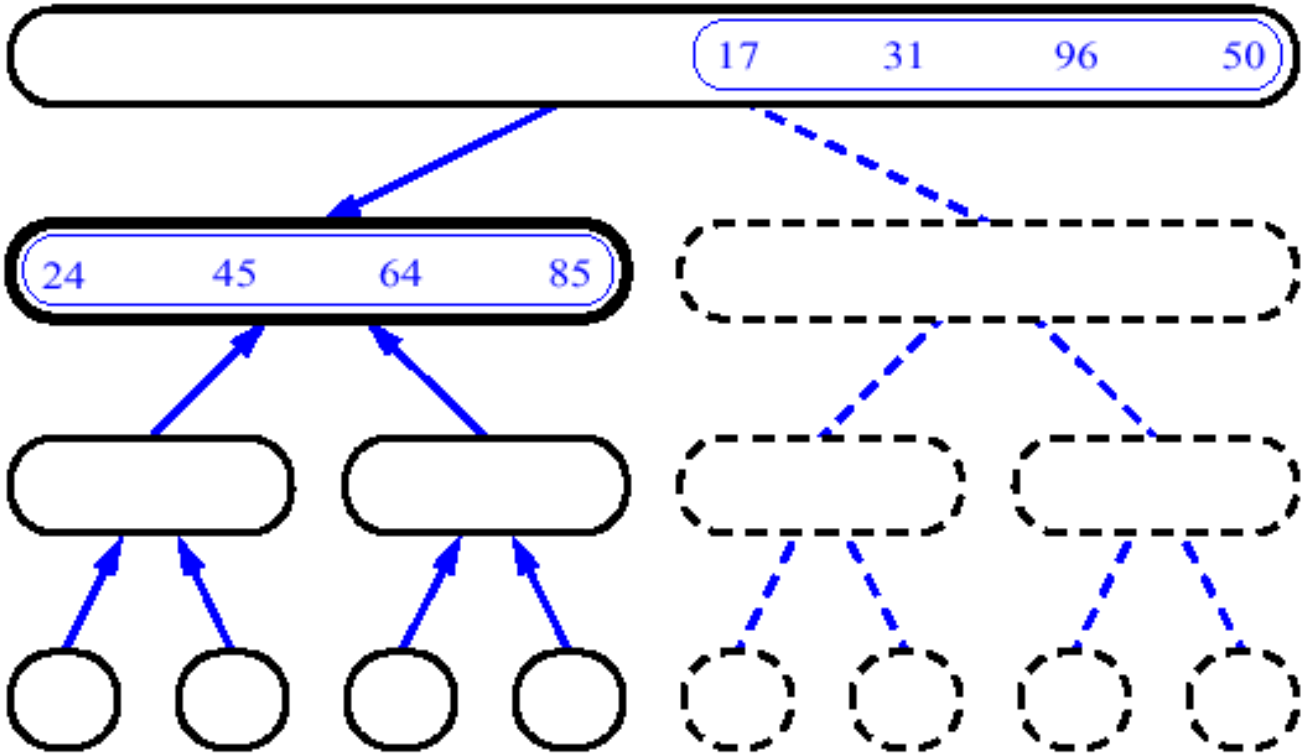
MergeSort (Example) - 15



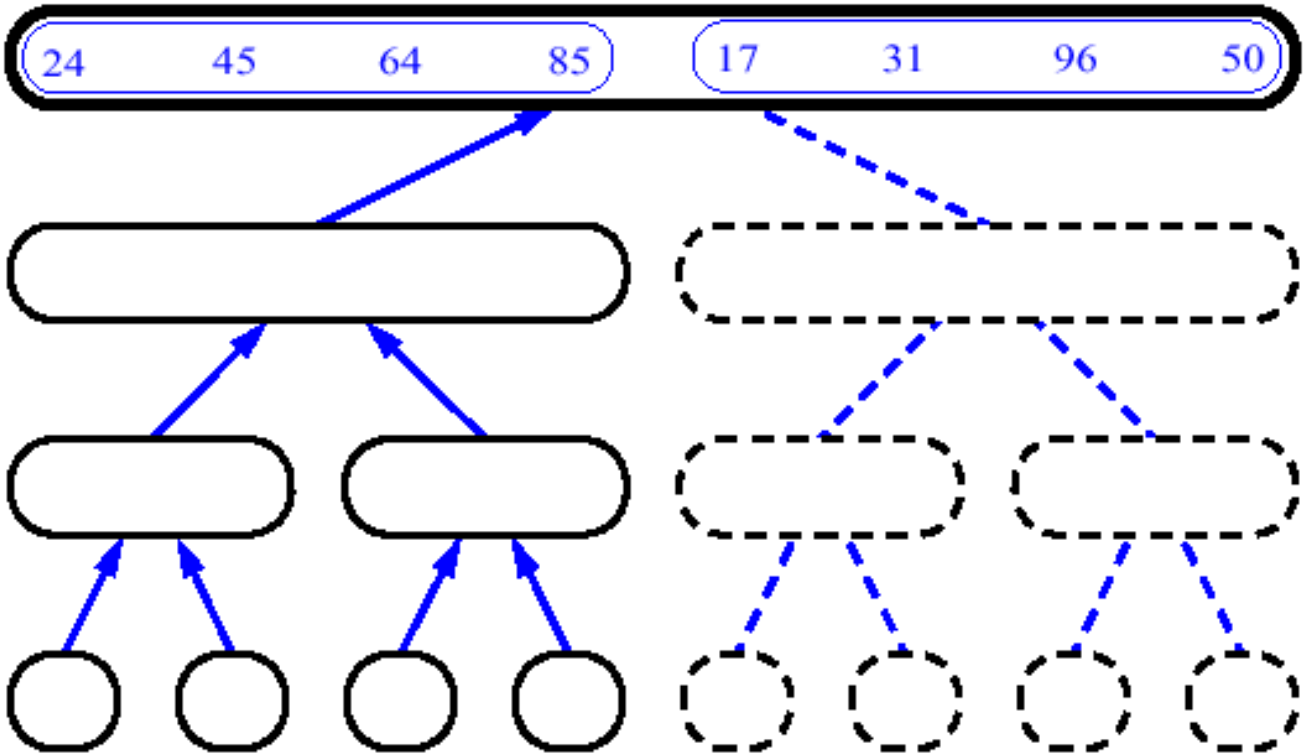
MergeSort (Example) - 16



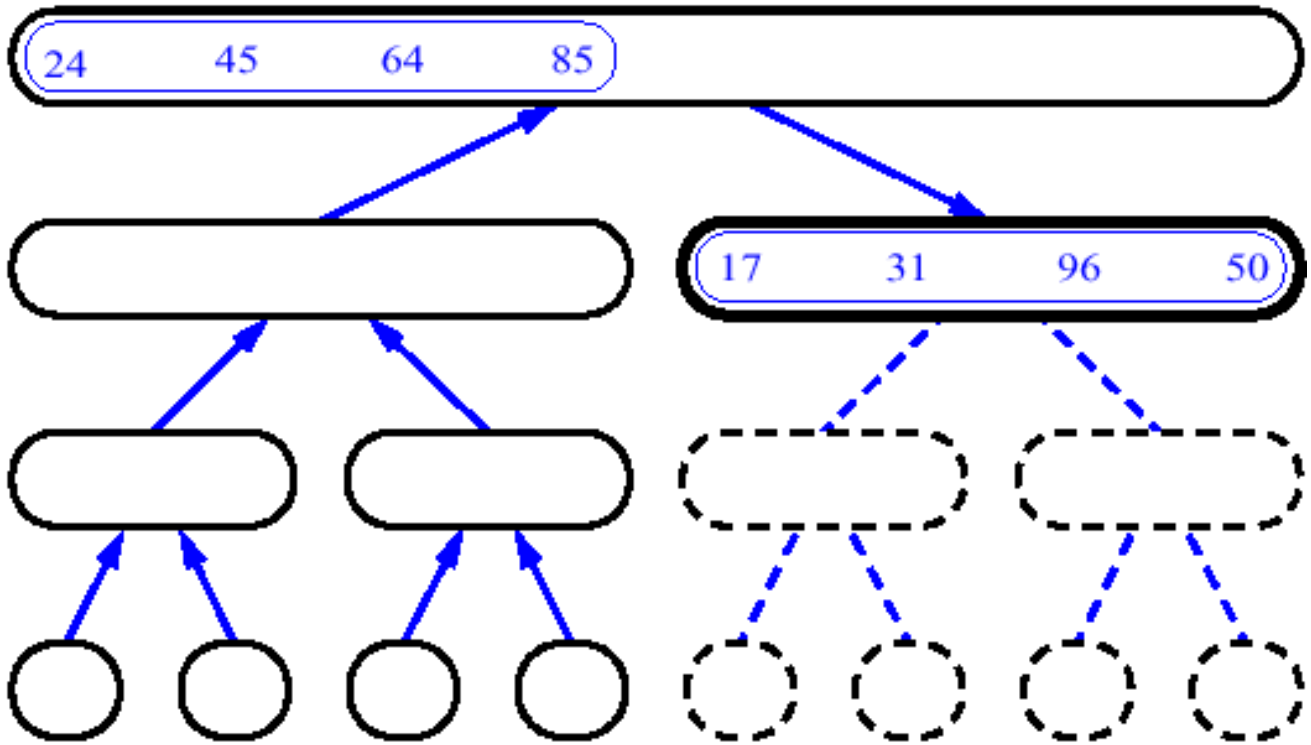
MergeSort (Example) - 17



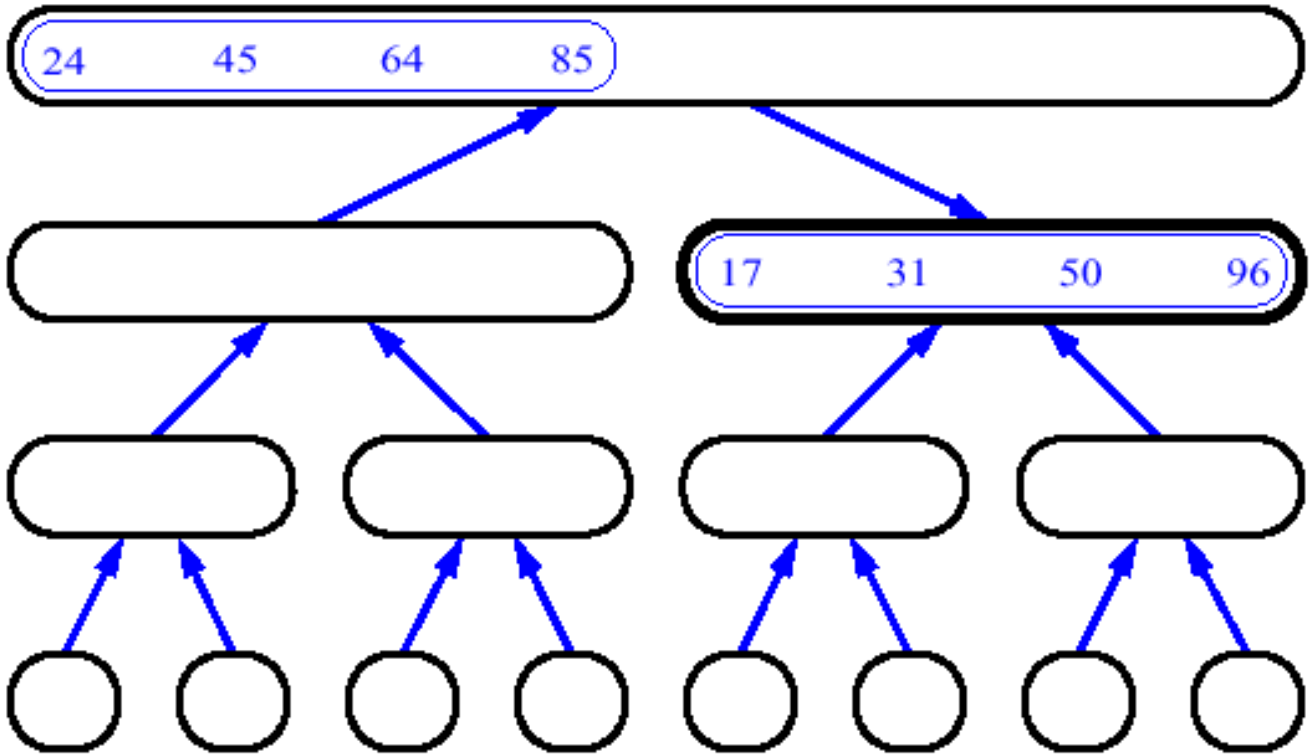
MergeSort (Example) - 18



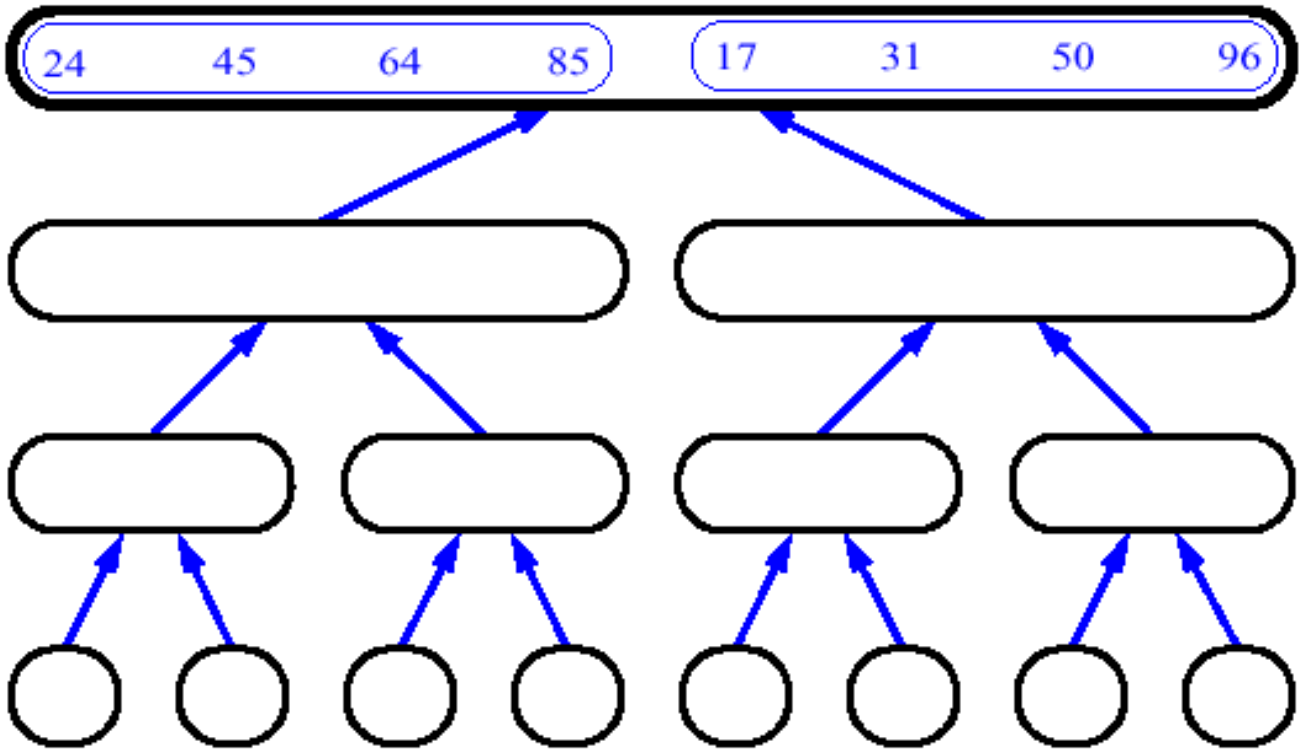
MergeSort (Example) - 19



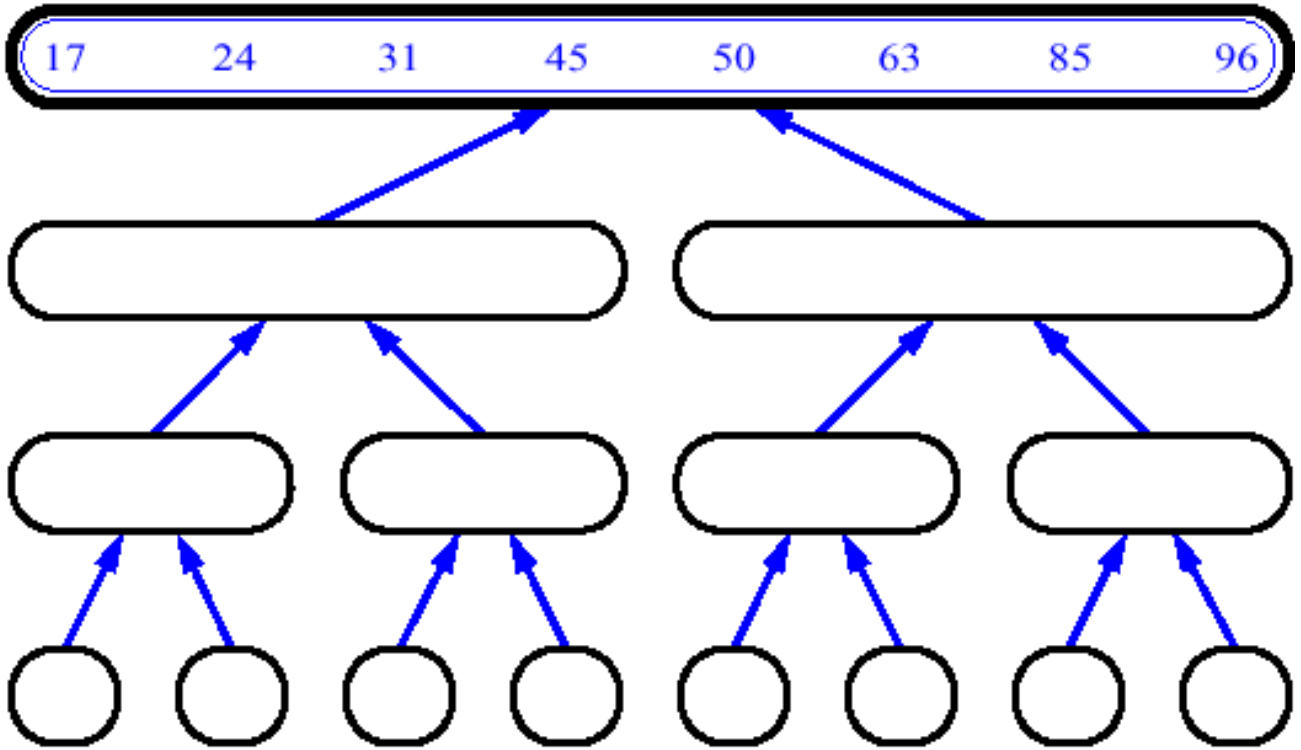
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



COMPLEXITY ANALYSIS

SORT	WORST CASE	AVERAGE CASE	BEST CASE
BUBBLE SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
INSERTION SORT	$O(n^2)$	$O(n^2)$	$O(n)$
MERGE SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QUICK SORT	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3) No disk may be placed on top of a smaller disk.

Algorithm

Tower (N, Beg, Aux, End)

1. If N = 1, then
    a) Write Beg→End
    b) Return

    else

    a)Call Tower (N-1, Beg, End, Aux)

b)Call Tower (1, Beg, Aux ,End)

c)Call Tower (N-1, Aux, Beg, End)

end

N=3 //N is No. of Disk
Tower(3,A,B,C)
where A is Beg
B is Aux.
C is End
Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 =}$
7 steps.

Time Complexity:- O(2^n)which is exponentioal.
This is computationally very expensive. Most of the recursive programs take exponential time, and that is why it is very hard to write them iteratively.

Space complexity is O(n).
After these analyses, we can see that time complexity of this algorithm is exponential but space complexity is linear.