

DATA STRUCTURE AND ALGORITHM

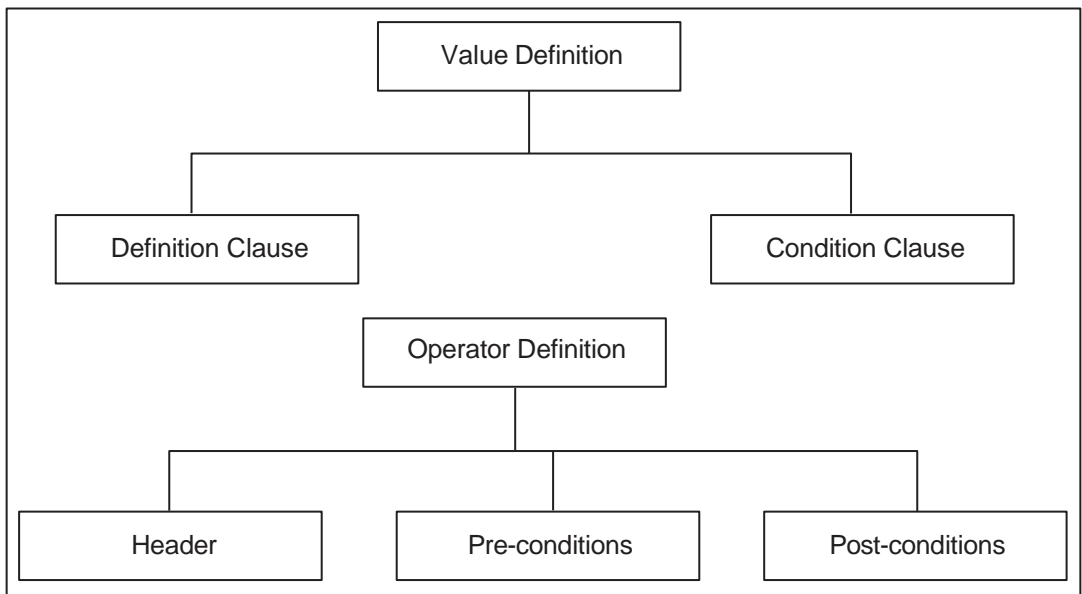
INTRODUCTION

A computer is a machine that processes information and data. The study of data structures includes how this information and data is related and how to use this data efficiently for various applications.

Logical properties of a data type are specified by abstract data type, or ADT.

- A data type consists of:
 - i) Values (value definition) and
 - ii) Set of operations on these values (Operator definition).
- These values and the operations on them form a mathematical construct that can be implemented using hardware or software data structures.

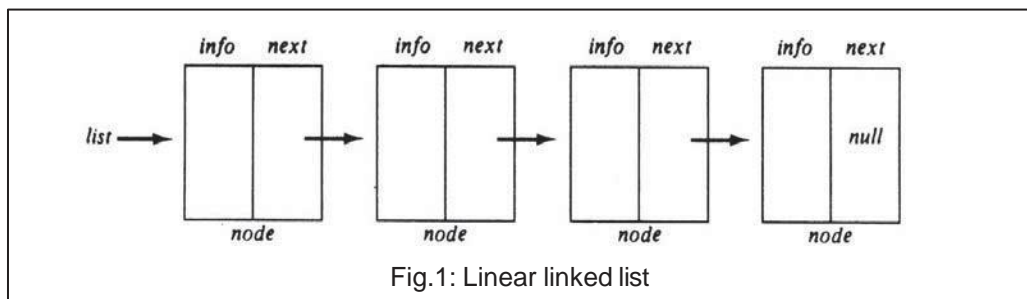
Note : ADT is not concerned with implementation details.



- Any two values in an ADT are equal if and only if the values of their components are equal.

LINKED LISTS

- There are certain drawbacks of using sequential storage to represent arrays.
 - A fixed amount of storage remains allocated to the array even when the structure is actually using a smaller amount or possibly no storage at all.
 - Only fixed amount of storage may be allocated, making overflow a possibility.
- In a sequential representation, the items of an array are implicitly ordered by the sequential order of storage. If $\text{arr}[x]$ represents an element of an array, the next element will be $\text{arr}[x + 1]$
- Suppose that the items of an array were explicitly ordered i.e. each item contained within itself the address of the next element. Such an explicit ordering gives rise to a data structure known as a Linear linked list as shown in the figure below.



- Each item in the list is called a node and it contains two fields.
 - Information field: It holds the actual element on the list.
 - The next address field: It contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a **pointer**.
- The entire linked list is accessed from an external pointer list that points to (contains the address of) the first node in the list.
- The next address field of the last node in the list contains a special value called null. This is not a valid address and is used to signal the end of a list.
- The list with no nodes on it is called the empty list or the null list. The value of the external pointer list to such a list is the null pointer. A list can be initialized to the empty list by the operation $\text{list} = \text{null}$.

INSERTING AND REMOVING NODES FROM A LIST

Insertion

A list is a dynamic data structure. The number of nodes on a list may vary as elements are inserted and removed. The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

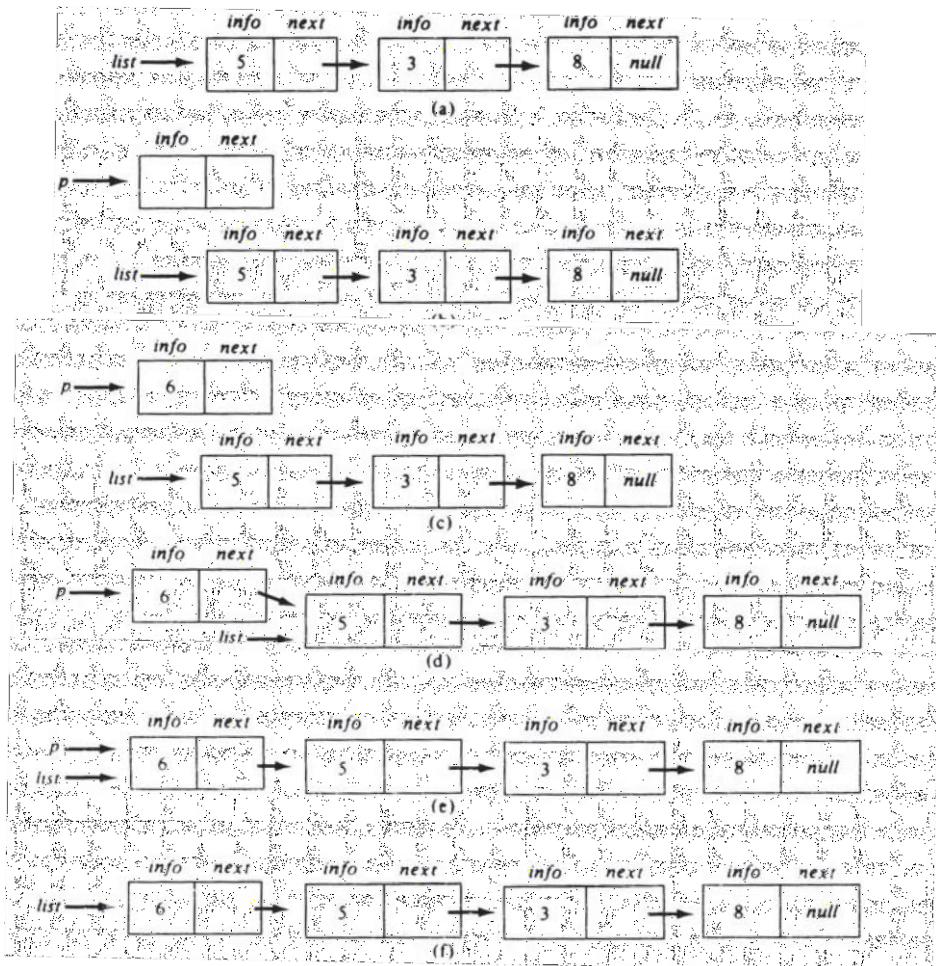


Fig.2: Adding an element to the front of a list

Example :

Suppose that we are given a list of integers as illustrated in figure 2(a) and we have to add the integer 6 to the front of that list i.e. we want the list in figure 2(f).

- Assume the existence of mechanism for obtaining empty nodes.
The operation `p = getnode();`
obtains an empty node and sets the contents of a variable named `p` to the address of that node. The value of `p` is then a pointer to this newly allocated node.
- In figure 2(b), we see the list and the new node after performing the `getnode` operation. The next step is to insert the integer 6 into the info portion of the newly allocated node. This is done by `info(p) = 6;` (The result is shown in figure 2(c))
- After setting the info portion of node (`p`), it is necessary to set the next portion of that node. Since node (`p`) is to be inserted at the front of the list, the node that follows should be the current first node on the list. Since the variable `list` contains the address of that first node, node (`p`) can be added to the list by performing the operation `next(p) = list;`
[This operation places the value of `list` (which is the address of first node on the list) into the next field of node(`p`)]

→ At this point, *p* points to the list with the additional item included. However, since *list* is the external pointer to the desired list, its value must be modified to the address of the new first node of the list. This can be done by performing the operation *list = p*; which changes the value of *list* to the value of *p*. figure 2(e) illustrates the result of this operation.

→ Generalized Algorithm

```

p = getnode( ) [allocate memory for new node]
info(p) = x;
next(p) = list;
list = p;

```

Deletion

The following figure shows the process of removing the first node of a nonempty list and storing the value of its info field into a variable *x*. The initial step and final step are shown in figure 3(a) and 3(f) respectively.

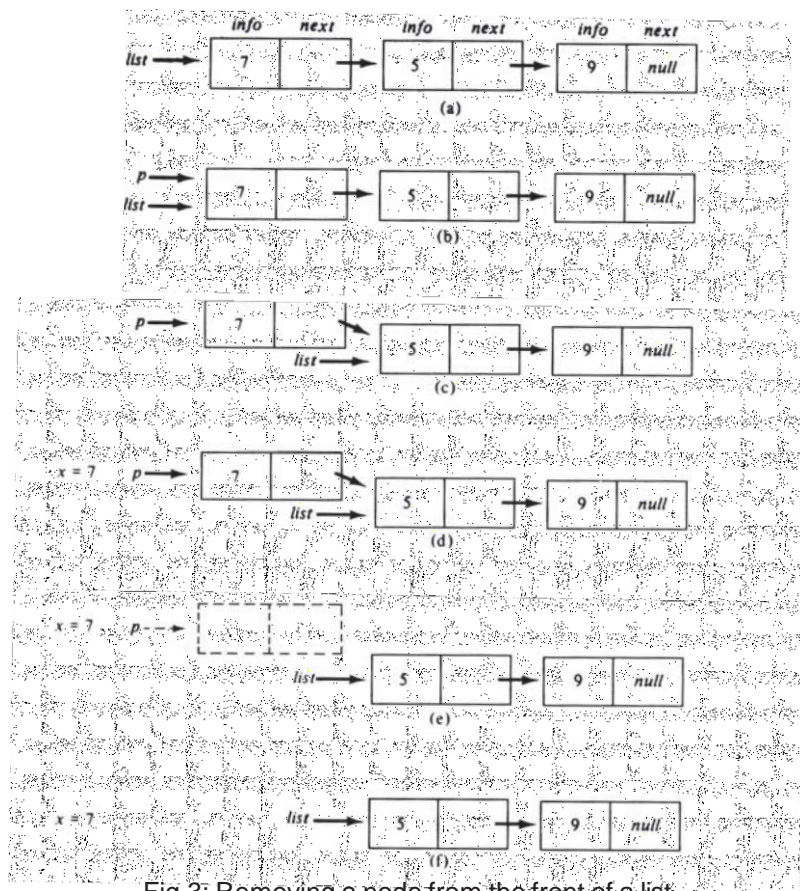


Fig.3: Removing a node from the front of a list

(Note: This process is exactly opposite to the process of adding a node)

→ To obtain figure 3(d) from figure 3(a) the following operations are performed.

```
p = list;
list = next(p);
x = info(p);
```

→ In figure 3(d), p still points to the node that was formally first on the list. That node is currently useless because it is no longer on the list and its information has been stored in x.

(Note : The node is not considered to be on the list despite the fact that next(p) points to a node on the list, since there is no way to reach node(p) from the external pointer *list*)

→ During the process of removing the first node from the list, the variable p is used as an auxiliary variable. The starting and ending configurations of the list make no reference to p. But once the value of p is changed there is no way to access the node at all, since neither an external pointer nor a next field contains its address. Therefore the node is currently useless and cannot be reused.

- The getnode creates a new node, whereas freenode destroys a node. Figure 3(e) and 3(f) depicts a node being freed.

STACK



A Stack is an ordered collection of items into which new items may be inserted or deleted only at one end, called the top of the stack.

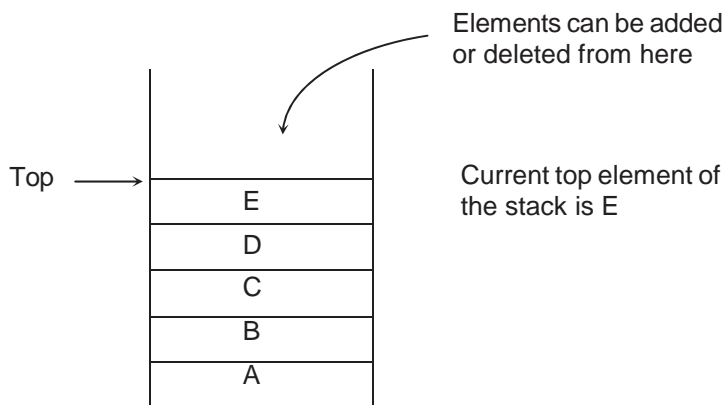
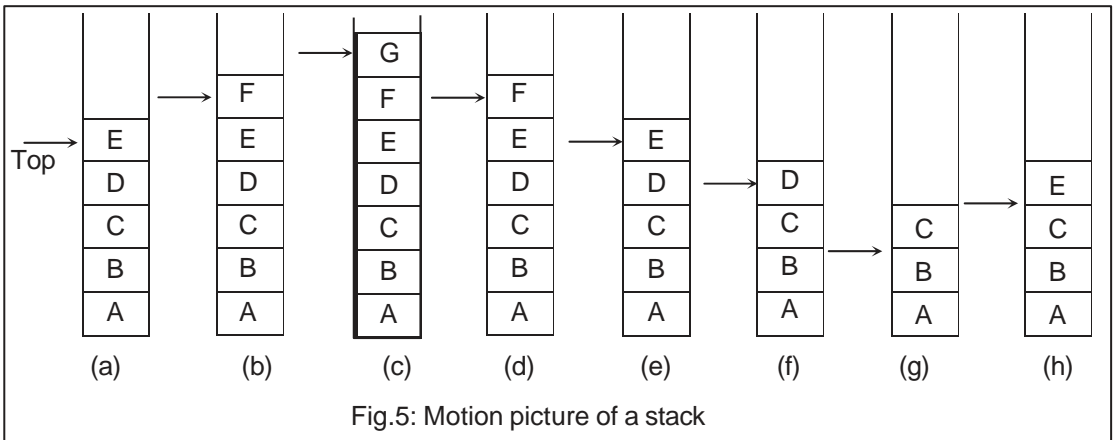


Fig.4: Stack containing stack terms

- A stack is different from an array as it has the provision for the insertion and deletions of items, thus a stack is a dynamic, constantly changing object.



- The last element inserted into a stack is the first element deleted. Thus stack is called a last-in, First-out (or LIFO) list.
- One cannot distinguish between frame (a) and frame (e) by looking at the stack's state at the two instances. No record is kept on the stack of the fact that two items had been pushed and popped in the meantime.
- The true picture of a stack is given by a view from the top looking down, rather than from a side looking in. Thus there is no perceptible difference between frame (e) and (d). One can compare the stacks at the two frames by removing all the elements on both stacks and compare them individually.

OPERATIONS ON STACK



When an item is added to a stack, it is pushed onto the stack.
When an item is removed, it is popped from the stack.

- Given a stack s and an item i , performing the operation $\text{push}(s, i)$ adds item i to the top of stack s .
- The operation

$$i = \text{pop}(s);$$
removes the element at the top of s and assigns its value to i .



As a push operation adds elements onto a stack, a stack is sometimes called a pushdown list.

- There is no upper limit on the number of items that may be kept in a stack, though a pop operation cannot be applied to the empty stack as such a stack has no elements to pop.

- The operation `empty(s)` determines whether stack `s` is empty or not. If the stack is empty; `empty(s)` returns the value `TRUE`, otherwise it will return the value `FALSE`.
- The operation `stacktop(s)` returns the top element of stack `s`. The `stacktop(s)` operation can be decomposed into a `pop` and a `push`. It is also called as `peek` operation.


```
i = stacktop(s);
is equivalent to
i = pop(s);
push (s, i);
```

`stacktop` is not defined for an empty stack.



The result of an illegal attempt to `pop` or access an item from an empty stack is called **underflow**.

Operation	Function
<code>Push(s, i)</code>	Adds item <code>i</code> on the top of stack <code>s</code>
<code>pop(s)</code>	Removes the top element of stack <code>s</code>
<code>empty(s)</code>	Determines whether or not stack <code>s</code> is empty
<code>stacktop(s)</code>	Returns the top element of stack <code>s</code> without popping it

APPLICATIONS OF STACKS

The place where stacks are frequently used is in evaluation of arithmetic expression. An arithmetic expression consists of operands and operators. The operands can be numeric values or numeric variables. The operators used in an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentiation.



If the operator is situated in between the operands, the representation is called **infix**.
 If the operator precedes the operands, the representation is called **prefix**.
 If the operator follows the operands, the representation is called **postfix**.

To fix the order of evaluation of an expression each language assigns to each operator a priority. Even after assigning priorities how can a compiler accept an expression and produce correct code?

A polish mathematician Jan Lukasiewicz suggested a notation called **Polish** notation, which gives two alternatives to represent an arithmetic expression. The notations are **prefix** and **postfix** notations. The 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. Hence, parentheses are not required while writing expressions in **Polish** notation.

While writing an arithmetic expression, the operator symbol is usually placed between two operands. For example,

$$A + B * C$$

$$A * B - C$$

$$A + B / C - D$$

$$A \$ B + C$$

This way of representing arithmetic expressions is called infix notation. While evaluating an infix expression usually the following operator precedence is used:

- Highest priority: Exponentiation ($\$$)
- Next highest priority: Multiplication ($*$) and Division ($/$)
- Lowest priority: Addition ($+$) and Subtraction ($-$)

If we wish to override these priorities we can do so by using a pair of parentheses as shown below.

$$(A + B) * C$$

$$A * (B - C)$$

$$(A + B) / (C - D)$$

The expressions within a pair of parentheses are always evaluated earlier than other operations.

In prefix notation the operator comes before the operands. For example, consider an arithmetic expression expressed in infix notation as shown below:

$$A + B$$

This expression in prefix form would be represented as follows:

$$+AB$$

The same expression in postfix form would be represented as follows:

$$AB+$$

In postfix notation, the operator follows the two operands.

The prefix and postfix expressions have three features:

- The operands maintain the same order as in the equivalent infix expression.
- Parentheses are not needed to designate the expressions unambiguously.
- While evaluating the expression the priority of the operators is irrelevant.
- Polish / Prefix notation (Infix to Prefix)
 - $(A + B) * C = [+AB] * C = * + ABC$
 - $A + (B * C) = A + [* BC] = +A * BC$
 - $(A + B) / (C - D) = [+AB] / [-CD] = /+AB - CD$
- Reverse polish / postfix Notation (Infix to Postfix)
 - $A \$ B * C - D + E / F / (G + H) = AB \$ C * D - EF / GH + /+$
 - $((A + B) * C - (D - E)) \$ (F + G) = AB + C * DE - - FG + \$$
 - $A - B / (C * D \$ E) = ABCDE \$ * / -$
- The computer 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.

Evaluation of a Postfix Expression

Let P be an arithmetic expression written in postfix notation. The following algorithm evaluates P using a STACK to hold operands.

Algorithm:

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 “)” is encountered.
3. If an operand is encountered, push it on STACK.
4. If an operator is encountered, then
 - a) Remove the two top elements of STACK, where X is the top element and Y is the next-to-top element.
 - b) Evaluate Y operator X.
 - c) Place the result of (b) back on STACK.
 [End of if structure]
 [End of 2 loop]
5. Set VALUE equal to the top element on STACK.
6. Exit.

Example of above concept:

Consider the following arithmetic expression P written in postfix notation.

P: 4, 5, 2, +, *, 16, 4, /, –

Below are the contents of STACK as each element of P is scanned.

Symbol Scanned	STACK
4	4
5	4, 5
2	4, 5, 2
+	4, 7
*	28
16	28, 16
4	28, 16, 4
/	28, 4
–	24
)	

The final number in STACK is 24, which is assigned to VALUE when “)” is scanned and thus is the final value of P.

Infix to Postfix Conversion

Let Q be an arithmetic expression written in infix notation Q, besides containing operands and operators also contain left and right parentheses.

The following algorithm is used to transform the infix expression Q to its equivalent postfix expression P. It uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK. To begin, push a left parenthesis onto STACK and add a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm:

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 \otimes 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 \otimes .
 - b) Add \otimes 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. [Do not add the left parenthesis to P]
 [End of If structure]
7. Exit

Example of above concept

Consider the following arithmetic infix expression Q:

$$Q: A + (B * C - (D / E \uparrow F) * G) * H$$

Convert the above infix expression into its equivalent postfix expression P.

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
\uparrow	(+ (- (/ \uparrow	A B C * D E
F	(+ (- (/ \uparrow	A B C * D E F
)	(+ (-	A B C * D E F \uparrow /
*	(+ (- *	A B C * D E F \uparrow /
G	(+ (- *	A B C * D E F \uparrow / G
)	(+	A B C * D E F \uparrow / G * -
*	(+ *	A B C * D E F \uparrow / G * -
H	(+ *	A B C * D E F \uparrow / G * - H
)		A B C * D E F \uparrow / G * - H * +

After the last step, the STACK is empty and the postfix equivalent of Q is

$$P: A B C * D E F \uparrow / G * - H * +$$

QUEUES



A queue is an ordered collection of items from which items may be deleted only at one end called the front of the queue and may be inserted only at the other end, called the rear of the queue.

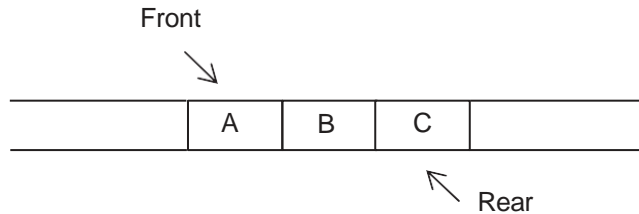


Fig: Queue

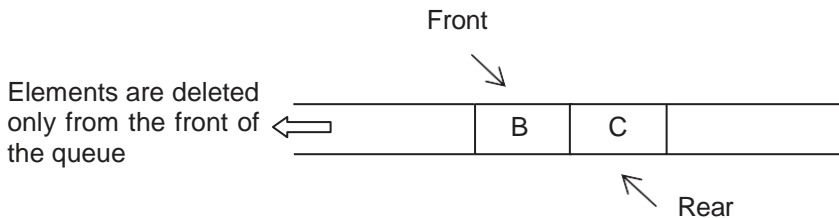


Fig. Queue after an element is deleted

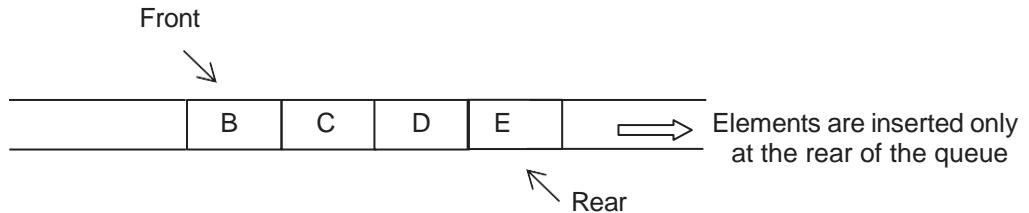


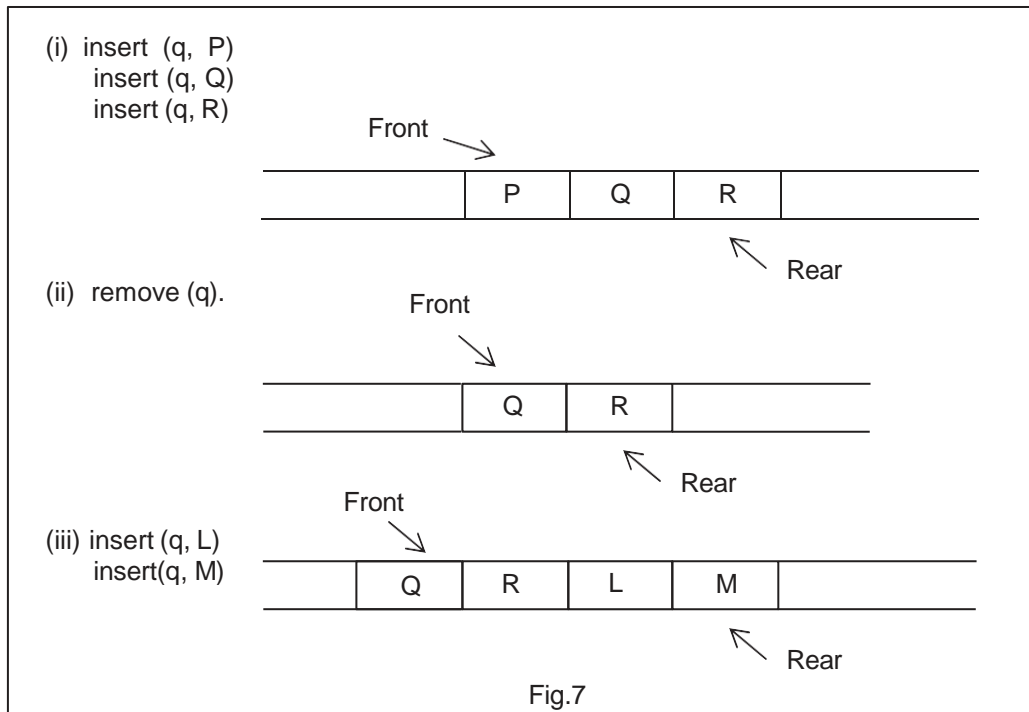
Fig.6: Queue after element is inserted

The first element inserted into a queue is the first element to be removed. Thus a queue is called a

FIFO (First-In-First-Out) list.

Operations on Queues

Operation	Function
insert (q,x)	inserts item x at the rear of queue q.
x = remove (q)	deletes the front element from queue q and sets x to its contents
empty (q)	returns false or true depending on whether or not the queue contains any elements

Example:

- As there is no limit to the number of elements a queue may contain, an insert operation can always be performed.
- There is no way to remove an element from a queue containing no elements, thus a remove operation can be applied only if the queue is non-empty.
- The result of an illegal attempt to remove an element from an empty queue is called underflow

PRIORITY QUEUE

The priority queue is a data structure in which the intrinsic ordering of the elements determines the results of its basic operation.

Types of priority queue

- An ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.
- A descending priority queue is collection of items into which items can be inserted arbitrarily and it allows deletion of only the largest item.

Priority queues have the following rules:

- (i) An element of higher priority is processed before any element of lower priority.
- (ii) Two elements with the same priority are processed according to the order in which they were added to the queue.

Prototype of Priority Queue:

A prototype of a priority queue is a timesharing system: programs of high priority are processed first and programs with the same priority form a standard queue.

Representation of a Priority Queue:**1. One way list representation of a priority queue**

- (a) Each node in the list will contain three items of information: an information field INFO, priority number PRN and a link number LINK.
- (b) A node x precedes a node y in the list
 - (i) When x has higher priority than y or
 - (ii) When both have the same priority but x was added to the list before y.

Thus the order in the one-way list corresponds to the order of the priority queue.

Example:

Figure 8 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 9 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK.

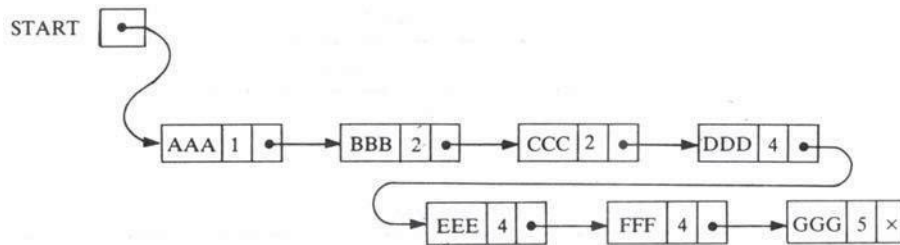


Fig.8

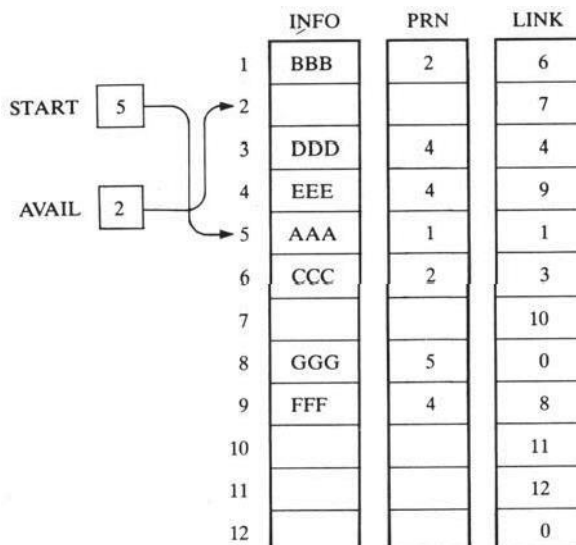


Fig.9

The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue. The outline of the algorithm follows.

Algorithm 1

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]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM
4. Exit.

The details of the algorithm, including the possibility of underflow, are left as an exercise.

Adding an element to our priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element. An outline of the algorithm follows.

Algorithm 2

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.

The above insertion algorithm may be pictured as a weighted object “sinking” through layers of elements until it meets an element with a heavier weight.

The details of the above algorithm are left as an exercise. The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traveling the list, one must also keep track of the address of the node preceding the node being accessed.

Example:

Consider the priority queue in figure 8. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in figure 10. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

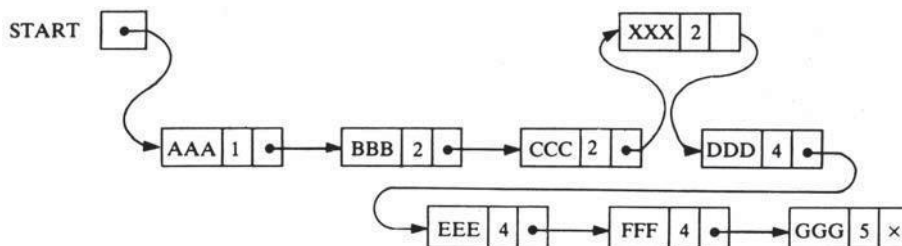


Fig.10

2. Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. a indicates this representation for the priority queue in b. Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

	FRONT	REAR		1	2	3	4	5	6
1	2	2	1		AAA				
2	1	3	2	BBB	CCC	XXX			
3	0	0	3						
4	5	1	4	FFF				DDD	EEE
5	4	4	5				GGG		

Fig.11

The following are outlines of algorithms for deleting and inserting elements in a priority queue that is maintained in memory by a two-dimensional array QUEUE, as above. The details of the algorithms are left as exercises.

Algorithm 3

The algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
Find the smallest K such that FRONT[K] \neq NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit

Algorithm 4

This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

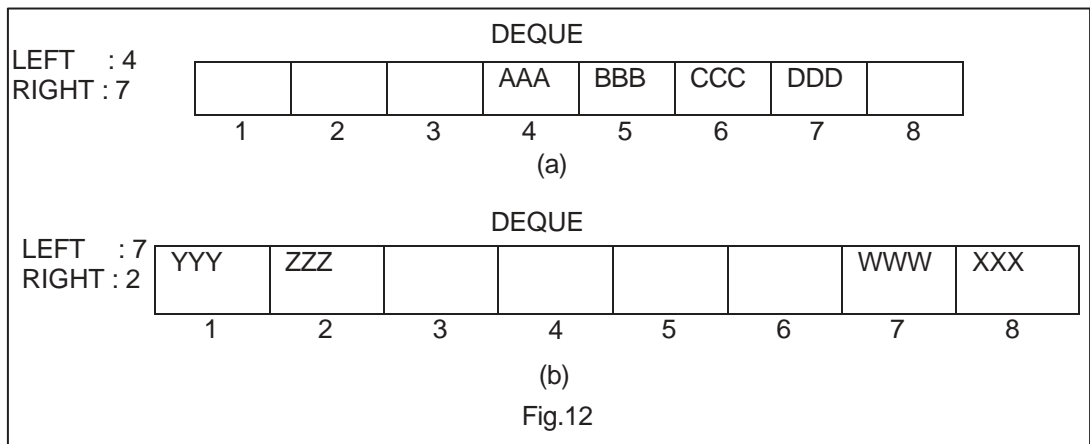
The array representation of a priority queue is more time-efficient than the one-way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation, overflow occurs when the number of elements on any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the total capacity.

DEQUES



A deque is a linear list in which elements can be added or removed at either end but not in the middle.

- Deque is maintained by a circular array DEQUE with pointer LEFT and RIGHT, which point to the two ends of the deque.
- The condition $LEFT = NULL$ will be used to indicate that a deque is empty.



- An input–restricted deque allows insertion at only one end of the list but allows deletion at both ends of the list.
- An output–restricted deque allows deletion at only one end of the list but allows insertion at both ends of the list.

Abstract Data Types

A useful tool for specifying the logical properties of a data type is the abstract data type or ADT. Fundamentally, a data type is a collection of values and a set of operations on those values. The collection and the operations form a mathematical construct that may be implemented using a particular hardware or software data structure. The term "abstract data type" refers to the basic mathematical concept that defines the data type.



In defining an abstract data type as a mathematical concept, we are not concerned with space or time efficiency. Those are implementation details. It is not necessary to implement a particular ADT on a particular piece of hardware or using a particular software system.

For example, we have already seen that the ADT integer is not universally implementable. Nevertheless, by specifying the mathematical and logical properties of a data type or structure, the ADT is useful guideline to implementers and a useful tool to programmers who wish to use the data type correctly.

Queue as an abstract Data type

The representation of a queue as an abstract data type is straightforward. We use eltype to denote the type of the queue element and parameterize the queue type with eltype.

```

abstract typedef <<eltype>> QUEUE(eltype) ;
abstract empty(q)
QUEUE (eltype) q ;
postcondition empty == (len(q) = 0) ;

abstract eltype remove (q)
QUEUE (eltype) q ;
precondition    empty (q) == FALSE ;
postcondition    remove == first (q') ;
                  q == sub (q', 1, len(q') - 1) ;

abstract insert (q, elt)
QUEUE (eltype) q ;
eltype elt ;
postcondition q == q' + <elt> ;

```

LINKED IMPLEMENTATION OF STACKS

- The operation of adding an element to the front of a linked list is similar to that of pushing an element onto a stack. The new item is addressed as the only immediately accessible item in a collection – in both cases.

Note: A stack can be accessed only through its top element and a list can be accessed only from the pointer to its first element and the operation of removing the first element from a linked list is similar to popping a stack.



A stack may be represented by a Linear Linked List. The first node of the list is the top of the stack.

If an external pointer s points to such a linked list, the operation $\text{push}(s, x)$ is implemented as :

```

p = getnode() ;
info(p) = x;
next(p) = s;
s = p

```

Consider the following figure,

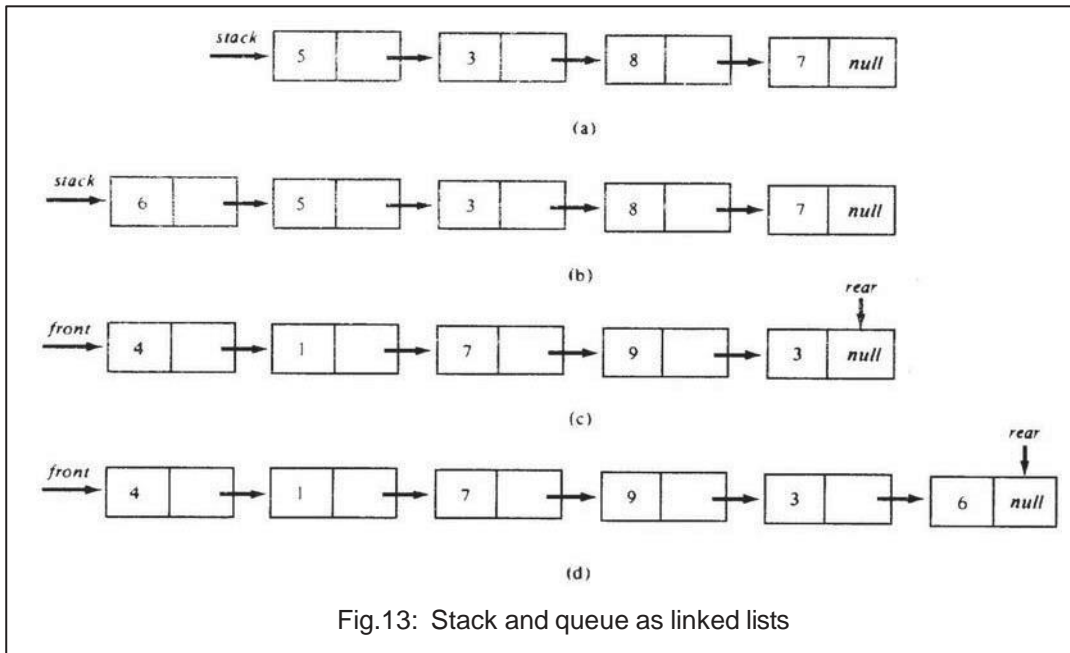


Fig.13: Stack and queue as linked lists

- Figure 13(a) shows a stack implemented as a linked list and figure 13(b) shows the same stack after another element has been pushed onto it.
- Advantages:
 - All stacks being used by a program can share the same available list and when a stack needs a node, it can obtain it from the single available list.
 - When a stack no longer needs a node, it returns the node to that same available list (of free nodes).
 - As long as the total amount of space needed by all the stacks at any one time is less than the amount of space initially available to them, each stack is able to grow and shrink to any size.
 - No space has been pre-allocated to any single stack and no stack is using space that it does not need.

LINKED IMPLEMENTATION OF QUEUES

- We know that items are deleted from the front of the queue and inserted at the rear. Let a pointer to the first element of a list represent the front of the queue and another pointer to the last element of the list represents the rear of the queue as shown in figure 13(c). Figure 13(d) illustrates the same queue after a new item has been inserted.
- Under the list representation, a queue consists of a list and two pointers q.front and q.rear.
The operation `empty(q)` and `x = remove(q)` are completely similar to `empty(s)` and `x = pop(s)`. (With the pointer q.front replacing s).

- We need to be careful when the last element is removed from a queue. In that case, `q.rear` must also be set to null, because in an empty queue both `q.front` and `q.rear` must be null.

Disadvantages of representing a stack or queue by a linked list:

- A node in a linked list occupies more storage than a corresponding element in an array, since two pieces of information per element are necessary in a list node (info and next), whereas only one piece of information is needed in the array implementation.
(Note: The space used for a list node is usually not twice the space used by an array element, since the elements in such a list usually consist of structures with many subfields.)
- Additional time is needed in managing the available list. Addition and deletion of each element from a stack or a queue involves a corresponding deletion or addition to the available list.

Comparison between linked lists and stacks:



Linked Lists are not only used for implementing stacks and queues but also other data structures. An item is accessed in a linked list by traversing the list from its beginning. A list implementation requires n operations to access the n^{th} item in a group while an array requires only a single operation.

- While inserting or deleting an element in the middle of a group of elements, the list is superior to an array.
For example: We want to insert an element x between the 3rd and 4th elements in an array of size 10 that currently contains seven items ($X[0]$ through $X[6]$). Items 7 through 4 must first be moved one slot each and the new element inserted in the newly available position 4, as shown in the figure below in figure 14.

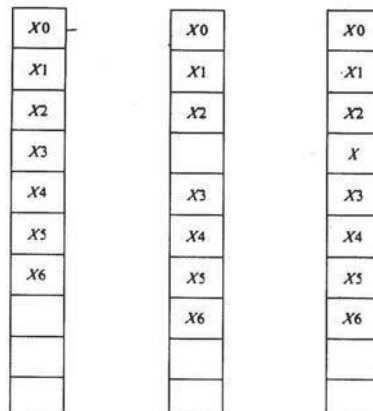
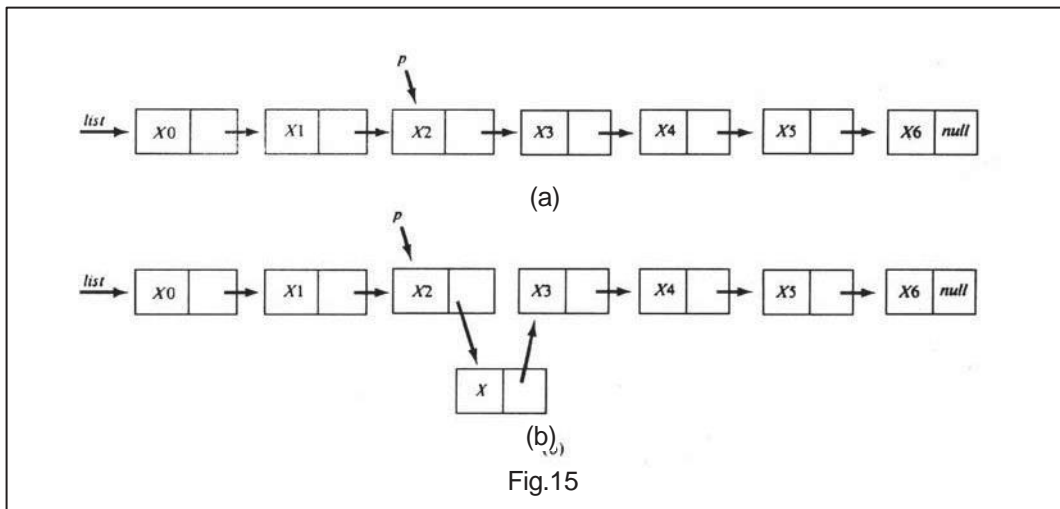


Fig.14

- In this case, insertion of one item involves moving four items in addition to the insertion itself. If the array contained 500 or 1000 elements, a correspondingly larger number of elements would have to be moved. Similarly to delete an element from an array without leaving any gap, all the elements beyond the element deleted must be moved one position.
- Suppose the items are stored as a list; if p points to an element of the list, then to insert a new element following p , involves allocating a node, inserting the information and adjusting two pointers. The amount of work required is independent of the size of the list or of where the new element is being inserted. See figure 15 for details.



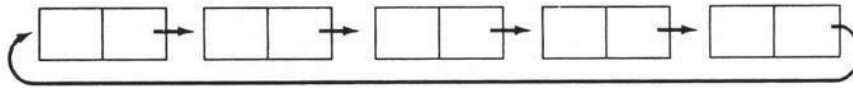
- An item can be inserted only after a given node and not before the node, because there is no way to proceed from a given node to its predecessor in a linear list. One would have to traverse the list from the beginning in that case.
- For insertion of an item before a desired node, the next field of its predecessor must be changed to point to the newly allocated node.
- To delete a node from a linear list it is insufficient to be given a pointer to that node. Since the next field of the node's predecessor must be changed to point to the node's successor and there is no direct way of reaching the predecessor of a given node. Therefore a pointer to the predecessor node must also be given.

Thus an ordered list is somewhat more efficient than an unordered list in implementing a priority queue.

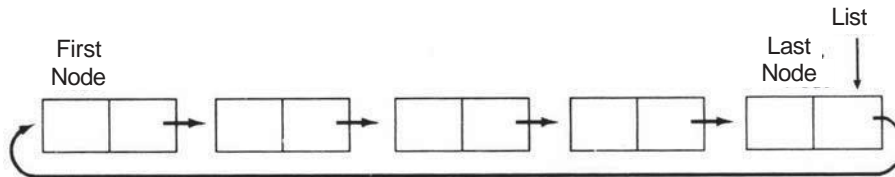
- Advantages of a list over an array for implementing a priority queue:
 - i) No shifting of elements or gaps is necessary in a list.
 - ii) An item can be inserted into a list without moving any other items, whereas this is impossible for an array unless extra space is left empty.

OTHER LIST STRUCTURES

1. Circular Lists



(a) Circular list



(b) First and last nodes of a circular list

Fig.16

- Let p be a pointer to a node in a linear list, but we cannot reach any of the nodes that precede $\text{node}(p)$. If a list is traversed, the external pointer to the list must be preserved to be able to reference the list again.



Suppose that a small change is made to the structure of a linear list, so that the next field in the last node contains a pointer back to the first node rather than the null pointer. Such a list is called a circular list.

- Circular list is shown in figure 16(a). From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we end up at the starting point.

Note: Circular list does not have a natural "first" or "last" node. We must therefore establish a first and last node by convention.

One Convention: Let the external pointer to the circular list point to the last node, and to allow the following node to be the first node as shown in figure 16(b). If p is an external pointer to a circular list, this convention allows access to the last node of the list by referencing $\text{node}(p)$ and to the first node of the list by referencing $\text{node}(\text{next}(p))$.

- A circular list can be used to represent a stack or queue.

2. Doubly Linked Lists

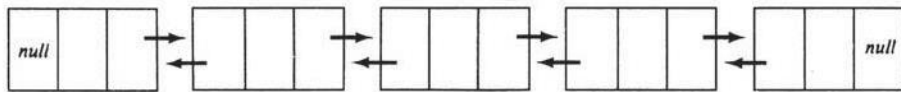
Although a circular linked list has advantages over a linear list, it still has some drawbacks, as follows:

One cannot traverse such a list backward, nor can a node be deleted from a circular linked list, given only a pointer to that node.

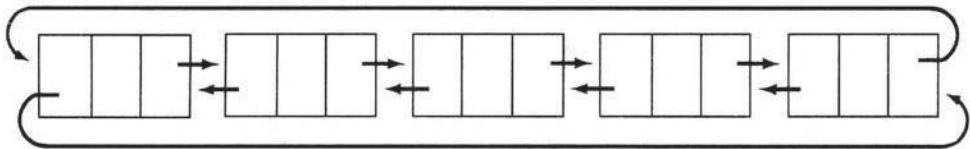
In cases where these facilities are required, the suitable data structure is a doubly linked list.



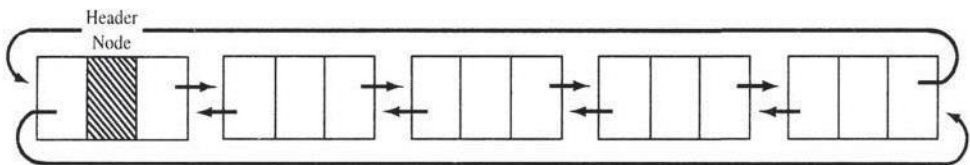
- Each node contains two pointers: one to its predecessor and another to its successor.
- A doubly linked list may be either linear or circular and may or may not contain a header node.



(a) A linear doubly linked list.



(b) A circular doubly linked list without a header.



(c) A circular doubly linked list with a header.

Fig.17: Doubly linked lists

- Nodes on a doubly linked list consists of three fields
 - i) An info field that contains the information stored in the node.
 - ii) Left and right fields that contains pointers to the nodes on either side.

The dynamic implementation of an array implementation of each node is shown in figure below.

Array Implementation	Dynamic Implementation
<pre>struct nodetype { int info ; int left, right ; }; struct nodetype node[NUMNODES];</pre>	<pre>struct node { int info ; struct node *left, *right ; }; typedef struct node *NODEPTR ;</pre>

Note: The available list for such a set of nodes in the dynamic implementation need not be doubly linked, since it is not traversed in both the directions. The available list may be linked together by using either a left or a right pointer.

PERFORMANCE ANALYSIS



The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

- Performance evaluation can be loosely divided into two major phases.
 - 1) A priori estimates (performance analysis)
 - 2) A posteriori testing (performance measurement)
- Space complexity

Consider the following 3 algorithms:

- Algo1 abc (a, b, c)


```
{
    return a + b + b * c + (a + b - c) / (a + b) + 5.0 ;
}
```
- Algo2 sum (a, n)


```
{
    s := 0.0 ;
    for i = 1 to n do
        s := s + a[i] ;
    return s ;
}
```
- Algo3 Rsum (a, n)


```
{
    if (n ≤ 0) then return 0.0 ;
    else return Rsum (a, n - 1) + a[n] ;
}
```

The space needed by each of these algorithms is seen to the sum of the following components:

- A fixed part that is independent of the characteristics of the input and output
 - for example : number and size of inputs.
 - It typically includes the instruction space i.e. space for code.
 - Space for simple variables, fixed size component variables and space for constants and so on.
- A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved.
 - It includes space needed by referenced variables.
 - Recursion stack space.

The space required $S(P)$ of any algorithm P can therefore be written as $S(P) = C + S_P$ (instance characteristics) where C is a constant.

Note:

- While analyzing the space complexity of an algorithm, we concentrate on estimating S_P (instance characteristics)
- For any given problem, we first need to determine which instance characteristics to use to measure the space requirement.

- For example:
For Algo1, the problem instance is characterized by the specific values of a, b and c. Making the assumption that one word is adequate to store the values of each of a, b, c and the result, we observe that the space needed by algorithm abc is independent of the instance characteristics.
 \therefore Sp (instance characteristics) = 0

Time Complexity



The time $T(p)$ taken by a program p is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. This runtime is denoted by T_p (instance characteristics).

To obtain such time at instance, we need to know how many computations i.e. additions, multiplications, subtractions, divisions performed by an algorithm. But still obtaining a correct formula for each algorithm is an impossible task, since the time needed for an addition, subtraction, multiplication and so on, often depends on the numbers being added, subtracted, multiplied and so on. The value of $T_p(n)$ for any given input size n can be obtained only experimentally. The program is typed, compiled and run on a particular machine. The execution time is physically clocked and $T_p(n)$ is obtained.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways.

- i) In the first method, we introduce a new variable, count, into the program. This is a global variable with initial value 0. Statements to increment count by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed; count is incremented by the step count of that statement.
- ii) The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This number is often arrived at, by first determining the number of steps per execution (s/e) of the statement and the total number of times (i.e. frequency) each statement is executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement. By combining these two quantities the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for entire algorithm is obtained.

For example

The Fibonacci sequence of numbers is 0, 1, 1, 2, 3, 5,

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence f_0 , then $f_0 = 0$, $f_1 = 1$ and in general,

$$f_n = f_{n-1} + f_{n-2} ; \quad n \geq 2$$

Algorithm Fibonacci (n)

// compute the n^{th} Fibonacci number

```
{  
    if ( $n \leq 1$ ) then  
        write (n) ;  
    else  
    {  
        fnm2 := 0 ; fnm1 := 1 ;
```



```

    for i := 2 to n do
    {
        fn := fnm1 + fnm2 ;
        fnm2 := fnm1; fnm1 := fn ;
    }
    write (fn) ;
}

```

To analyze the time complexity of this algorithm, we need to consider two cases: (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1 , lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8 and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n + 1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total number of steps for the case $n > 1$ is therefore $4n + 1$.

ASYMPTOTIC NOTATION (O , Ω , θ)

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2, \dots\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes. It is sometimes convenient, however, to abuse asymptotic notation in a variety of ways. For example, the notation is easily extended to the domain of real numbers or alternatively, restricted to a subset of the natural numbers. It is important, however, to understand the precise meaning of the notation so that when it is abused, it is not misused. This section defines the basic asymptotic notations and also introduces some common abuses.

(1) Big oh (O)

The θ -notation asymptotically bounds a function from above and below. When we have only an **asymptotic upper bound**, we use O -notation. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$



The function $f(n) = O(g(n))$ (read as 'f of n is big oh of g of n') iff there exist positive constants c and n_0 , such that $f(n) \leq c * g(n)$ for all $n, n \geq n_0$.

We use O -notation to give an upper bound on a function, to within a constant factor. Figure 18(b) shows the intuition behind O -notation. For all values n to the right of n_0 , the value of the function $f(n)$ is on or below $g(n)$.

To indicate that a function $f(n)$ is a member of $O(g(n))$, we write $f(n) = O(g(n))$. Note that $f(n) = \theta(g(n))$ implies $f(n) = O(g(n))$, since θ -notation is a stronger notation than O -notation. Written set-theoretically, we have $\theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any

quadratic function $an^2 + bn + c$, where $a > 0$ is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$. What may be more surprising is that any linear function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = 1$.

When we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is.

Using O -notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm, as we shall see later, immediately yields an $O(n^2)$ upper bound on the worst-case running time: the cost of the inner loop is bounded from above by $O(1)$ (constant), the indices i and j are both at most n , and the inner loop is executed at most once for each of the n^2 pairs of values for i and j .

Since O -notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm on arbitrary inputs as well. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on every input. For example, when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n , the actual running time depends on the particular input of size n . That is, the running time is not really a function of n . What we mean when we say "the running time is $O(n^2)$ " is that the worst-case running time (which is a function of n) is $O(n^2)$, or equivalently, no matter what particular input of size n is chosen for each value of n , the running time on that set of inputs is $O(n^2)$.

For Example:

- 1) The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$.
- 2) The function $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$.
- 3) The function $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to C for any constant C and all $n \geq n_0$.

We write $O(1)$ to mean a computing time that is a constant.

- $O(n)$ is called linear
- $O(n^2)$ is called quadratic
- $O(n^3)$ is called cubic
- $O(2^n)$ is called exponential



If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n , than if it had taken $O(n)$. Similarly $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

- From the definition of $O(\text{Big-oh})$, it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$.

If $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$

Proof:

$$\begin{aligned}
 f(n) &\leq \sum_{i=0}^m |a_i| n^i \\
 &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\
 &\leq n^m \sum_{i=0}^m |a_i| \quad \text{for } n \geq 1
 \end{aligned}$$

So $f(n) = O(n^m)$

(2) Omega (Ω)

Just as O -notation provides an asymptotic upper bound on a function, Ω -notation provides an **asymptotic lower bound**. For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0$

The intuition behind Ω -notation is shown in figure 18(c). For all values n to the right of n_0 , the value of $f(n)$ is on or above $g(n)$.



The function $f(n) = \Omega(g(n))$ (read as “ f of n is omega of g of n ”) iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n, n \geq n_0$.

Theorem:

For any two functions $f(n)$ and $g(n)$, $f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Since Ω -notation describes a lower bound, when we use it to bound the best-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

The running time of insertion sort therefore falls between $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n . Moreover, these bounds are asymptotically as tight as possible: for instance, the running time of insertion sort is not $\Omega(n^2)$, since insertion sort runs in $\theta(n)$ time when the input is already sorted. It is not contradictory, however, to say that the worst-case running time of insertion sort is not $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time. When we say that the running time (no modifier) of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen for each value of n , the running time on that set of inputs is at least a constant time $g(n)$, for sufficiently large n .

For example:

- i) The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$)
- ii) The function $3n + 3 = \Omega(n)$ as $(3n + 3) \geq 3n$ for $n \geq 1$.
- iii) The function $3n + 3 = \Omega(1)$
 - As in the case of big-oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we say that $3n + 3 = \Omega(n)$ and $6 \times 2^n + n^2 = \Omega(2^n)$ we almost never say that $3n + 3 = \Omega(1)$ or $6 \times 2^n + n^2 = \Omega(1)$ even though both of these statements are correct.



If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

(3) Theta (θ)

In the previous section, we found that the worst-case running time of insertion sort is $T(n) = \theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions,

$$\theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

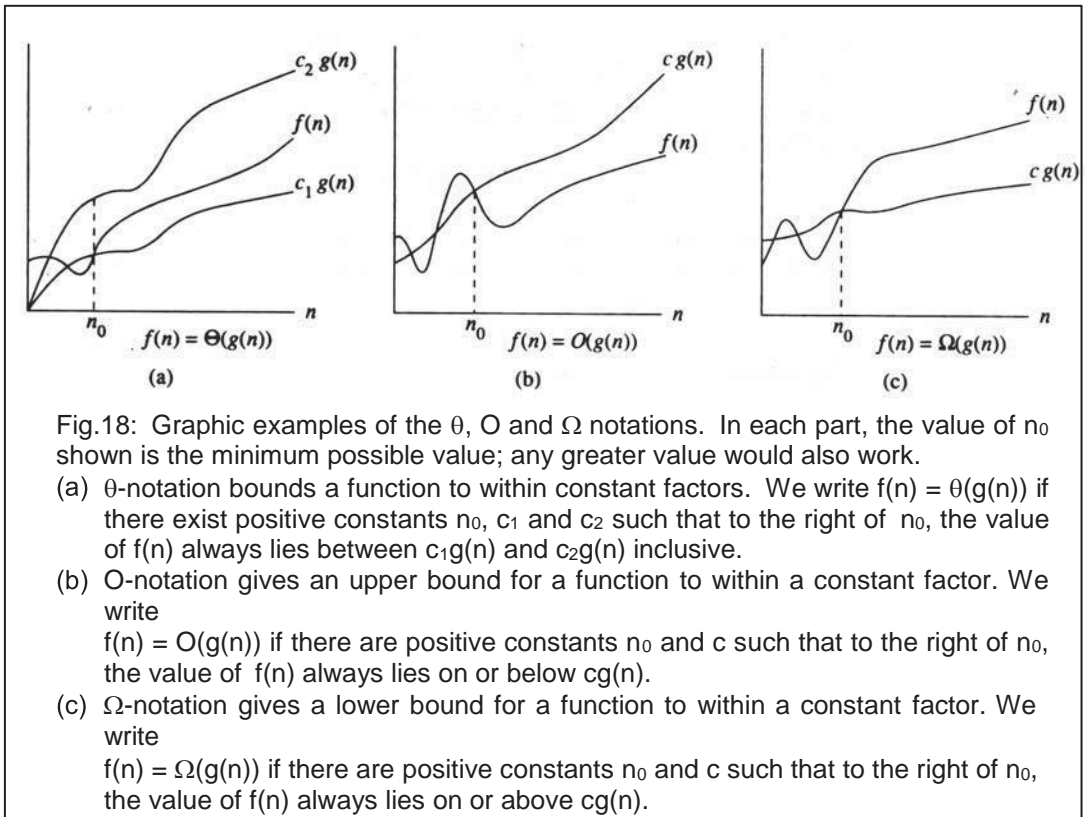


The function $f(n) = \theta(g(n))$ (read as f of n is theta of g of n) iff there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$.

A function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n . Although $\theta(g(n))$ is a set, we write “ $f(n) = \theta(g(n))$ ” to indicate that $f(n)$ is a member of $\theta(g(n))$, or “ $f(n) \in \theta(g(n))$.” This abuse of equality to denote set membership may at first be confusing but we shall see later in this section that it has advantages.

Figure 18(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \theta(g(n))$. For all values of n to the right of n_0 , the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

The definition of $\theta(g(n))$ requires that every member of $\theta(g(n))$ be **asymptotically nonnegative**, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\theta(g(n))$ is empty. We shall therefore assume that every function used within θ -notation is asymptotically nonnegative.



Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n . A tiny fraction of the highest-order term is enough to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of θ -notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

As an example, consider any quadratic function $f(n) = an^2 + bn + c$, where a , b and c are constants and $a > 0$. Throwing away the lower order terms and ignoring the constant yields $f(n) = \theta(n^2)$. Formally, to show the same thing, we take the constants $c_1 = a/4$,

$c_2 = 7a/4$, and $n_0 = 2 \cdot \max\left(\lceil |b|/a \rceil, \sqrt{\lceil |c|/a \rceil}\right)$. The reader may verify that

$0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all $n \geq n_0$. In general, for any polynomial $p(n) = \sum_{i=0}^d a_i n^i$, where the a_i are constants and $a_d > 0$, we have $p(n) = \theta(n^d)$.

Since any constant is a degree-0 polynomial, we can express any constant function as $\theta(n^0)$ or $\theta(1)$. This latter notation is a minor abuse, however, because it is not clear what variable is tending to infinity. We shall often use the notation $\theta(1)$ to mean either a constant or a constant function with respect to some variable.

For example:

- i) The function $3n + 2 = \theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 1$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$.
- ii) The function $3n + 3 = \theta(n)$
 - The theta notation is more precise than both the big-oh and omega notations. The function $f(n) = \theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.
 - If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$ then $f(n) = \theta(n^m)$

(4) Little “Oh” (o)

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ (“little-oh of g of n” as the set

$$O(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.



The function $f(n) = o(g(n))$ (read as f of n is little oh of g of n) iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for some constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for all constants $c > 0$. Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is, $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Some authors use this limit as a definition of the o-notation; the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

For example:

- i) The function $3n + 2 = o(n^2)$ since $\lim_{n \rightarrow \infty} \frac{3n + 2}{n^2} = 0$
- ii) The function $6 \times 2^n + n^2 = o(3^n)$

(5) Little “omega” (ω)

By analogy, ω -notation is to Ω -notation as o-notation is to O-notation. We use ω notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$$f(n) \in \omega(g(n)) \text{ if and only if } g(n) \in o(f(n))$$

Formally, however, we define $\omega(g(n))$ (“little-omega of g of n”)

as the set $\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$.

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$.

The relation $f(n) = \omega(g(n))$ implies that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, if the limit exists.

That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.



The function $f(n) = \omega(g(n))$ (read as f of n is little omega of g of n) iff $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

Simple rules to determine the order of functions

1. The powers of n are ordered according to the exponent. n^a is $O(n^b)$ if and only if $a \leq b$.
2. The order of $\log n$ is independent of the base taken for the logarithms; that is $\log_a n$ is $O(\log_b n)$ for all $a, b > 1$.
3. A logarithm grows more slowly than any positive power of n : $\log n$ is $O(n^a)$ for any $a > 0$, but n^a is never $O(\log n)$ for $a > 0$.
4. Any power n^a is $O(b^n)$ for all a and all $b > 1$, but b^n is never $O(n^a)$ for any $b > 1$ or for any a .
5. If $a < b$, then a^n is $O(b^n)$, but b^n is not $O(a^n)$.
6. If $f(n)$ is $O(g(n))$ and $h(n)$ is an arbitrary function, then $f(n)h(n)$ is $O(g(n)h(n))$.
7. The above rules may be applied recursively (a chain rule) by substituting any function of n for n . For example, $\log \log n$ is $O((\log n)^{1/2})$. To verify this fact, replace n by $\log n$ in the statement " $\log n$ is $O(n^{1/2})$."

The order of growth of the running time of an algorithm, gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Comparison of Functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$f(n) = \theta(g(n))$	and	$g(n) = \theta(h(n))$	imply	$f(n) = \theta(h(n))$,
$f(n) = O(g(n))$	and	$g(n) = O(h(n))$	imply	$f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$	and	$g(n) = \Omega(h(n))$	imply	$f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$	and	$g(n) = o(h(n))$	imply	$f(n) = o(h(n))$,
$f(n) = \omega(g(n))$	and	$g(n) = \omega(h(n))$	imply	$f(n) = \omega(h(n))$.

Reflexivity:

$$\begin{aligned}f(n) &= \theta(f(n)), \\f(n) &= O(f(n)), \\f(n) &= \Omega(f(n)).\end{aligned}$$

Symmetry:

$$f(n) = \theta(g(n)) \text{ if and only if } g(n) = \theta(f(n)).$$

Transpose Symmetry:

$$\begin{aligned}f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)), \\f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).\end{aligned}$$

Because these properties hold for asymptotic notations, one can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b :

$$\begin{aligned}f(n) = O(g(n)) &\approx a \leq b, \\f(n) = \Omega(g(n)) &\approx a \geq b, \\f(n) = \theta(g(n)) &\approx a = b, \\f(n) = o(g(n)) &\approx a < b, \\f(n) = \omega(g(n)) &\approx a > b,\end{aligned}$$

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy:

For any two real numbers 'a' and 'b', exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, the functions n and $n^{1+\sin n}$ cannot be compared using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Standard Notations and Common Functions

This section reviews some standard mathematical functions and notations and explores the relationships among them. It also illustrates the use of the asymptotic notations.

Monotonicity:

A function $f(n)$ is **monotonically increasing** if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is **monotonically decreasing** if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is strictly increasing if $m < n$ implies $f(m) < f(n)$ and **strictly decreasing** if $m < n$ implies $f(m) > f(n)$.

Floors and Ceilings:

For any real number x , we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x ") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x "). For all real x ,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

For any integer n ,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n,$$

and for any integer n and integers $a \neq 0$ and $b \neq 0$,

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \quad \dots(1)$$

and

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor \quad \dots(2)$$

The floor and ceiling functions are monotonically increasing.

Polynomials:

Given a positive integer d , a **polynomial in n of degree d** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^d a_i n^i,$$

where the constant a_0, a_1, \dots, a_d are the coefficients of the polynomial and $a_d \neq 0$. A polynomial is **asymptotically positive** polynomial $p(n)$ of degree d , we have $p(n) = \theta(n^d)$. For any real constant $a \geq 0$, the function n^a is monotonically increasing and for any real constant $a \leq 0$, the function n^a is monotonically decreasing. We say that a function $f(n)$ is **polynomially bounded** if $f(n) = n^{O(1)}$, which is equivalent to saying that $f(n) = O(n^k)$ for some constant k .

Exponentials:

For all real $a \neq 0$, m , and n , we have the following identities:

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= 1/a \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n} \end{aligned}$$

For all n and $a \geq 1$, the function a^n is monotonically increasing in n . When convenient, we shall assume $0^0 = 1$.

The rates of growth of polynomials and exponentials can be related by the following fact.

For all real constants a and b such that $a > 1$,

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 \quad \dots(3)$$

$n \rightarrow \infty$ a

from which we can conclude that

$$n^b = o(a^n)$$

Thus, any positive exponential function grows faster than any polynomial.

Using e to denote 2.71828..., the base of the natural logarithm function, we have for all real x ,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \dots(4)$$

where “!” denotes the factorial function defined later in this section.

For all real x , we have the inequality

$$e^x \geq 1 + x, \quad \dots(5)$$

where equality holds only when $x = 0$. When $|x| \leq 1$, we have the approximation

$$1 + x \leq e^x \leq 1 + x + x^2 \quad \dots\dots(6)$$

When $x \rightarrow 0$, the approximation of e^x by $1 + x$ is quite good:

$$e^x = 1 + x + \theta(x^2)$$

(In this equation, the asymptotic notation is used to describe the limiting behavior as $x \rightarrow 0$ rather than as $x \rightarrow \infty$.) We have for all x ,

$$\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n} \right)^n = e^x$$

Logarithms:

We shall use the following notations:

$$\begin{aligned} \lg n &= \log_2 n \text{ (binary logarithm), } \ln \\ n &= \log_e n \text{ (natural logarithm), } \lg^k n \\ &= (\lg n)^k \text{ (exponentiation),} \\ \lg \lg n &= \lg(\lg n) \text{ (composition).} \end{aligned}$$

An important notational convention we shall adopt is that logarithm functions will apply only to the next term in the formula, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. For $n > 0$ and $b > 1$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$ and n ,

$$\begin{aligned} a &= b^{\log_b a} \\ \log_c(ab) &= \log_c a + \log_c b \\ \log_b a^n &= n \log_b a \\ \log_b a &= \frac{\log_c a}{\log_c b} \\ \log_b(1/a) &= -\log_b a \\ \log_b a &= \frac{1}{\log_a b} \\ a^{\log_b n} &= n^{\log_b a}. \quad \dots\dots(7) \end{aligned}$$

Since changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor, we shall often use the notation “ $\lg n$ ” when we don’t care about constant factors, such as in O -notation. Computer scientists find 2 to be the most natural base for logarithms because so many algorithms and data structures involve splitting a problem into two parts.

There is a simple series expansion for $\ell n(1+x)$ when $|x| < 1$:

$$\ell n(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots\dots$$

We also have the following inequalities for $x > -1$:

$$\frac{x}{1+x} \leq \ell n(1+x) \leq x, \quad \dots\dots(8)$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is **polylogarithmically** bounded if $f(n) = \ell g^{O(1)} n$. We can relate the growth of polynomials and polylogarithms by substituting $\ell g n$ for n and 2^a for a in equation (3), yielding

$$\lim_{n \rightarrow \infty} \frac{\log^b n}{\log^a n} = \lim_{n \rightarrow \infty} \frac{\ell g^b n}{2^a} = 0$$

From this limit, we can conclude that

$$\ell g^b n = o(n^a)$$

for any constant $a > 0$. Thus, any positive polynomial function grows faster than any polylogarithmic function.

Factorials:

The notation $n!$ (read “ n factorial”) is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

A weak upper bound on the factorial function is $n! \leq n^n$, since each of the n terms in the factorial product is at most n . **Stirling’s approximation**,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + \theta \left(\frac{1}{n} \right) \right), \quad \text{.....(9)}$$

where e is the base of the natural logarithm, gives us a tighter upper bound and a lower bound as well.

Using Stirling’s approximation, one can prove

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \ell g(n!) &= \theta(n \ell g n). \end{aligned}$$

The following bounds also hold for all n :

$$\sqrt{2\pi n} \left(\frac{n}{e} \right)^n \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e} \right)^{n+(1/12n)} \quad \text{.....(10)}$$

The iterated logarithm function:

We use the notation $\ell g^* n$ (read “log star of n ”) to denote the iterated logarithm, which is defined as follows. Let the function $\ell g^{(i)} n$ be defined recursively for nonnegative integers i as

$$\ell g^{(i)} n = \begin{cases} n & \text{if } i = 0, \\ \ell g(\ell g^{(i-1)} n) & \text{if } i > 0 \text{ and } \ell g^{(i-1)} n > 0, \end{cases} \quad \text{(i-1)}$$

$$\ell g^{(i)} n = \text{undefined} \quad \text{if } i > 0 \text{ and } \ell g^{(i-1)} n \leq 0 \text{ or } \ell g^{(i-1)} n \text{ undefined}$$

Be sure to distinguish $\lg^{(i)}n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the i^{th} power). The iterated logarithm function is defined as

$$\lg^* n = \min \{ i \geq 0 : \lg^{(i)} n \leq 1 \}.$$

The iterated logarithm is a very slowly growing function:

$$\begin{aligned}\lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5.\end{aligned}$$

Since the number of atoms in the observable universe is estimated to be about 10^{80} , which is much less than 2^{65536} , we rarely encounter a value of n such that $\lg^* n > 5$.

Fibonacci numbers:

The **Fibonacci numbers** are defined by the following recurrence:

$$\begin{aligned}F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \quad \text{for } i \geq 2.\end{aligned}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,

Fibonacci numbers are related to the **golden ratio** ϕ and to its conjugate ϕ^{\wedge} , which are given by the following formulas:

$$\begin{aligned}\phi &= \frac{1 + \sqrt{5}}{2} && \text{.....(11)} \\ &= 1.61803\text{.....}, \\ \phi^{\wedge} &= \frac{1 - \sqrt{5}}{2} \\ &= -0.61803\text{...}.\end{aligned}$$

Specifically, we have

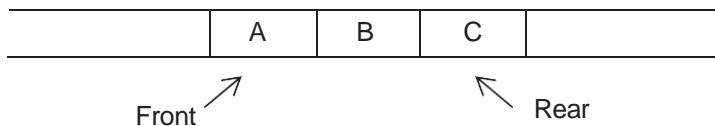
$$F_i = \frac{\phi^i - \phi^{\wedge i}}{\sqrt{5}},$$

which can be proved by induction. Since $|\phi^{\wedge}| < 1$, we have $|\phi^{\wedge i}| / \sqrt{5} < 1 / \sqrt{5} < 1/2$, so that

the ϕ^{th} Fibonacci number F_i is equal to $(\phi^i / \sqrt{5})$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

LMR (LAST MINUTE REVISION)

- A useful tool for specifying the logical properties of a data type is the abstract data type or ADT. A data type is a collection of values and a set of operations on those values. The term “abstract data type” refers to a basic mathematical concept that defines the data type.
- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end called the top of the stack. A stack is also called a LIFO list.
- Various primitive operations perform on stacks:
 1. Push: inserting an element on stack.
 2. Pop: deleting an element from the stack.
- Since the push operation is what adds elements to a stack, a stack is sometimes called a pushdown list.
If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called an empty stack.
- The nesting depth at a particular point in an expression is the number of scopes (left parentheses) that have been opened but not yet closed at that point. This is the same as the number of left parenthesis encountered whose matching right parenthesis have not yet been encountered.
- The parentheses count at a particular point in an expression as the number of left parentheses minus the number of right parentheses that have been encountered in scanning the expression from its left end upto that particular point. If the parentheses count is non-negative, it is the same as the nesting depth.
- An expression can be represented in either of the following formats:
 - i) infix : $A + B$
 - ii) postfix : $AB+$
 - iii) prefix : $+AB$
 where A and B are operands and + is an operator and all these are interconvertible with various operator precedence rules.
- A queue is an ordered collection of items from which elements may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end called the rear of the queue.)



The result of an illegal attempt to remove an element from an empty queue is called underflow.

- A priority queue is a data structure in which the intrinsic ordering of the elements determines the results of its basic operations. There are two types:
 - i) Ascending priority queue: is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.
 - ii) Descending priority queue: is similar but allows deletion of only the largest item.

- Drawbacks of stacks and queues implemented using arrays (sequential storage):
 - A fixed amount of storage remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all.
 - Also no more than that fixed amount of storage may be allocated, thus introducing the possibility of overflow.
- To avoid above drawbacks, use lists to implement stacks and queues.

Linked List

- Drawbacks of using sequential storage to represent stacks and queues :
 - (i) A fixed amount of storage remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all.
 - (ii) Only fixed amount of storage may be allocated so overflow is possible.
- Suppose that the items of a stack or a queue were explicitly ordered i.e. each item contained within itself the address of the next item. Such an explicit ordering gives rise to a data structure known as a Linear Link list.
- A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and removed.
 - Each item in the list is called a node and it contains two fields:
 - (i) Information field: It holds the actual element on the list.
 - (ii) The next address field: It contains the address of the next node in the list.Such an address, which is used to access a particular node, is known as a pointer.
 - The entire linked list is accessed from an external pointer list that points to (contains the address of) the first node in the list.
 - The next address field of the last node in the list contains a special value called as null. This is not a valid address and used to signal the end of a list.
 - The list with no nodes on it is called the empty list or the null list. The value of external pointer list to such a list is the null pointer. A list can be initialized to the empty list by the operation; list:= null.

Inserting and Removing Nodes from a List

- A list is a dynamic data structure. The number of nodes on a list may vary as elements are inserted and removed. The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

Linked implementation of Stacks

- The operation of adding an element to the front of a linked list is similar to that of pushing an element onto a stack.

A new item is added as the only immediately accessible item in a collection in both cases.
 - A stack may be represented by a Linear linked list. The first node of the list is the top of the stack.
- Advantages:
- i) All stacks being used by a program can share the same available list of nodes and when any stack needs a node, it can obtain it from the single available list.

- ii) As long as the total amount of space needed by all the stacks at any one time is less than the amount of space initially available to them, each stack is allowed to grow and shrink to any size.
- iii) No space has been pre-allocated to any single stack and no stack is using space that it does not need.

Linked implementation of Queues:

- Linked Lists are not only used for implementing stacks and queues but also as data structures. An item is accessed in a linked list by traversing the list from its beginning. A list implementation requires n operations to access the n^{th} item in a group but an array requires only a single operation.
- A priority queue implemented as an ordered linked list requires examining an average of approximately $n/2$ nodes for insertion but only one node for deletion.
- Advantages of a list over an array for implementing a priority queue.
 - i) No shifting of elements or gaps is necessary in a list.
 - ii) An item can be inserted into a list without moving any other items, whereas this is impossible for an array unless extra space is left empty.

- **Performance Analysis:**

The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion

- Performance evaluation can be loosely divided into two major phases.
 - 1) A priori estimates (performance analysis)
 - 2) A posteriori testing (performance measurement)

- **Time Complexity:**

The time $T(p)$ taken by a program p is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. This runtime is denoted by T_p (instance characteristics).

- **Asymptotic Notation (O , Ω , θ)**

- 1) **Big oh (O)**

The function $f(n) = O(g(n))$ (read as 'f of n is big oh of g of n') iff there exist positive constants C and n_0 , such that $f(n) \leq C * g(n)$ for all n , $n \geq n_0$.

- 2) **Omega (Ω)**

The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants C and n_0 such that $f(n) \geq C * g(n)$ for all n , $n \geq n_0$.

- 3) **Theta (θ)**

The function $f(n) = \theta(g(n))$ (read as f of n is theta of g of n) iff there exist positive constants C_1 , C_2 and n_0 such that $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all n , $n \geq n_0$.

4) Little “Oh” (o)

The function $f(n) = o(g(n))$ (read as f of n is little oh of g of n) iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

5) Little “omega” (ω)

The function $f(n) = \omega(g(n))$ (read as f of n is little omega of g of n) iff $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$.

• Simple rules to determine the order of functions:

1. The powers of n are ordered according to the exponent. n^a is $O(n^b)$ if and only if $a \leq b$.
2. The order of $\log n$ is independent of the base taken for the logarithms; that is $\log_a n$ is $O(\log_b n)$ for all a, $b > 1$.
3. A logarithm grows more slowly than any positive power of n: $\log n$ is $O(n^a)$ for any $a > 0$, but n^a is never $O(\log n)$ for $a > 0$.
4. Any power n^a is $O(b^n)$ for all a and all $b > 1$, but b^n is never $O(n^a)$ for any $b > 1$ or for any a.
5. If $a < b$, then a^n is $O(b^n)$, but b^n is not $O(a^n)$.
6. If $f(n)$ is $O(g(n))$ and $h(n)$ is an arbitrary function, then $f(n) h(n)$ is $O(g(n) h(n))$.
7. The above rules may be applied recursively (a chain rule) by substituting any function of n for n. For example, $\log \log n$ is $O((\log n)^{1/2})$. To verify this fact, replace n by $\log n$ in the statement “ $\log n$ is $O(n^{1/2})$ ”.



ASSIGNMENT – 1

Duration : 45 Min.

Max. Marks : 30

Q1 to Q6 carry one mark each

1. A pushdown list is a
(A) Queue (B) List
(C) Doubly link list (D) Stack.
2. Underflow is an
(A) illegal attempt to pop (B) illegal attempt to push
(C) illegal attempt to empty a stack (D) None of these
3. The time required to insert an element in a stack with linked implementation is
(A) $O(1)$ (B) $O(\log_2 n)$
(C) $O(n)$ (D) $O(n \log_2 n)$
4. Stacks cannot be used to _____
(A) allocate resources (like CPU) by the operating system
(B) evaluate an arithmetic expression in postfix form
(C) implement recursion
(D) convert a given arithmetic expression in infix form to its equivalent postfix form.
5. Select the correct postfix notation for the following :
NOT A OR NOT B AND NOT C
(A) A NOT B NOT C NOT
(B) A NOT B NOT C NOT AND OR
(C) A NOT B NOT C NOT OR
(D) ABC NOT
6. For addition of long positive integers, the best suitable data structure is _____.
(A) circular lists with header nodes
(B) circular lists without header nodes
(C) circular stack
(D) circular queue.

Q7 to Q18 carry two marks each

Consider the following stack of characters, where STACK is allocated N = 8 memory cells

STACK : A, C, D, F, K, –, –, –

Now answer Q7. and Q8.

7. Overflow _____
 (A) does not occur
 (B) will occur when STACK contain 8 elements and there is a PUSH operation
 (C) will occur when STACK contain 9 elements and there is a PUSH operation
 (D) Insufficient data.
8. C will be deleted before D from the stack when _____
 (A) overflow occurs
 (B) A will be deleted before C
 (C) F will be deleted before K
 (D) Not possible to delete
9. Suppose STACK is allocated N = 6 memory cells and initially STACK is empty. Then find the output of the following module.
 1. Set AAA := 2 and BBB := 5
 2. Call PUSH (STACK, AAA)
 Call PUSH (STACK, 4)
 Call PUSH (STACK, BBB + 2)
 Call PUSH (STACK, 9)
 Call PUSH (STACK, AAA + BBB)
 3. Repeat while TOP ≠ 0 ;
 Call POP(STACK, ITEM)
 Write : ITEM
 [End of loop.]
 4. Return.

(A) 2, 4, 7, 9, 7, –

(C) 2, 4, 9, 11, 9, –

(B) 7, 9, 7, 4, 2

(D) 9, 11, 9, 4, 2

Let a and b denote positive integers. Suppose a function Q is defined recursively as follows :

$$Q(a, b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b, b) + 1 & \text{if } b \leq a \end{cases}$$

Now answer Q10 – Q12

10. Find Q(2, 3) and Q(14, 3)
 (A) 0, 4
 (B) 1, 7
 (C) 0, 7
 (D) 1, 4
11. Find Q(5861, 7) =
 (A) 897
 (B) 837
 (C) 3786
 (D) 6742

12. The given function finds the _____
 (A) factorial upto the given limit
 (B) Fibonacci series sum upto the given limit
 (C) quotient when a is divided by b
 (D) None of these

Suppose each data structure is stored in a circular array with N memory cells numbered 1 to N, then answer Q13 and Q14.

13. The number NUMB of elements in a queue (in terms of FRONT and REAR if $\text{FRONT} \leq \text{REAR}$) is _____
 (A) $\text{FRONT} + \text{REAR} - 1$ (B) $\text{REAR} - \text{FRONT} + 1$
 (C) $\text{FRONT} - \text{REAR} - 1$ (D) $\text{REAR} + \text{FRONT} + 1$
14. The number NUMB of elements in a queue (in terms of FRONT and REAR if $\text{REAR} < \text{FRONT}$) is _____
 (A) $N + \text{REAR} - \text{FRONT} - 1$ (B) $\text{REAR} - \text{FRONT} - 1 - N$
 (C) $\text{REAR} + \text{FRONT} - 1 - N$ (D) None of these
15. If P is a pointer to a node, node (p) refers to the node pointed by p, info (p) refers to the information portion of that node, and next (p) refers to the next address portion. If next (p) is not null then info (next (p)) refers to _____.
 (A) the information of the node that follows node (p) in the list
 (B) the address portion of the node that follows node (p) in the list
 (C) the information portion of the node that follows next (p) in the list
 (D) the address portion of the node that follows next (p) in the list
16. An unordered list which is used as a priority queue, requires examining _____ for insertion and _____ for deletion (with n nodes).
 (A) one node, n/2 nodes (B) n nodes, n nodes
 (C) one node, n nodes (D) n/2 node, n nodes
17. Doubly Linked Lists _____
 i) are linear
 ii) are circular
 (A) Only i (B) i, ii
 (C) Only ii (D) None of these
18. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack at any one time when the algorithm analyzes : (() ()) (()) ?
 (A) 1 (B) 2
 (C) 3 (D) 4



TEST PAPER – 1**Duration : 30 Min.****Max. Marks : 25****Q1 to Q5 carry one mark each**

1. Abstract data type specifies
 - (A) How to store data into databases
 - (B) How to retrieve individual data elements
 - (C) Logical properties of a data type
 - (D) All the above
2. How many values can be held by an array defined as
 $A(-1..n, 1..n)$?
 - (A) n
 - (B) n^2
 - (C) $n(n+1)$
 - (D) $n(n+2)$
3. A prototype of a priority queue is a _____
 - (A) Timesharing system
 - (B) Batch processing system
 - (C) Multitasking system
 - (D) None of these
4. Suppose each data structure is stored in a circular array with N memory cells, then find the number $NUMB$ of elements in a deque in terms of $LEFT$ and $RIGHT$.
 - (A) $RIGHT - LEFT + 1 \pmod{N}$
 - (B) $RIGHT + LEFT - 1 \pmod{N}$
 - (C) $RIGHT + LEFT + 1 \pmod{N}$
 - (D) $RIGHT - LEFT - 1 \pmod{N}$
5. The five items: U, V, W, X and Y are pushed onto a stack one after the other starting from U . The stack is popped four times and each element is inserted in a queue. Then two elements are deleted from the queue and pushed back on the stack and then one item is popped from the stack. Then the popped item is _____
 - (A) U
 - (B) V
 - (C) W
 - (D) X

Q6 to Q13 carry two marks each

6. The operation
 $i = \text{pop}(s)$
 $\text{Push}(s, i)$
is equivalent to
 - (A) $i = \text{stacktop}(s)$
 - (B) $\text{empty}(s)$
 - (C) $\text{Remove}(i)$
 - (D) $\text{Add}(i)$
7. Evaluate the following postfix notation:
 $A : 6, 9, 2, +, *, 12, 3, /, -$
 - (A) 62
 - (B) 66
 - (C) 83
 - (D) None of these

8. Suppose a given algorithm requires two stacks A and B, then one can define an array STACK1 with n_1 elements for stack A and an array STACK2 with n_2 elements, for Stack B. Suppose instead, we define a single array STACK with $n = n_1 + n_2$ elements for stacks A and B together, such that we define STACK[1] as the bottom of stack A and let A "grow" to the right and STACK[n] as the bottom of stack B and let B "grow" to the left. Then which of the following statements is/are correct?
- In second implementation overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements.
 - The second technique will usually decrease the number of times overflow occurs even though we have not increased the total amount of space reserved for the two stacks.
 - In the second implementation, overflow will occur even when A or B individually have more than n_1 or n_2 elements.
- (A) Only i (B) i, ii
(C) ii, iii (D) i, iii

Let n denote a positive integer, suppose a function L is defined recursively as follows :

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ \lfloor L(\lfloor n/2 \rfloor) \rfloor + 1 & \text{if } n > 1 \end{cases}$$

(Here $\lfloor k \rfloor$ denotes the "floor" of k i.e. the greatest integer which does not exceed k .)

Answer Q9 and 10.

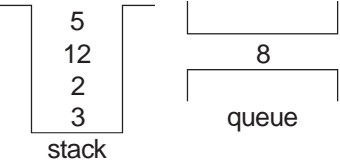
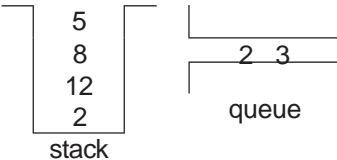
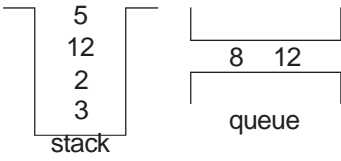
9. Find $L(25)$
(A) 25 (B) 7
(C) 4 (D) 10
10. The given function can be used to find _____
(A) $\lfloor \cos(n) \rfloor$ (B) $\lfloor \sin(n) \rfloor$
(C) $\lfloor \log_2 n \rfloor$ (D) None of these
11. Suppose a deque is stored in a circular array with N memory cells. At which of the following conditions is the deque full?
- LEFT = N and RIGHT = 1
 - LEFT = RIGHT + 1
 - LEFT = 1 and RIGHT = N
 - LEFT = RIGHT - 1 + N
- (A) i, ii (B) ii, iv
(C) ii, iii (D) i, iv

12. Consider a deque maintained by a circular array with N memory cells. When an element is added to the deque, the RIGHT is _____ and if element is deleted, then the RIGHT is _____
- (A) decreased by $(\text{mod } N)$, increase by 1
 (B) decreased by 1, not affected
 (C) increases by $1(\text{mod } N)$, not affected
 (D) increased by $1(\text{mod } N)$, decreased by $1 \text{ mod } (N)$
13. If each element on a stack were a structure occupying ten words, the addition of an eleventh word to contain a pointer _____.
- (A) increases the space requirement by 20 percent
 (B) increases the space requirement by 10 percent
 (C) decreases the space requirement by 20 percent
 (D) decreases the space requirement by 10 percent

Linked Answer Question

- a. We have a stack of integers s and a queue of integers q . Draw a picture of s and q after the following operations:

pushstack $(s, 3)$
 pushstack $(s, 12)$
 enqueue $(q, 5)$
 enqueue $(q, 8)$
 popstack (s, x)
 pushstack $(s, 2)$
 enqueue (q, x)
 dequeue (q, y)
 pushstack (s, x)
 pushstack (s, y)

- (A)  (B) 
- (C)  (D) None of these

- 14(b). With reference to (a) find the addition of all the elements from stack and queue.

- (A) Stack = 22, queue = 8
 (B) Stack = 27, queue = 5
 (C) Stack = 22, queue = 20
 (D) None of these