Basic Data Structures

Lecture delivered by:

Venkatanatha Sarma Y

Assistant Professor

MSRSAS-Bangalore

Session Objectives

- To introduce and discuss the basic types of data structures, their properties and their applications
- To discuss the need for Abstract Data Type (ADT) definitions for data structures
- To define the ADTs for the basic data structures
- To discuss the implementation and complexity of basic data structures and their operations
- To illustrate the use of Array/Vector and List ADT for implementing compound data structures like Stack and Queue

N D Gangadhar MSRSAS

Session Objectives

- To introduce and discuss the Tree ADT and its operations
- To learn about the different types of Tree Traversal algorithms
- To discuss the close relationship between Trees and Searching techniques
- To introduce the most important class of Trees, Binary Trees
- To discuss the features and applications of Binary Trees
- To introduce Binary Search Trees and discuss their use in Searching

N D Gangadhar MSRSAS

Arrays

N D Gangadhar MSRSAS

Array

- An array is the most basic data structure to visualise
- Most, if not all, computer languages provide it as a builtin

- **Definition** An array is a collection of elements stored in contiguous memory locations providing random access to its elements
  - The elements are ordered according to an index set, which is a set of integers

- Random access – accessing any element takes the same time
- Indexing could be either 0-based or 1-based

Applications of Arrays

- Despite its simplicity, an Array has wide range of applications
- Best data structure for in memory storage of infrequently changing data
- Many sorting applications use arrays explicitly or as auxilary data structures
- For implementating other and more complecated data structures
- ... such as Stack, Queue, List, and Search Tree

Data Structures and ADTs

N D Gangadhar MSRSAS

Data Structures and ADTs

- A Data Structure is
  - A collection of elements of a type
  - Along with an set of operations defined on it

- This leads to Abstract Data Types (ADTs)
  - The Data Structure defined independent of its implementation
  - This abstracted definition of a Data Structure and its Operations constitute the ADT
  - Provides a unified interface independent of the implementation details
  - May store redundent information to aid efficient operations to be performed on it

- Example: The List data structure could actually be implemented using an array

N D Gangadhar MSRSAS

The Principal ADTs

- Vector
- Stack
- Queue
- Sequence (a.k.a. List)
- Tree
- Graph

- The ADTs form heirachical dependencies; for example,

  [Array] ←Vector ← Stack ← Queue

- Most are actually classes of ADTs; for instance, the class List has as members

  LinkedList, DoublyLinkedList, CircularList, . . .

  with their own heirachical relationships (generalisation or specialisation)

N D Gangadhar MSRSAS

Object Oriented Approach to Data Structures - 1

- Object Oriented approach
  - Data Structure ADT is implemented as a Class
  - With Data Structure Operations as Class methods
- The public interface of a Queue implementation in Java:

```
public interface Queue<E> {
 /**
  * Returns the number of elements in the queue.
  * @return number of elements in the queue.
  */
  public int size();
 /**
  * Returns whether the queue is empty.
  * @return true if the queue is empty, false otherwise.
  */
  public boolean isEmpty();

Cont'd ...
```

Object Oriented Approach to Data Structures – 2

```
/**
  * Inspects the element at the front of the queue.
  * @return element at the front of the queue.
  * @exception EmptyQueueException if the queue is empty.
  */
  public E front() throws EmptyQueueException;
 /**
  * Inserts an element at the rear of the queue.
  * @param element new element to be inserted.
  */
  public void enqueue (E element);
 /**
  * Removes the element at the front of the queue.
  * @return element removed.
  * @exception EmptyQueueException if the queue is empty.
  */
  public E dequeue() throws EmptyQueueException;
}
```

Vectors

N D Gangadhar MSRSAS

The Vector ADT

- **Vector**
  - A data structure that extends the notion of an Array in which elements are stored in an order, according to their rank. The rank of an element is the number of elements preceding a given element.

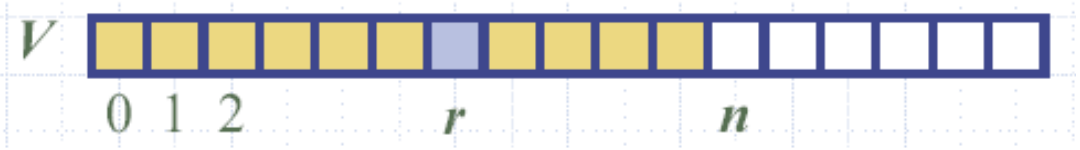
- Main Operations
  - size, isEmpty
  - elementAtRank, replaceAtRank, insertAtRank, deleteAtRank

- While the direct application is for small data base applications, other applications are indirect
  - Used as auxiliary data structures in applications
  - As component(s) of more complex data structures

N D Gangadhar MSRSAS

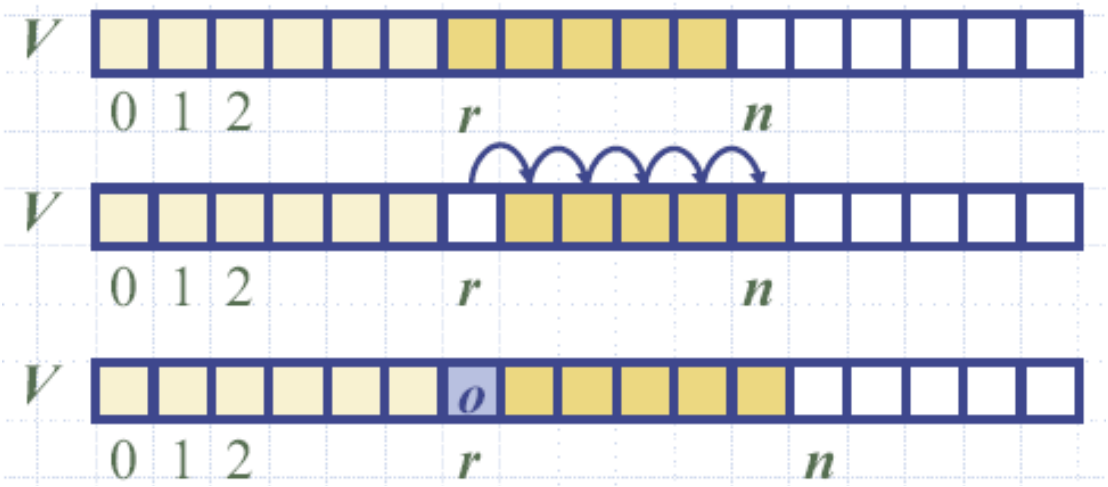
Array Based Vector Implementation

- Like many ADTs, Vector can be realised using the simple Array
  - A variable n keeps track of the current occupancy level
  - size and isEmpty are O(1)
  - elementAtRank(r) is also O(1); it simply returns V[r]
  - Similarly, replaceAtRank is O(1)



Operations insertAtRank and deleteAtRank

- insertAtRank(r) needs to shift all the elements above rank *r* to higher rank
  - The worst case is when r = 0 and this is O(n)



- deleteAtRank(r) has to shuffle down the higher rank-ed elements
  - Again, the worst case is when r= 0 and is O(n)

N D Gangadhar MSRSAS

Dynamic Sizing of Array Based Vector

- The extra tail room is important to make Array based Vector usable
  - The important thing is to optimise the sizing for multiple operations
- Operational behaviour
  - Array size needs to be dynamically adjustable
  - Helps in making insertAtRank and deleteAtRank efficient
  - The dumb strategy is to grow the Array one-element a time following each insertAtRank request
  - That would make insertAtRank take N operations to add N elements starting from Array full state
- What is cost per operation, averaged over multiple operations?
  - It is termed amortised complexity of operation
- What is a good strategy?
  - It is clear that we should keep future insertAtRank requests in mind

Array Growth Strategies

- Two popular strategies
  - Constant size increments: Increase size by a constant c number
  - Constant factor increments: Increments size by a constant factor (doubling is most often used)
- Which makes a better strategy?
  - The first difference is the space cost
  - Algorithms need to think of space costs too!
  - Main factor is that n is changing dynamically and differently for each of the strategies!
  - Amortised complexity takes this into account
  - Counts the average time per operation over a series of operations
- Constant factor increments strategy is actually efficient in the amortised sense

Amortised Complexity of Constant Size Increments

- To analyse the behaviour
  - Consider the case when the initial Array size is 1 and we perform N insertAtRank(1) operations
  - Remember, each of these is O(n), where n is the occupancy at the time of execution of the operation
  - The number of times Array is resized is

  $k = N/c$

- Total time taken for N insertAtRank(1) is

  $T(N) = N + c + 2c + ... + kc = N + k(k+1)c/2$

- So the amortised complexity of insertAtRank in this scenario is

$T(N)/N = 1 + k(k+1)c/2N = (N + c)/2c = O(N)$

N D Gangadhar MSRSAS

Amortised Complexity of Constant Factor Increments

- For simplicity take size doubling strategy (others give the same answer)
- Consider the same scenario as above

- The number of times the size is doubled can be found as

  N = 1. (2 2 … 2) = 2^k → k = log N

- Total time taken for N insertAtRank(1) is

  T(N) = N +1+2+ … +2^k = N +2^(k+1)-1 = 2N-1

- So the *amortised complexity* of insertAtRank in this scenario is

  T(N)/N = 2 – 1/N = O(1)!

N D Gangadhar MSRSAS

Shrinking a Array Based Vector

- Why should we shrink the size of a Array based Vector?
  - We can save space
  - And have greater occupancy
- What strategy to adopt?
  - From above amortised complexity analysis, it is better to follow constant factor strategy
  - Here, it is take the constant to be proper fraction; often ½
- Shrink when the occupancy falls to 1=2 of existing Array size?
  - No! Trashing phenomenon:

    insertAtRank → deleteAtRank → insertAtRank → . . .
  - Need more memory in the system behaviour— hystersis
- Solution is
  - Shrink by a factor 1=2 at an occupancy level of 1=4 current size
  - The new Array is half full
  - Double up on reaching maximum size

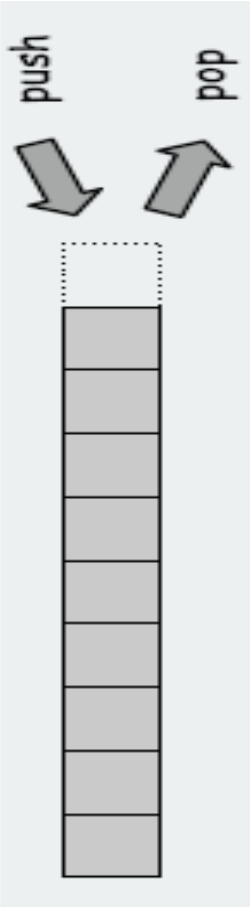
N D Gangadhar MSRSAS

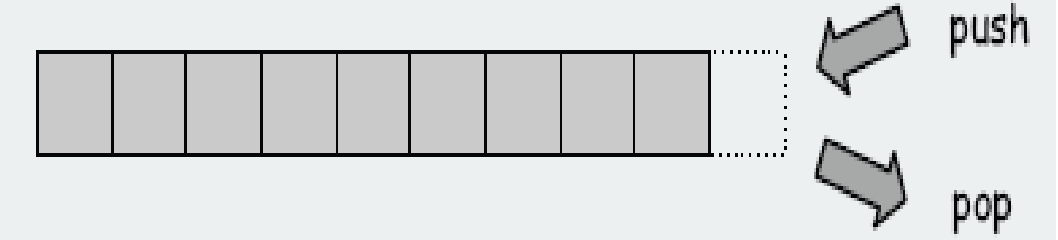
Stacks

N D Gangadhar MSRSAS

The Stack ADT

- Stack A special kind of Vector where insertions and deletions are allowed at one end only (“top”)





N D Gangadhar MSRSAS

Stack Operations

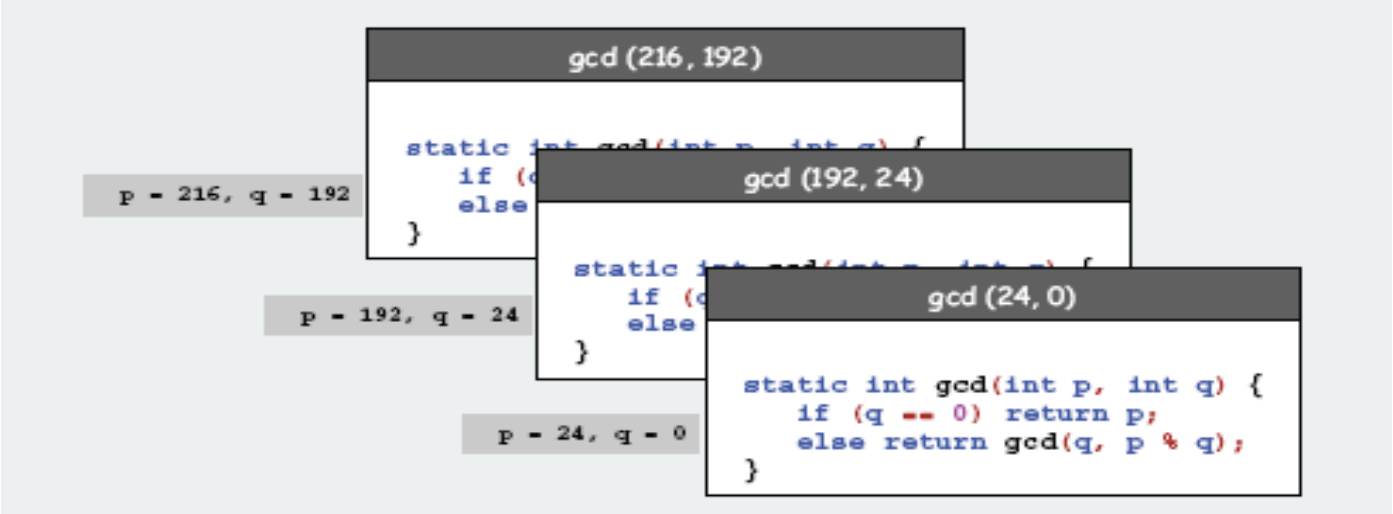
- The main Operations are
  - push(e): Puts element e at the top of the Stack
  - pop: Removes and returns the top-most element from the Stack
  - peek: Returns (with out removing) the top-most element
  - It is important to consider the cases full Stack and empty Stack

- Other operations that are common with Vector
  - size, isEmpty
  - None of the other Vector operations (like insertAtRank) make sense—no rank-ing
  - (Look ahead: look at PriorityQueue and Deque)

- In Java, java.util.Stack extends java.util.Vector

System Applications of Stack

- Stack is one of most used data strucuture for system programming
  - Compilers use it all the time to impelement function calls
  - Function call: push local environment and return address
  - Return: pop return address and local environment
- Recursive function call:



  - One can always use an explicit stack to unravel a recursion

Infix Arithmetic Expression Evaluation – 1

- Dijkstra's two-Stack algorithm
  - Infix notation:

(1 + (2+3) * (4 * 5))

- The algorithm
  - Operand: push onto to the ValueStack
  - Operator: push onto to the OperatorStack
  - Left parenthesis: Ignored
  - Right parenthesis:
    - pop operator from OperatorStack and two values from ValueStack
    - Apply the operator on the operands
    - push the result onto the ValueStack

Infix Arithmetic Expression Evaluation – 2

Sedgewick's Java Implementation of Dijkstra's Algorithm

```
public class Evaluate {
   public static void main(String[] args) {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")")) {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Postfix Expression Evaluation

- A remarkable fact:

  The two-Stack algorithm evaluates to the same values if the operators occur after the two operand values

  (1(23+)*(45*)+)

  Postfix or “Reverse Polish” notation

- Then no need of any paranthesis!

  123+45**+

- Needs only *one* stack!

- Lead to Stack based computer languages
  - C, C++, Java, Python, etc.

N D Gangadhar MSRSAS

Array Based Stack Implementation – 1

- The implementation of Stack based on Array
  - Actually extend Vector ADT
  - The additional features are pop and push
  - Restrict the behaviour of insertAtRank and deleteAtRank
  - ... Override with appropriate calls to pop and push
  - For prototypes, see next slide
- Dynamic resizing strategy
  - One devised for Vector holds good

N D Gangadhar MSRSAS

Array Based Stack Implementation – 2

```
function size:
    return 1+t
end function

function push(e):
    if t = N - 1 then
        StackFullException
    else
        t ← t + 1
        S[t] = e
    end if
end function
```

```
function pop:
    if isEmpty() then
        StackEmptyException
    else
        o ← S[t]
        t ← t – 1
        return o
    end if
end function
```

N D Gangadhar MSRSAS

Stack Deprecation

- Stack is deprecated
- Stack is a frequently used and standard data structure
- It is now deprecated in favour of Deque
- A more consistent and complete behaviour is provided by Deque
- Thus, Stack is now to be realised as a special case of Deque

- Java's suggests Stack use be inherited from Deque
  - Deque<Integer> stack = new ArrayDeque<Integer >( ) ;

Queues

N D Gangadhar MSRSAS

Queues

- What is a Queue?
  - A queue stores elements and serves them
  - Simulates the real life queues
  - A waiting room attached to a server

- A real life example: Queue formed at a bank ATM
  - There is waiting room (outside the booth!)
  - There is a server (the ATM proper)
  - Each of the customers enters the booth for service
  - The server serves the customer
  - Waiting customers enter the service in the order of their arrival
  - First-In-First-Out (FIFO) queue

Queues in Computer Systems

- Queueing and Queue data structures are extensively used in variousaspects of computing
  - Programming: FIFO's and Priority Queues for IPC
  - Accessing peripherals: Requests are queued
  - Kernel scheduling: CPU and other resources are scheduled by the kernel and made available to processes
  - ... A highly evolved priority queue
  - Simulation of Discrete Event Systems, e.g., the ATM

- In addition, Queue (just like Vector or Stack) is used
  - For realising algorithms
  - As a component of other complicated data structures

The Queue Class of Data Structures

- Queue is a class of data structures
  - All of them share a common ADT
  - The benefit of defining an ADT
- Some of the members of this class
  - FIFO Queue (generally refered to as simply Queue)
  - PriorityQueue: Actually a class by itself
  - Preemptive PriorityQueue
  - Non-preemntive PriorityQueue
  - Last-In-First-Out (LIFO) Queue
  - .. A Stack is a LIFO Queue!
  - Randomised service Queue
- Other and more elaborate classes of Queue's
  - Queue with multiple waiting rooms
  - Multiple Server Queue's

N D Gangadhar MSRSAS

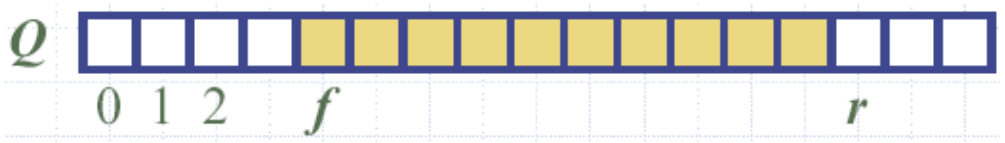
The Queue ADT

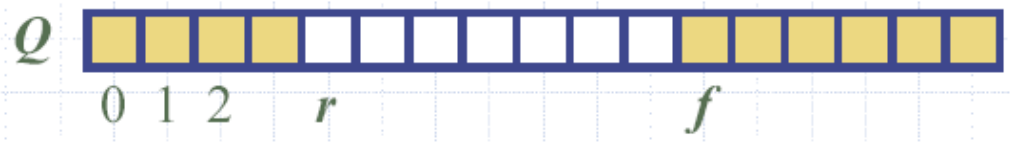
- A Queue ADT extends the Stack ADT (and hence Vector ADT)
  - Allows insertion at any location according to a pre-specified rule—enqueuing rule
  - Items are removed from any location according to a pre-specified rule—scheduling rule
  - Both these rules together form the Queue discipline/rule
  - ... E.g., FIFO, LIFO, Pre-emptive Priority

- Queue Operations
  - size: inherited from Stack and Vector
  - enqueue(e): insert the element e into the Queue according to the enqueuing rule
  - dequeue: remove and return the element being served by the Queue according to the dequeuing rule
  - front: returns (without removing) the Head of the Line (HOL) element (one that is being served)

Array Based Implementation of Queue

- We will restrict our explaination to a FIFO Queue
  - Envision corresponding Array implementation of LIFO
  - Compare it with the Array implementation of Stack!
- Two approaches
  - Regular Array implementation
  - Circular Array implementation





Array Based Realisation of Queue ADT Operations

```
function size:
    return (N - f  + r) mod N
end function

function enqueue(e):
    if size = N - 1 then
        FullQueueException
    else
        Q[r] ← e
        r ← (r+1) mod N
    end if
end function
```

```
function isEmpty:
    return (f = r)
end function

function dequeue:
    if isEmpty then
        EmptyQueueException
    else
        o ← Q[f ]
        f  ← (f +1) mod N
        return o
    end if
end function
```

Lists

N D Gangadhar MSRSAS

The List/Sequence ADT

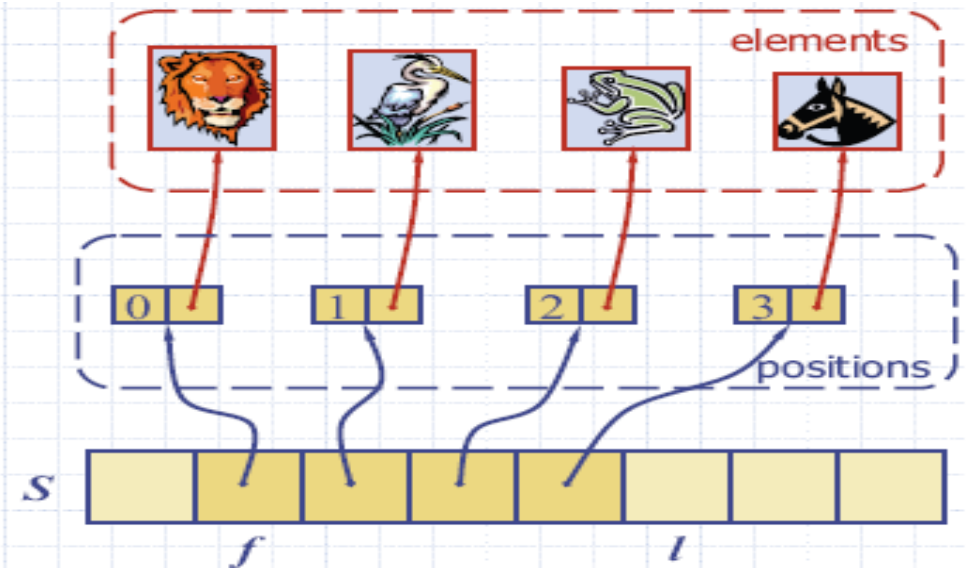
- List data structure is a collection of elements each of which is in turn defined via a Position ADT
- A List is also known as a Sequence
- Position ADT
  - Position ADT abstracts the idea of a place in a data structure
  - Unifies an Array cell or a node in a Linked List
  - The lone Operation
  - element: Returns the element stored
- The List ADT replaces the rank based ordering of the Vector ADT
  - Uses Position based addressing instead
  - And establishes a before-after relationship among the Positions that form the List

List Operations

- The List Operations are of three types
- General Operations
  - size and isEmpty
- Accessor Operations
  - frist and last (Positions)
  - next(p) and previous(p) Position of p
- Update Operations
  - replace(p,e): Replaces the current value at p by e
  - remove(p): Removes Position p
  - insertBefore(p) and insertAfter(p)
  - insertFirst and insertLast: Re-definition needed for changing First and Last

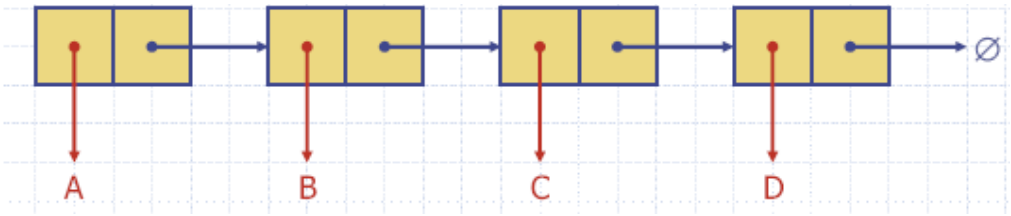
Array Based List Implementation

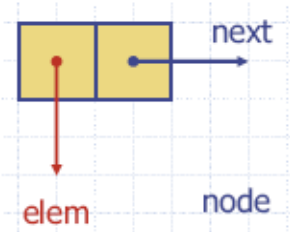
- The Array based implementation of the List ADT
  - Recollect the Circular Array based Vector implementation
  - The Circular Array can be used to store Position information
  - Each Position itself stores the actual element
- This form of implementation is known as Node List
  - As opposed to Array List which is the same as a Vector with different interface and Operations



Singly Linked List

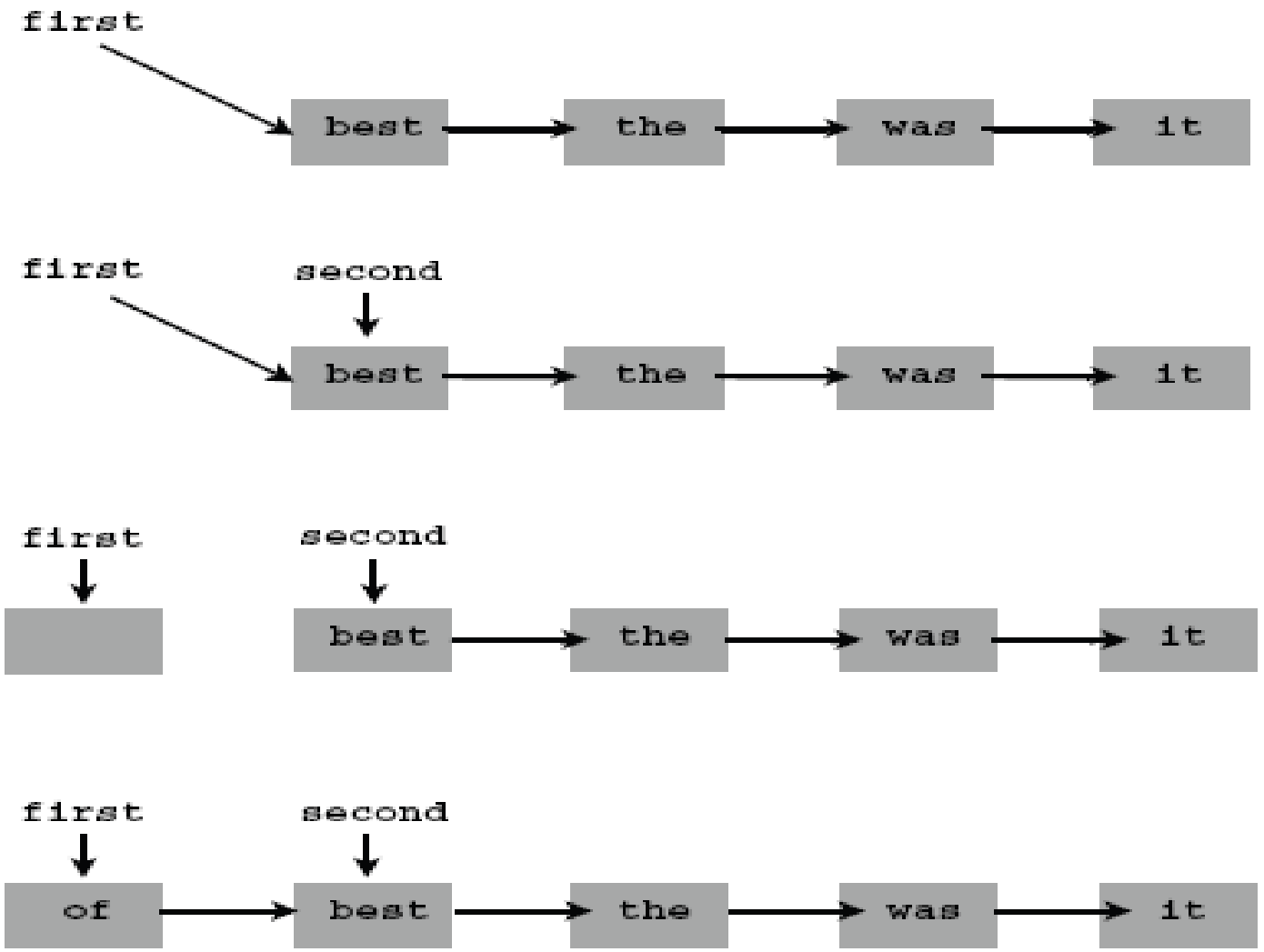
- A Singly Linked List is the simplest realisation of the List ADT





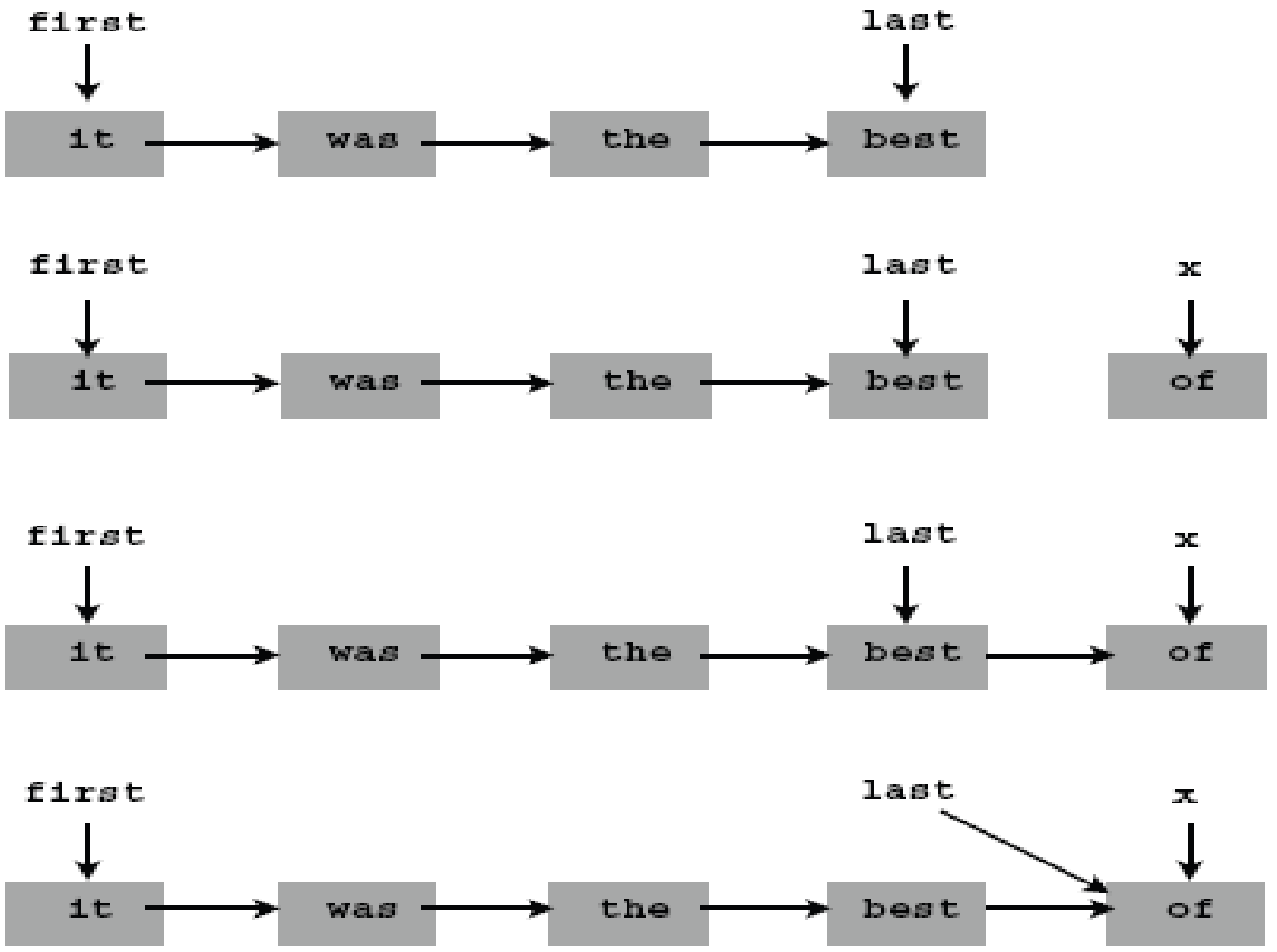
- Because of the Position based addressing, all the List Operations in a Linked List run in constant O(1) time

List Based Stack Implementation

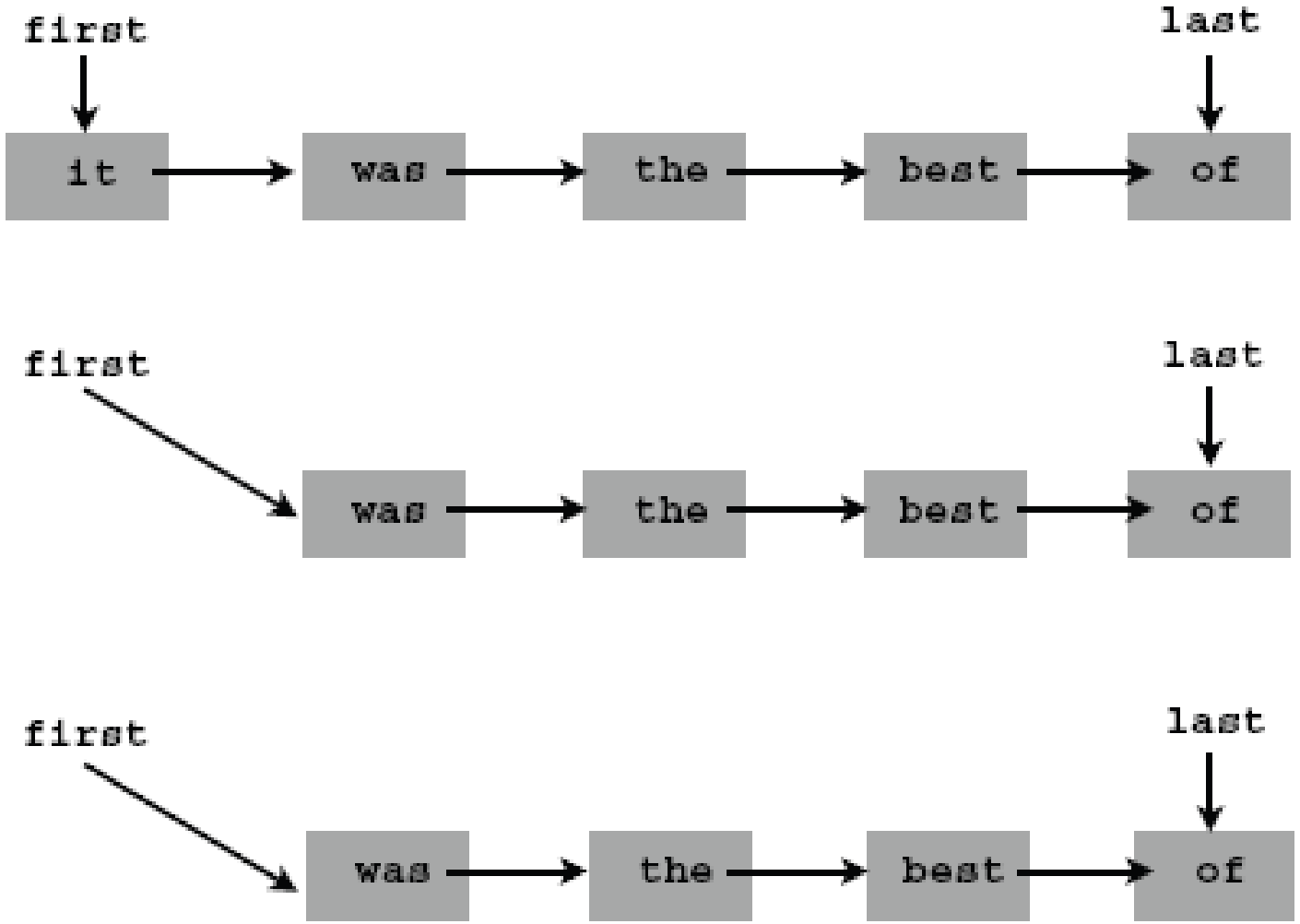


N D Gangadhar MSRSAS

Linked List Impelementation of Queue – 1

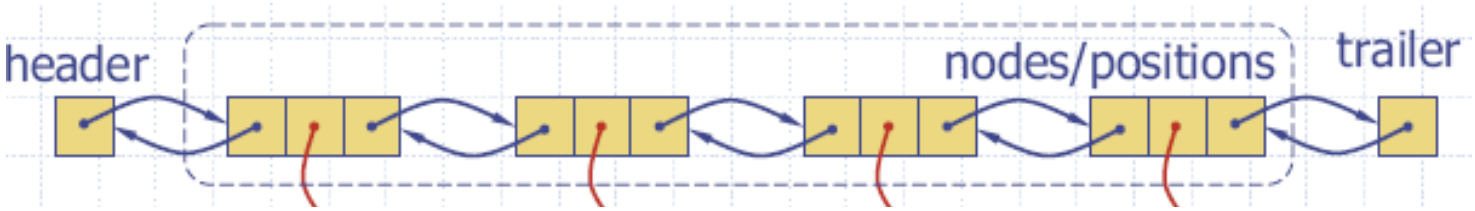


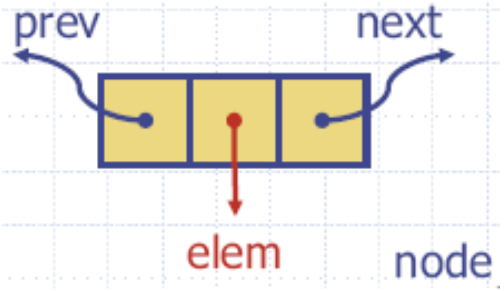
Linked List Implementation of Queue – 2



Doubly Linked List

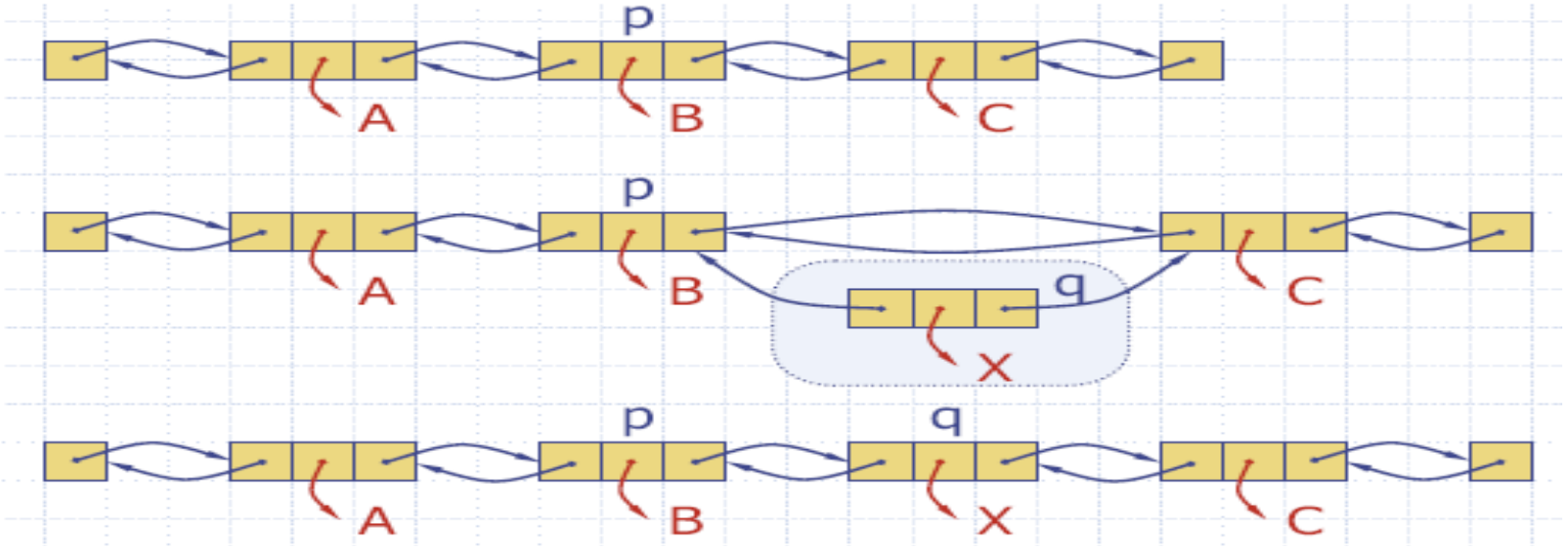
- Doubly Linked List provides a better implementation of the Link ADT (than Singly Linked List)





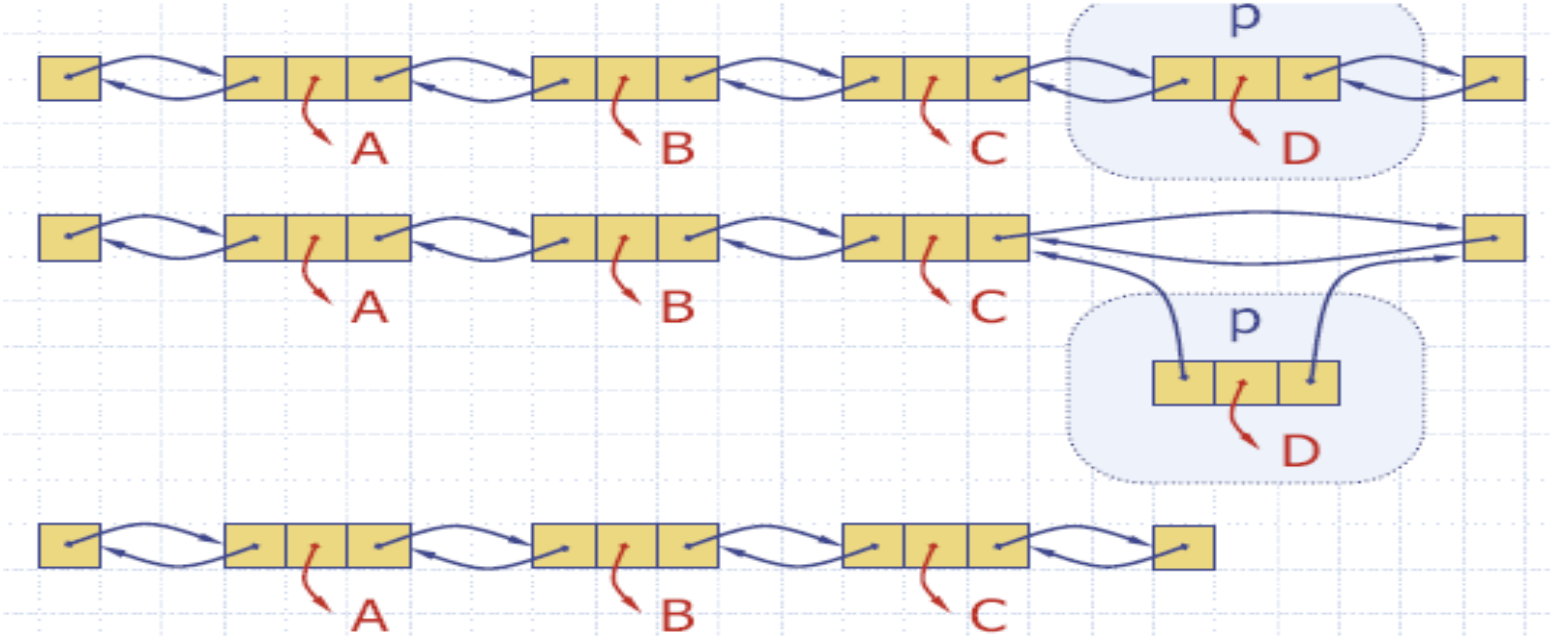
N D Gangadhar MSRSAS

Doubly Linked List insertAfter




```
function insertAfter(p;e):
        q ← new node
        q.element ← e
        q.previous ← p
        q.next ← p.next
        (p.next).previous ← q
        p.next ← q
end function
```

Doubly Linked List remove




```
function remove(p):
    t ← p.element
    (p.previous).next ← p.next
    (p.next).previous ← p.previous
    p.previous  ← NULL
    p.next  ← NULL
    return t
end function
```

Trees

N D Gangadhar MSRSAS

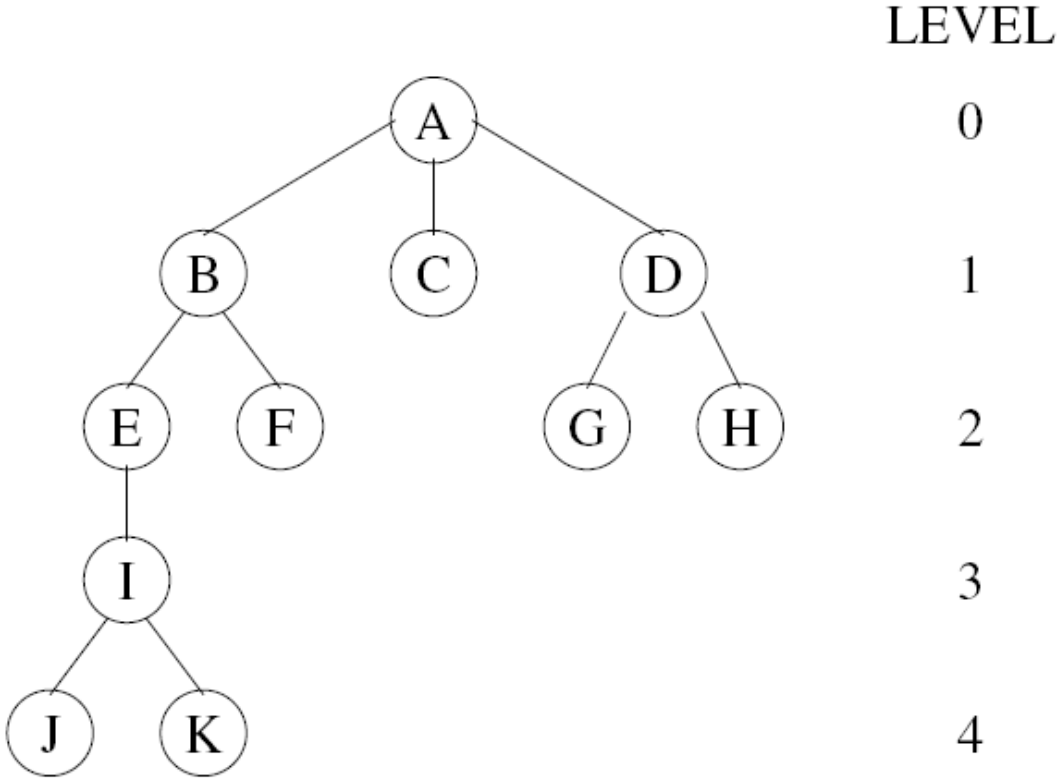
Tree Data Structures

Trees in Computer Science terminology refer to hierarchical structured data representations

The actual visual representation of a Tree has *root* at the top and branches grown *down* ending in *leaves*

- Trees consist of *nodes* with a parent-child relation

- They provide a structure that can be applied in a wide variety of applications



Terminology

Root: A node without a parent

Internal Node: A node with at least one child

Leaf (or **External Node):** A node without a child

Ancestor of a node: Parent, grand-parent, grand-grand-parent, *etc.*

Descendent of a node: Child, grand-child, great-grand-child, *etc.*

Depth of a node: Number of ancestors of the node

Height of the Tree: Maximum depth of any node in the Tree

Tree ADT

We will follow the Position abstraction (not standard but useful) to abstractly represent a node

Generic Methods:

size, isEmpty, elements, positions

Accessor Methods

root, parent, children

Query Methods

isInternal, isLeaf/isExternal, isRoot

Update Method:

replace (n, m)

Additional methods for specialised Trees

Tree Traversal

The fundamental *algorithm* on Trees is Traversal

- Visit each node of the Tree
- Optionally perform some operation during each visit

Types of Tree traversals

- Preorder traversal
- Postorder traversal

Tree traversal is recursive in nature; the traversal algorithms are expressed recursively

Preorder Traversal

In a preorder traversal, each node is visited before all of its descendents

Algorithm ***preOrder*** (v):

visit (v)
// Do something at v
for each w ← *children* (v)
  preOrder (w)
end for

Useful for prefix expression evaluation and processing structured documents (e.g., XML)

Postorder Traversal

In postorder traversal of a Tree, each node is visited *after* its descendents

Algorithm postOrder (v):

```
for each w ← child (v)
  postOrder (w);
end for
visit (v);
// Do something at v
```

Useful for postfix expression evaluation and accumulation of values at ancestor nodes (*e.g.*, hierarchical disk usage information)

Binary Trees

N D Gangadhar MSRSAS

Binary Trees

A very import class of Trees

Each node has at most two children

- The children of each node are *ordered (*or, *ranked)*
- The order is used to place the children on the *left* and *right* branches of the node
- Left child and right child

Alternative *recursive definition*

- A Binary Tree is either a Tree with a single node
- Or, a Tree whose root has an ordered pair of children, each of which is a Binary Tree

Applications

- Binary (*e.g.*, arithmetic) expression evaluation
- Binary search algorithms
- Decision logic implementation

Binary Tree Properties

n: Number of nodes

e: number of external nodes

i: number of internal nodes

h: height of the tree

Properties

h + 1 <= e <= 2^h

2h + 1 <= n <= 2^(h+1) - 1

log (n + 1) – 1 <= h <= (n-1)/2

Binary Tree ADT

The Binary Tree ADT extends the Tree ADT

Additional methods in Binary Tree ADT

left (p)

right (p)

hasLeft (p)

hasRight (p)

Additional methods are defined by data structures implementing the ADT

E.g., height minimising balancing

Inorder Traversal

Binary Tree ADT makes it possible to add another traversal method, namely inorder traversal

An internal node is the junction of the left and right descendent subtrees

In an inorder traversal, the left subtree is visited before the node itself is visited followed by the right subtree traversal

Algorithm ***inOrder*** (v)

```
if hasLeft (v)
  inOrder (left (v))
visit (v)  // And do something at v
if hasRight (v)
  inOrder (right (v))
```

N D Gangadhar MSRSAS

Array Based Tree Implementation

Use a Vector data structure to store the Tree nodes

Rank of a node is defined by

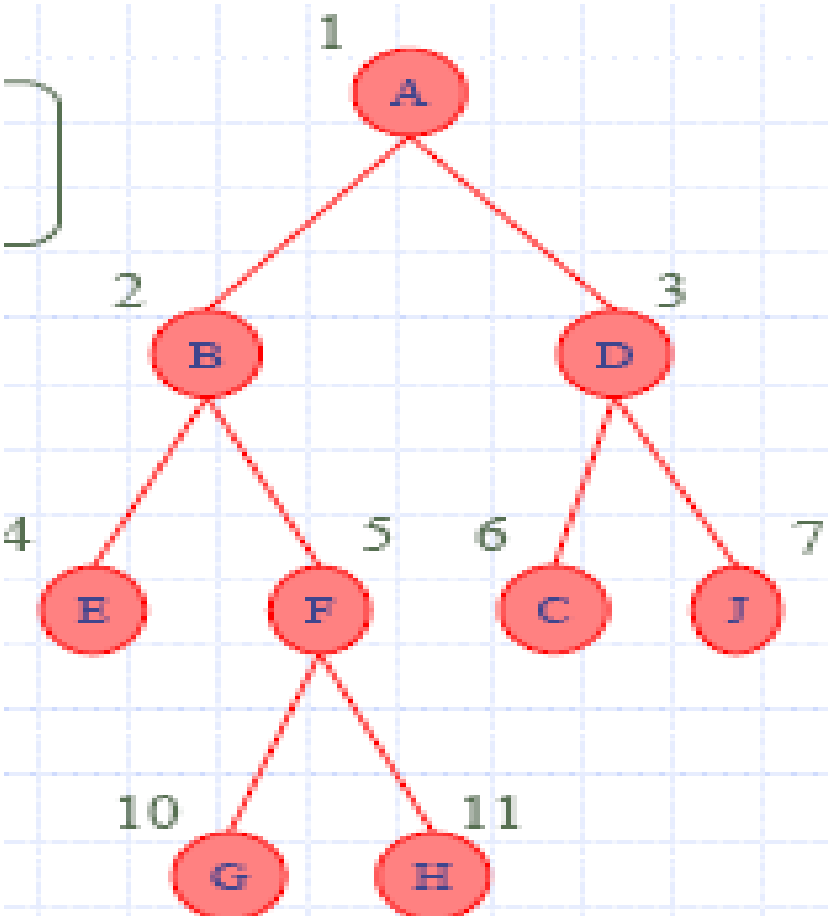
rank (root) = 1

If node is left child

rank (node) = 2 *rank* (*parent* (node))

If node is right child

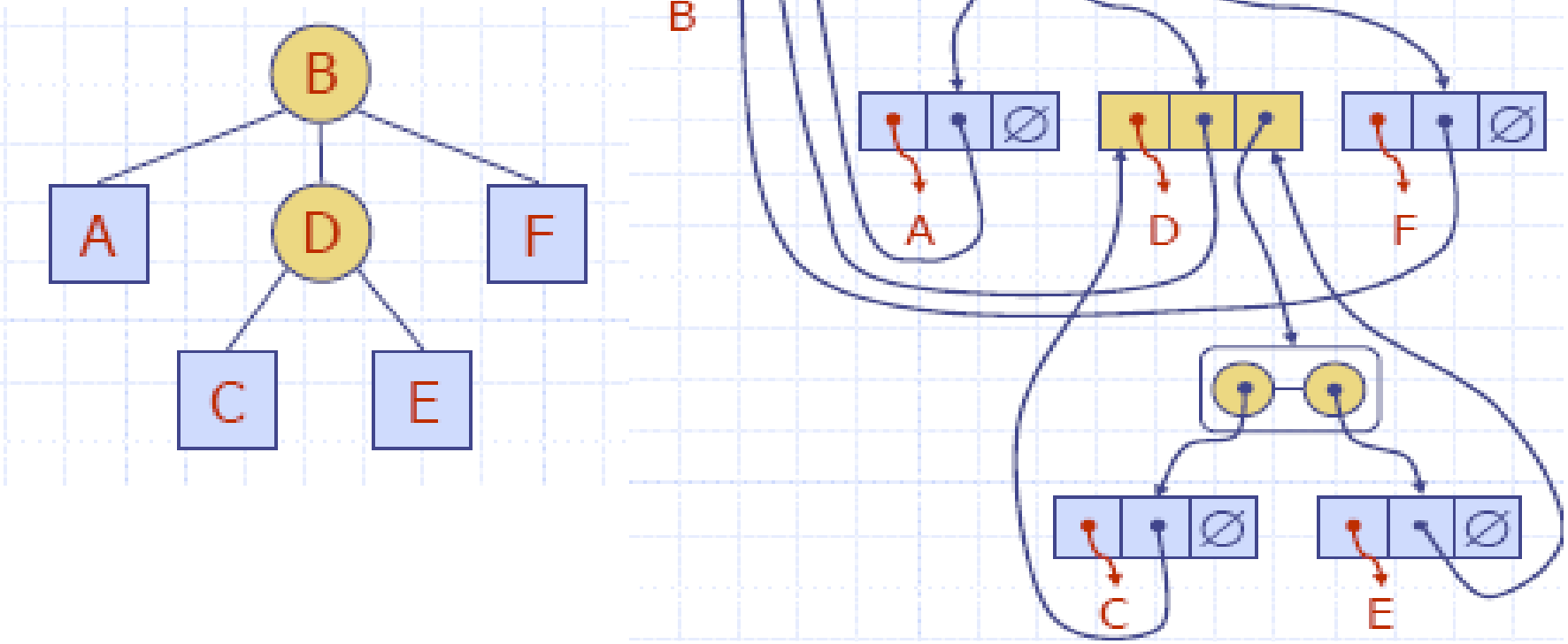
rank (node) = 2 *rank* (*parent* (node)) + 1



List Based Tree Implementation

Each Tree node is stored in an List object storing

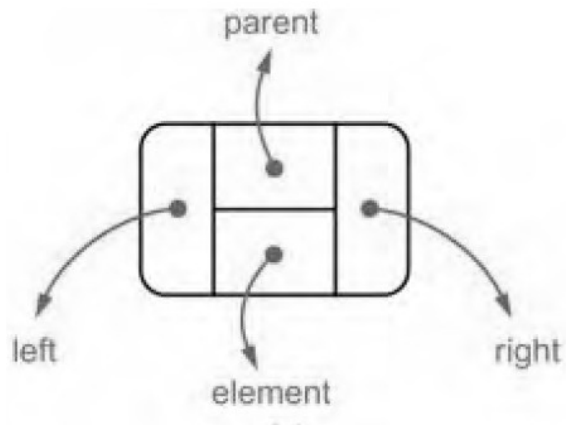
node element, parent node, sequence of children nodes

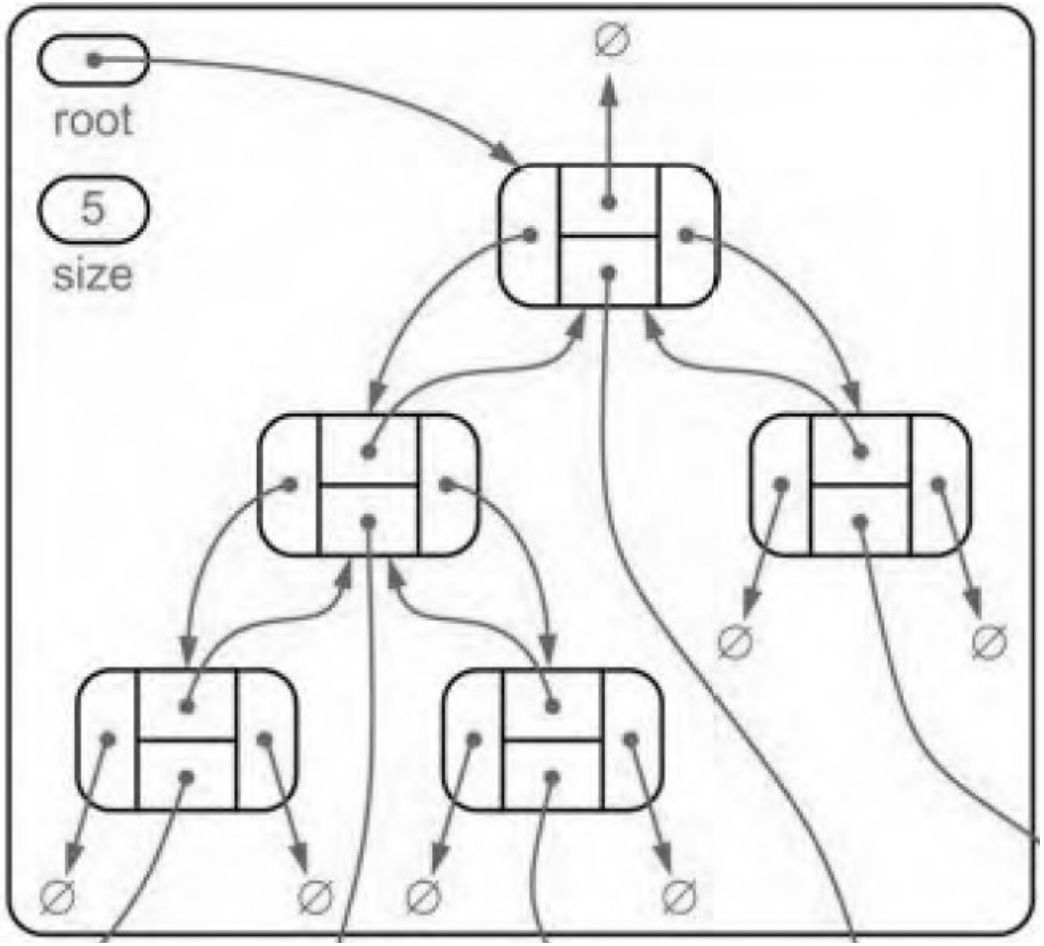


List Implementation of Binary Tree

The Binary Tree node now has four components

element, parent, left and right





Euler Tour Traversal - 1

Euler Tour Traversal unifies and generalises Binary Tree traversal

In inorder, preorder and postorder traversals each node is visited only once

By relaxing this requirements, we can define a general traversal, Euler Tour

Each node is visited three times in an Euler Tour

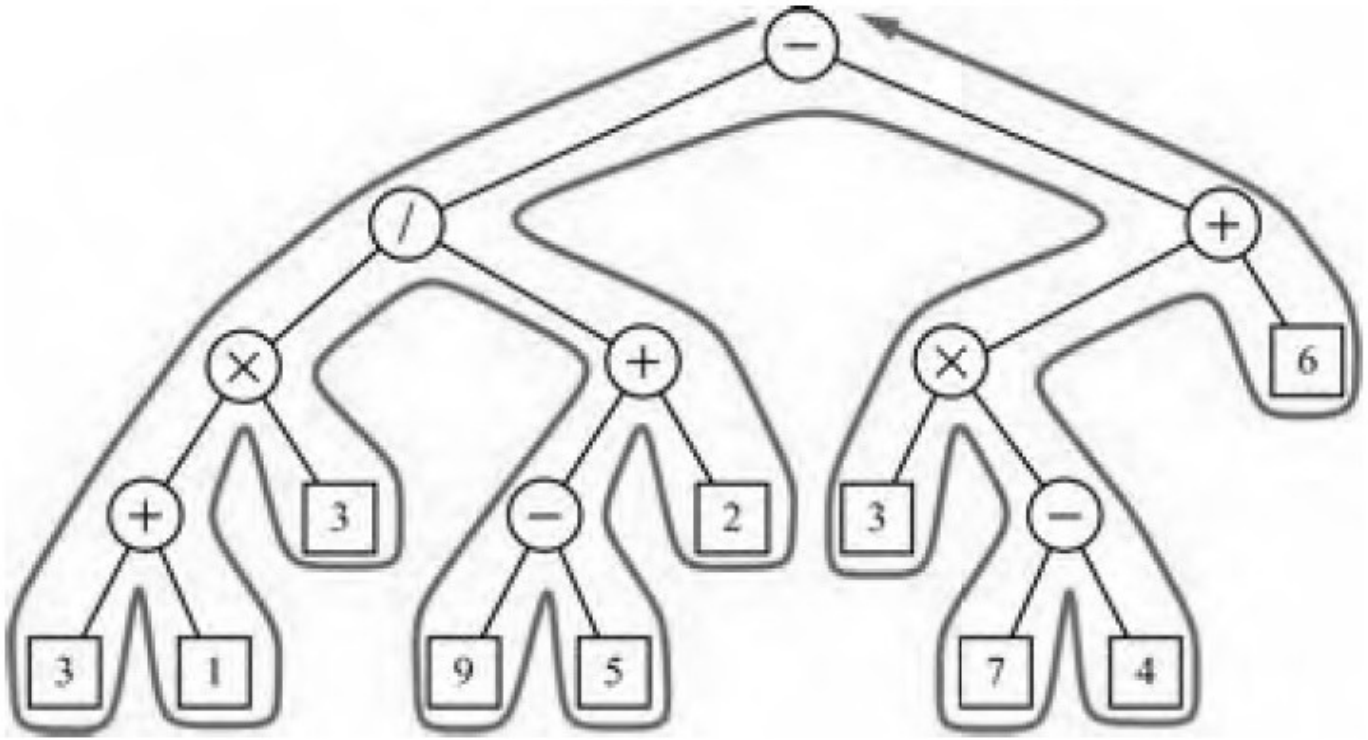
Inorder, preorder and postorder traversals become special cases

Useful in succinctly expressing general kind of algorithms on Trees

Euler Tour Traversal - 2

Each node is visited 3 times

- On the left (preorder)
- From below (inorder)
- On the right (postorder)



Summary

- ADT is an abstract model of a data structure along with its operations
- ADT makes it possible to express algorithms independent of the underlying implementation of the data structure
- The complexity of ADT operations give an *upper* bound on the complexity of actual implementation of the data structure
- Array and List data structures are fundamental building blocks of other data structures
- Stack and Queue ADTs are data structures in which there is restriction on the insertion and deletion of elements
  - In a Stack, elements can be inserted and removed only at one end
  - In a Queue, elements can be inserted at one end and can be removed only from the other end

Summary

- Trees are two dimensional data structures incorporating parent-child relationship
- The additional spatial structure allows design new and/or improved algorithms
- Binary Trees are among the most used data structures
- Arithmetic expressions (and polynomials) are best represented and evaluated using Binary Trees
- Efficient Searching and Sorting algorithms can be designed and analysed using Binary Trees
- The height of a Tree is crucial for efficient implementation of algorithms using it