DATA STRUCTURES AND ALGORITHMS

SLIDE 3

Linked lists

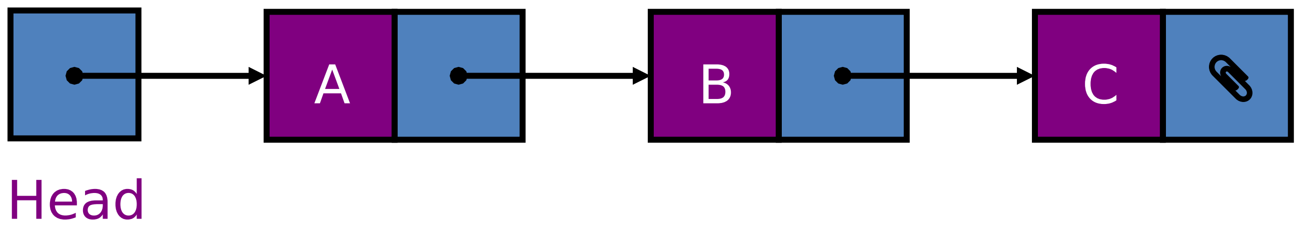
- A linked list is a data structure specially designed to overcome the limitations of a linked list
- The linked list is a very flexible **dynamic data structure**: items may be added to it or deleted from it at will.
- An array (linear list) allocates memory for all its elements lumped together as one block of memory.
- In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or

Linked lists

- The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain.

- Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field, which is a pointer used to link one node to the next node.

Linked lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`

Linked Lists: the List Node

The list node is a simple self-referential structure that stores an item of data, and a reference to the next item.

```
class ListNode
{
   int data;

   ListNode next;

   public ListNode(int data)
   {
      this.data = data;
      this.next = null;
   }
}
```

The data variable is where the information to be stored resides. It may be of any primitive or reference type appropriate for the data.

The next variable is the self-referential link to the next data item.

The constructor initialises the node object by storing the data that was given as an argument, and sets the next reference to **null**.

Linked Lists: the Header Class

The header class is the public interface for the linked list. It is where the functionality is stored (as methods), and contains a link to the first item of the list (the ‘head’.)

```
class List
{
   ListNode head;

   List()
   {
      head = null;
   }

   add();
   find();
   delete();
}
```

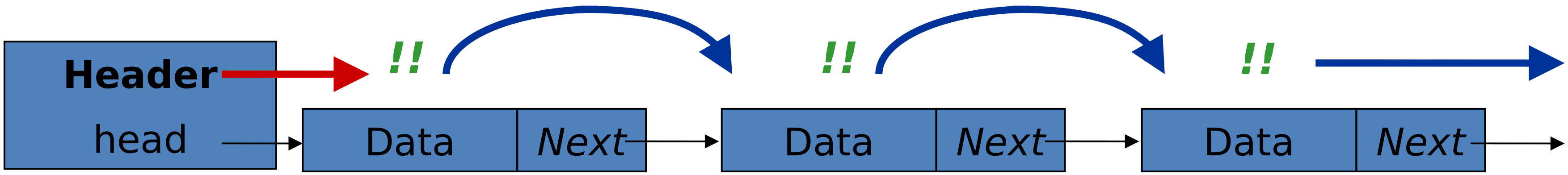
The head variable is a reference to the first item in the list.

The constructor initialises the list by setting the head to **`null`** (an empty list.)

The methods provide a way to use the list. They each access the structure through the head reference.

Linked Lists: List Traversal (1)

It is sometimes necessary to traverse the entire length of the list to perform some function (for example, to count the number of items, or display summary information.)



Step 1: Step through the list from the header node forward.

Step 2: Perform the desired operation at that node.

Step 3: Move onto the next node, until the end of the list is reached.

List traversal forms the basis of many of the list manipulation operations such as add, retrieve and delete.

Linked Lists: List Traversal (2)

The code below will traverse the entire list, and print out the data contained in each node.

```
public void traverse()
{
    ListNode current = head;

    while (current != null)
    {
        System.out.println(current.data);

        current = current.next;
    }
}
```

Step 1: Maintain a variable to store the current position in the list.

Step 2: Continue stepping through the list, until the end of the list (a **`null`** reference) is reached.

Step 3: At each step, print out the data present in the current node.

Because of the way a linked list is defined, we can only access data in one direction, and sequentially (one item after another.)

Linked Lists: Adding Data

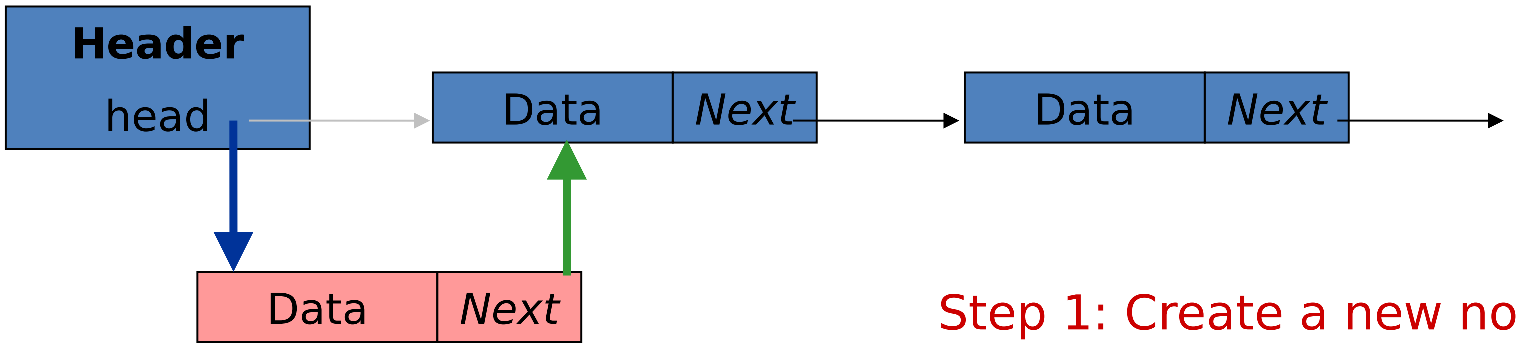
Data is added to a linked list by wrapping the data to add into a node, and then placing that node at the appropriate place in the data structure.

Depending on the circumstances and purpose of the list, there are a number of places where data may be added:

- **At the start (head) of a list**
- **In the middle of the list**
- **At the end (tail) of the list**
- **At the appropriate place to preserve sort order**

Linked Lists: Adding Data to the Head (1)

Adding data to the head of a list is the easiest and quickest way in which it can be done.



Step 1: Create a new node to store the given data.

Step 2: Set the new node's next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

The order in which the link manipulations are done are very important; they must always be done from right to left, otherwise data nodes will be lost.

Linked Lists: Adding Data to the Head (2)

The algorithm is simple to translate into source code, as each step corresponds with just one simple instruction.

```
public void addToHead(int data)
{
   ListNode newNode = new ListNode(data);

   newNode.next = head;

   head = newNode;
}
```

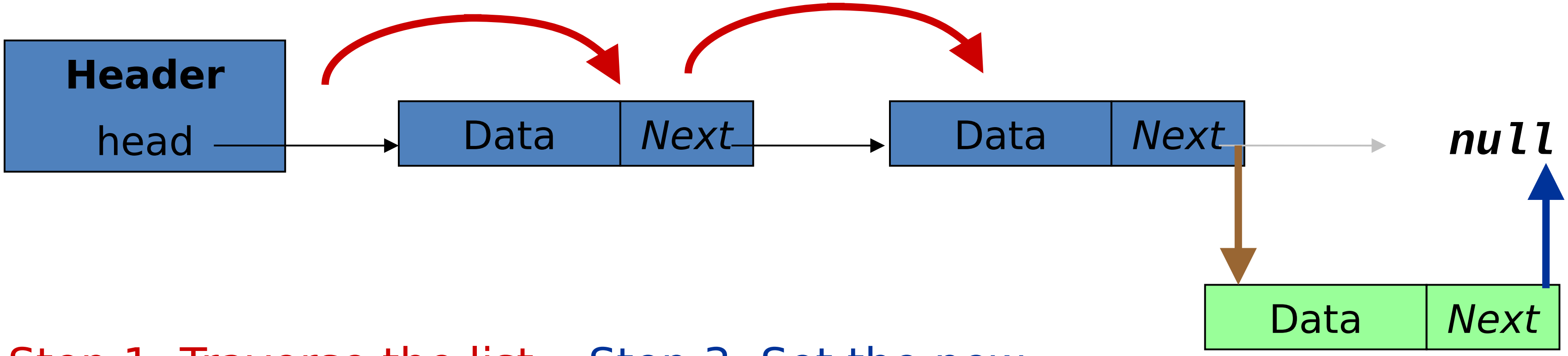
Step 1: Create a new node to store the given data.

Step 2: Set the new node's next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (1)

Adding data to the middle or tail of the list is essentially the same process. The diagram below illustrates adding to the end (tail.)



Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (2)

```
public void addToTail(int data)
{
    ListNode insert = head;

    while (insert.next != null)
        insert = insert.next;

    ListNode newNode = new ListNode(data);

    newNode.next = insert.next;

    insert.next = newNode;
}
```

Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

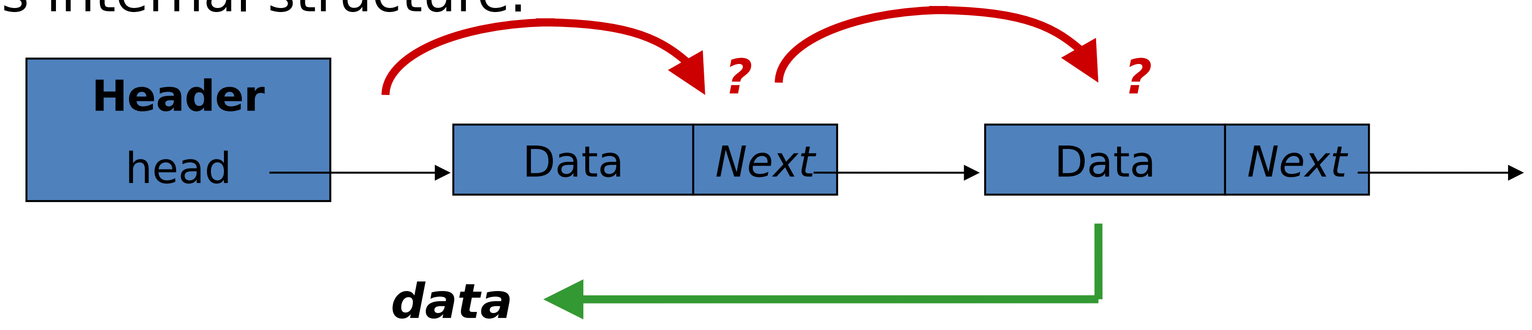
Step 4: Reset the insertion point's next reference to point to the newly created node.

Tip: the traversal in step 1 could have been avoided by maintaining a tail pointer in the header class (as well as a head.)

Linked Lists: Retrieving Data (1)

Data retrieval consists of traversing the data structure until a matching node is found. The data portion of the node is then returned (if the data is not found, some form of failure signal should be returned instead.)

The whole data node should not be returned, as it is part of the list's internal structure.



Step 1: Traverse the list.

Step 2: While traversing, compare the search 'key' value with the data in each node.

If the data keys match, return the data portion of the node.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Retrieving Data (2)

```
public int retrieve(int key)
{
    ListNode current = head;

    while (current != null)
    {
        if (current.data == key)
            return data;

        current = current.next;
    }

    return -1;
}
```

Step 1: Traverse the list.

Step 2: While traversing, compare the search ‘key’ value with the data in each node.

If the data keys match, return the data portion of the node.

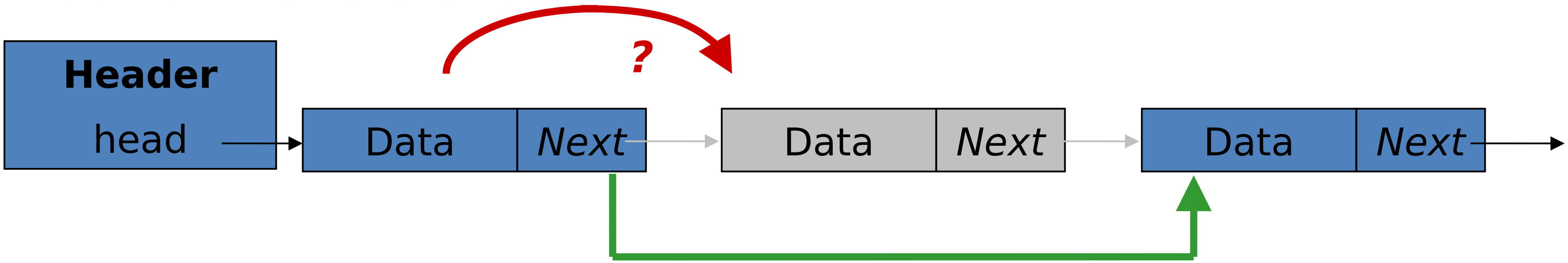
Step 3: If the list is exhausted without the data being found, return a search failure signal.

Note: this program assumes that only positive integers are being stored. This way, the calling program can easily assume that a non-positive answer (e.g. -1) is the signal for a failed retrieval attempt.

Linked Lists: Deleting Data (1)

Deletion is very similar to retrieval. As before, the list is traversed to find data matching a given ‘key’ value. However, instead of returning the data, the node is to be deleted.

The node can be deleted by having the next references ‘jump over’ the node to delete. To do this, the node before the one to delete must be known, and as such, the traversal needs to keep track of two references.



Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, ‘jump over’ the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Deleting Data (2)

```
public boolean delete (int key)
{
   ListNode current = head;
   ListNode previous = null;

   while (current != null)
   {
      if (current.data == key)
      {
         previous.next = current.next;
         return true;
      }

      previous = current;
      current = current.next;
   }

   return false;
}
```

Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, ‘jump over’ the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Note: a **`boolean`** variable is used in this code to return success (**`true`**) and failure (**`false`**).

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- In Linked Lists we don’t need to know the size in advance.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

- **Applications of Linked Lists**
  - Linked lists are used to implement stacks, queues, graphs, etc.

EXERCISE

- DISCUSS THE VARIUOS TYPES OF LINKED LISTS GIVING THEIR ADVANTAGES AND DISADVANTAGES
  5 MARKS
- EMAIL: course101.work@gmail.com

To be submitted by Friday 4/06/2021

Introduction to Stacks

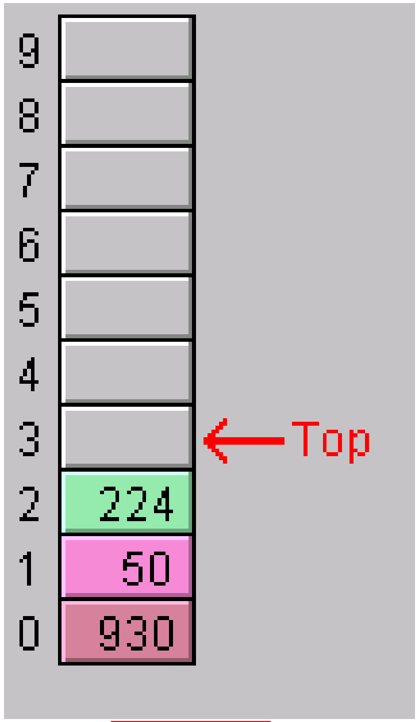
- What is a Stack?
- Stack implementation using array.
- Stack implementation using linked list.
- Applications of Stacks.

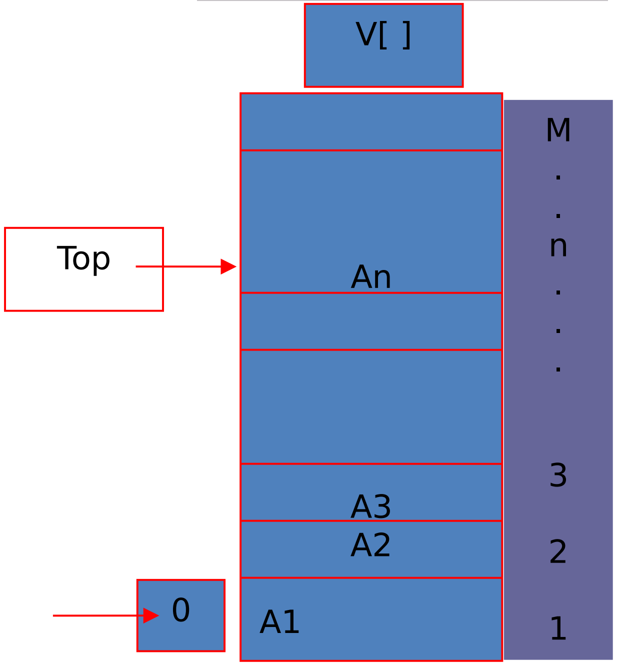
Stack ADT

- Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order.

Stacks: *Definition:*

- Is a Linear list in which all insertions and deletions are made at one end called the Top. E.g. in Linear List <a_1 a_2 ...a_n > deletion and insertion can only be on element an (Top element).
- It is a LIFO (Last In First Out) list
- Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the





Stacks

- The storage is downwards-up so that all operations are on Top element.

- A pointer is restricted to the Top element.

- If there is no element in stack (n=0.) The top will be -1

Note:

- Index used from 0 i.e. V[0].....V[n-1].
- [Top = -1 means empty Stack]
- [Top = M-1 means full Stack] for an M-capacity storage Stack.

Stack operations

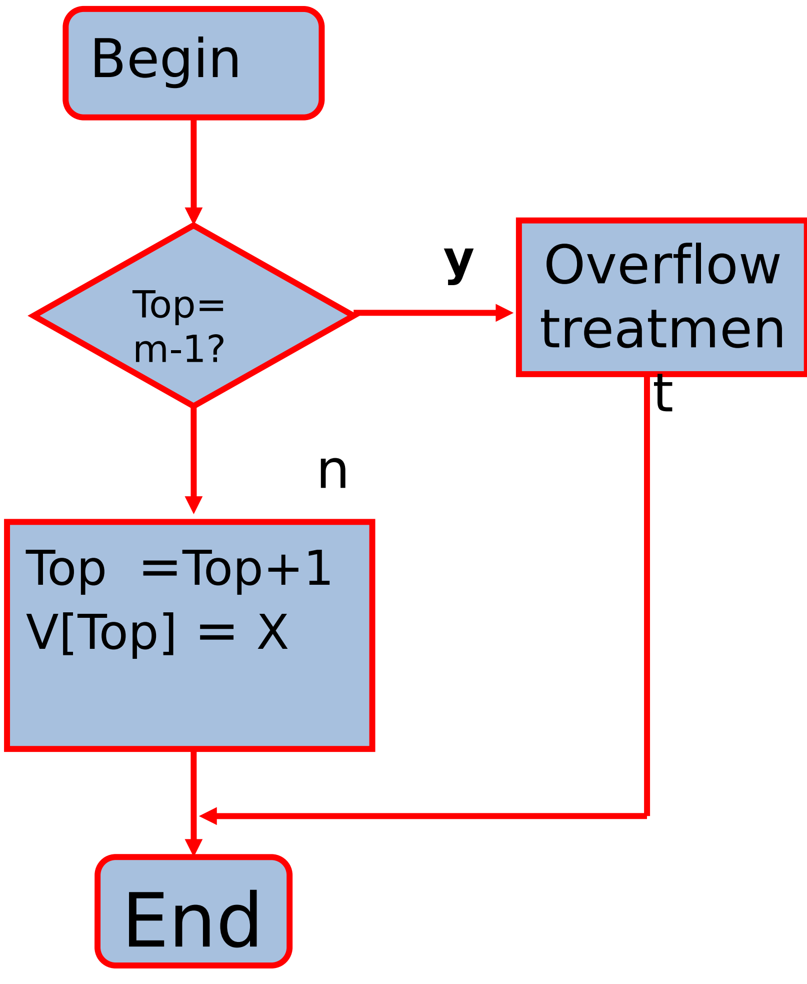
- makeNull(s) – make s be an empty stack
- top(s) – return the element at the top of the stack
- pop(s) – return and delete the element at the top of the stack. The stack size reduces by 1
- push(x,s) – insert element x at the top of the stack. The size of the stack increases by one.
- empty(s) – returns true if the stack has no elements.

Insertion in a stack

- For example inserting new element say X (Fig)

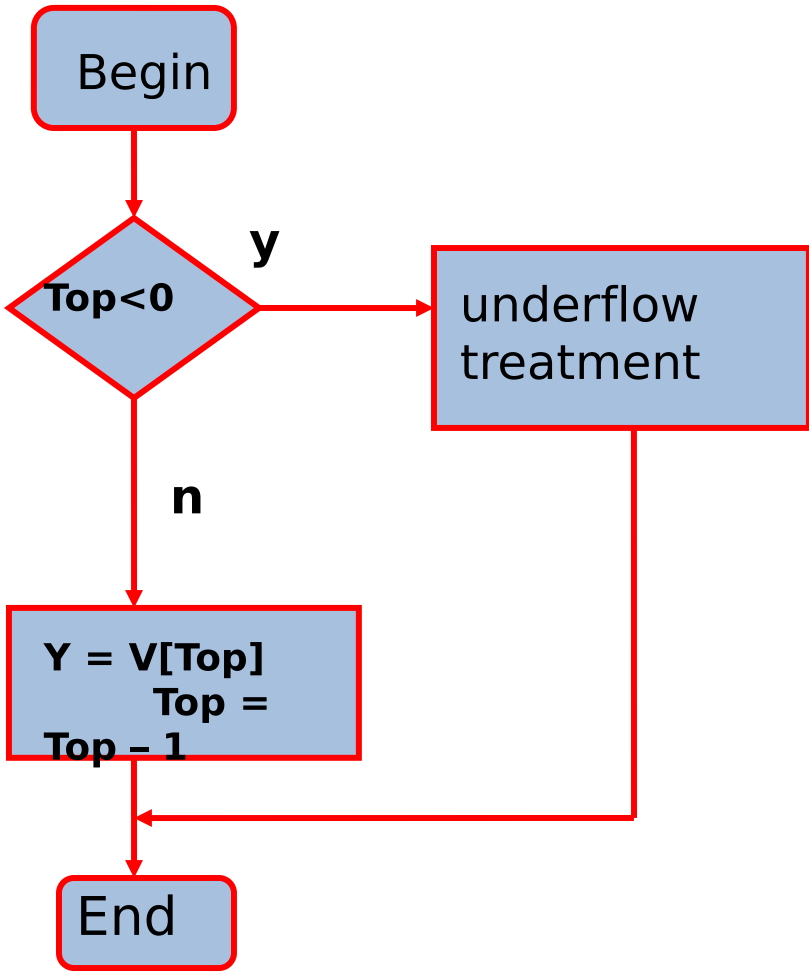
- A test is made of whether Stack is full: If full then abort the procedure or else move pointer Top to position Top+1 then insert element X, i.e. V[Top] = X

- X becomes the new top element:



Stack:Deleting an element (Top element)

- Declare temporary variable Y is to store the deleted element:
- If Stack is empty i.e. Top = -1 then there occurs what is known as an underflow.
- Otherwise delete the top element that is, V[top] , and adjust pointer Top to (Top-1).
- The deleted element Y may be printed out



A Stack ADT Specification (1)

How do we separate implementation from specification?
Java's interface is the solution. The user need see only the method signatures specified in the interface; the user need not be concerned about the implementation.

```
public interface StackMethods
{
   public abstract void push(Object item) throws
                              StackOverflowException;
   public abstract void pop() throws
                              StackUnderflowException;
   public abstract Object topItem() throws
                              StackUnderflowException;
   public abstract boolean isEmpty();
   public abstract boolean isFull();
}
```

A Stack ADT Implementation (1)

With the specification defined, we can write an implementation.

This implementation represents the stack with an array, and the interface methods manipulate the array to simulate the stack operations.

Note that no exceptions are being handled; this is the responsibility of the user of this class.

```
public class Stack implements StackMethods
{
   public Object theStack[];
   private final int MAXSIZE = 100;
   private int nextPos;

   public Stack()
   {
      theStack = new Object[MAXSIZE];
      nextPos = 0;
   }
```

The data is being stored in generic Object data types.

A Stack ADT Implementation (2)

```
public void push(Object item) throws StackOverflowException
{
   if (nextPos < MAXSIZE)
   {
      theStack[nextPos] = item;
      nextPos++;
   }
   else
      throw new StackOverflowException();
}

public void pop() throws StackUnderflowException
{
   if (nextPos > 0)
      nextPos--;
   else
      throw new StackUnderflowException();
}
```

A Stack ADT Implementation (2)

```
public Object topItem() throws StackUnderflowException
{
   if (nextPos > 0)
      return theStack [nextPos - 1];
   else
      throw new StackUnderflowException();
}

public boolean isEmpty()
{
   if (nextPos == 0)
      return true;
   else
      return false;
}
```

```
public boolean isFull()
   {
      if (nextPos == MAXSIZE)
         return true;
      else
         return false;
   }
}
```

Another Stack ADT Implementation (2)

```
public class Stack implements StackMethods
{
    public ListNode theStack;

    public Stack()
    {
        theStack = null;
    }

    public void push(Object item) throws
                                  StackOverflowException
    {
        ListNode temp = theStack;
        theStack = new ListNode (item);
        theStack.next = temp;
    }
```

This method has the throws clause to satisfy the interface, but it will never happen.

Another Stack ADT Implementation (3)

```
public void pop() throws StackUnderflowException
{
   if (theStack == null)
      throw new StackUnderflowException();
   else
      theStack = theStack.next;
}

public Object topItem() throws StackUnderflowException
{
   if (theStack == null)
      throw new StackUnderflowException();
   else
      return theStack.data;
}
```

Stack ADT Implementation

- Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

Applications of Stacks

- Some direct applications:
  - Conversion of tail-recursive algorithms to iterative ones. [Note: Tail recursion will be covered in a later lesson]
  - Keeping track of method calls: Method activation records are saved on the run-time stack
  - Evaluation of arithmetic expressions by compilers [infix to postfix conversion, infix to prefix conversion, evaluation of postfix expressions]

- Some indirect applications
  - Auxiliary data structure for some algorithms
    - Example: Converting a decimal number to another base
  - Component of other data structures
    - Example: In this course we will use a stack to implement a Tree iterator

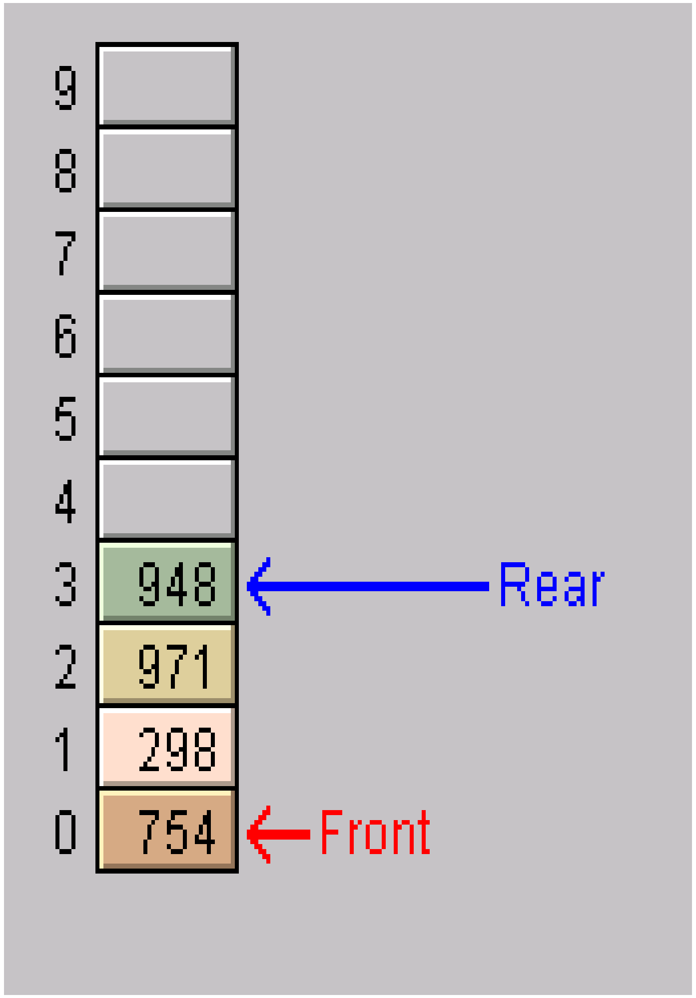
Application of Stacks - Evaluating Postfix Expressions

(5+9)*2+6*5

- An ordinary arithmetical expression like the above is called infix-expression -- binary operators appear in between their operands.
- The order of operations evaluation is determined by the precedence rules and parentheses.
- When an evaluation order is desired that is different from that provided by the precedence, parentheses are used to override precedence rules.

Queues: *Definition:*

- Is a linear list in which all insertions and deletions are restricted:
- Uses FIFO algorithm
- All insertions into the queue take place at one end called the <u>rear</u> while all deletions take place at the other end called the <u>front</u>



Queue: *Operations:*

- **makeNull(q) –** makes a queue empty and returns an empty queue
- **peek(q) –** returns the first element on a queue
- **enqueue(x,q)** – inserts element q at the end of the queue
- **dequeue(q)–** deletes the first element element of the queue
- **Empty(q)** – returns true iff the queue is empty.

Queues Insertion (enqueue);Example

- Insertion of element x

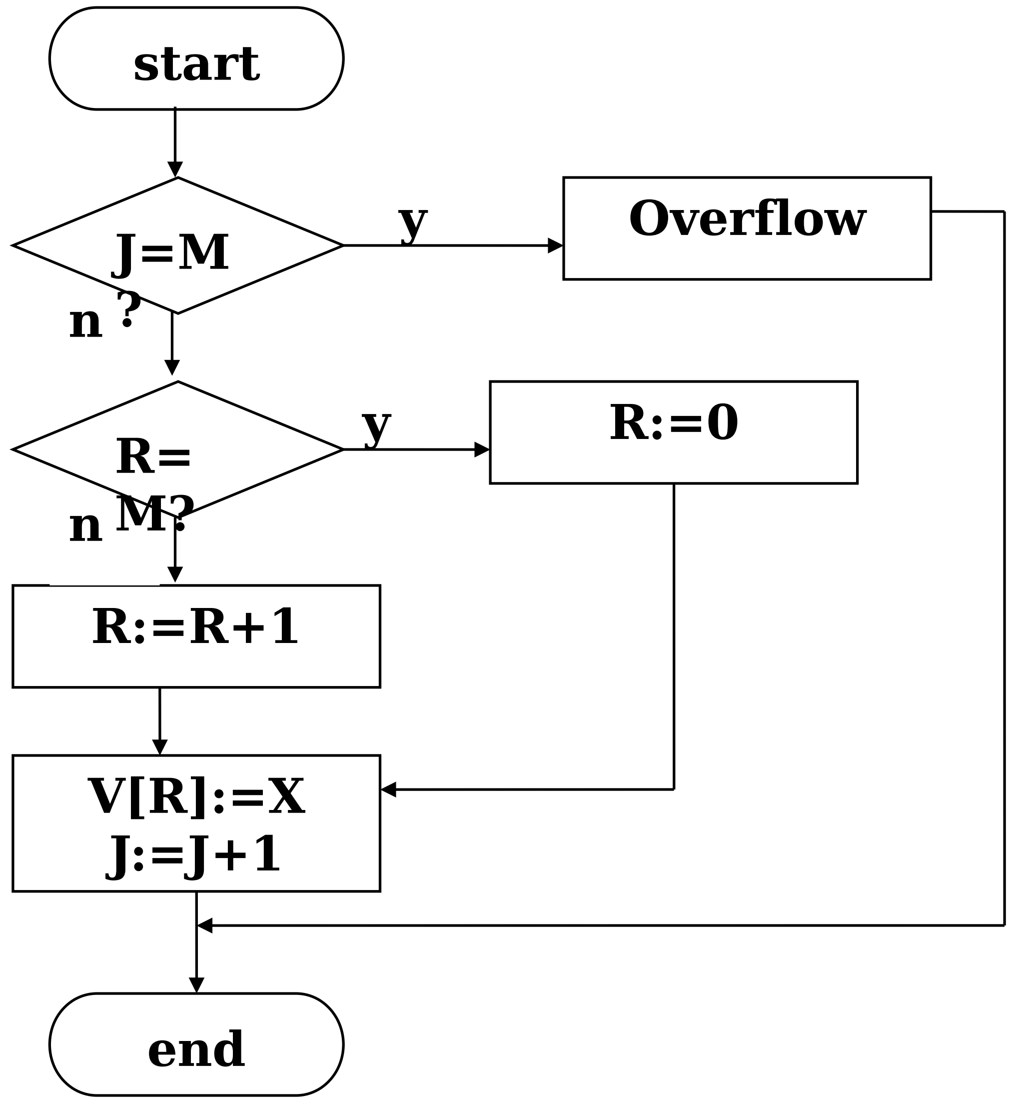
- For this, it's imperative to check whether queue is full or not.

R= Rear points to rear Element

F= Front points to front element
M – size of the array

J – the number of elements currently in the list. its 0 if queue is empty and M if queue is full

Initially, R is -1 and F is 0.



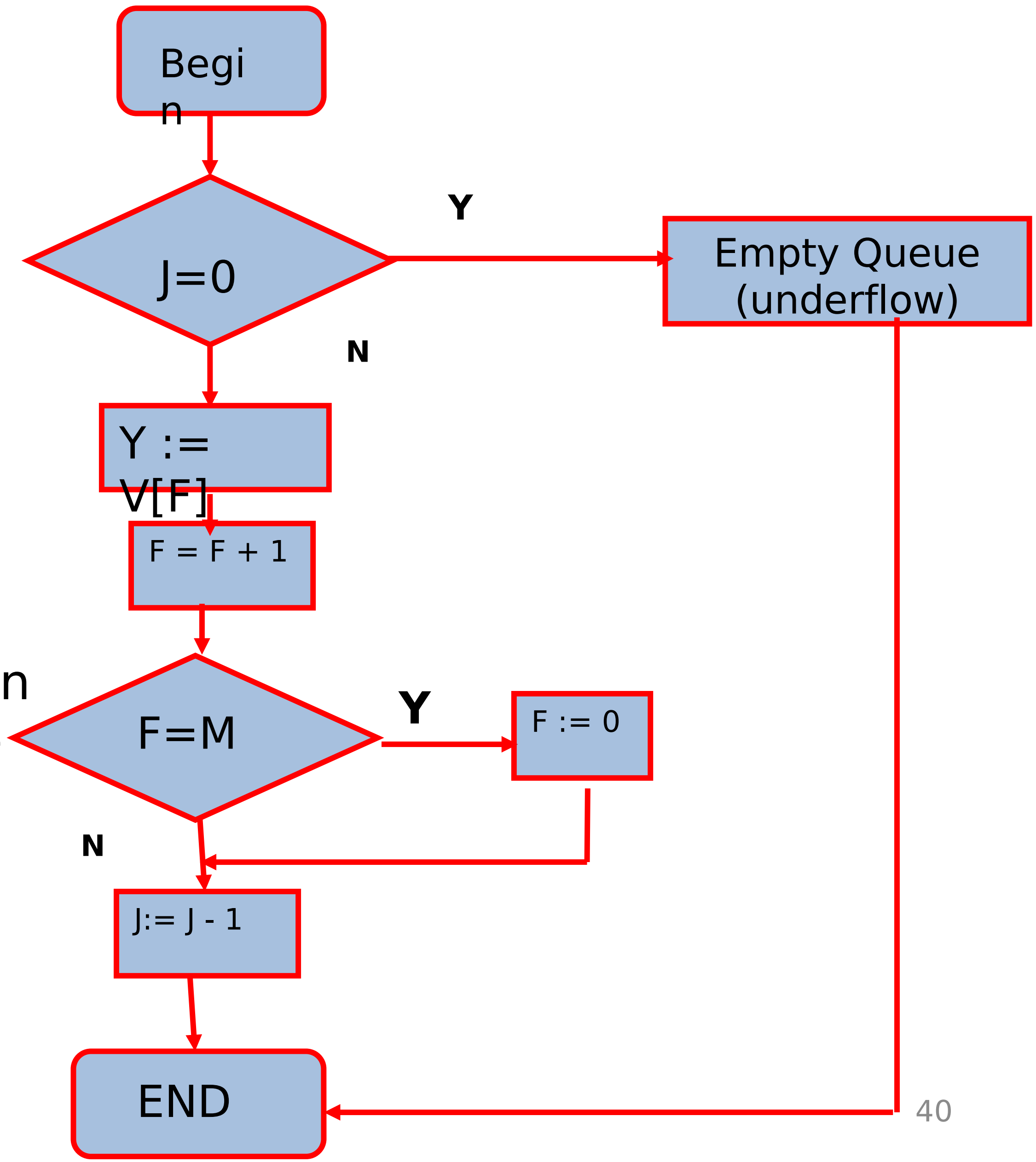
Queues deletion/dequeue;Example

- Deletion of an element

- For this, it’s imperative to check whether queue is empty or not.

R= Rear points to rear Element

F= Front will point to position after first element (after the deletion).



Queue Specification

```
interface QueueMethods
{
   public void enqueue(Object item) throws
                                 QueueOverflowException;
   public void dequeue() throws QueueUnderflowException;
   public Object peek() throws QueueUnderflowException;
   public boolean isEmpty();
   public boolean isFull ();
}

class QueueOverflowException extends Exception
{
   public QueueOverflowException(){;}
}
class QueueUnderflowException extends Exception
{
   public QueueUnderflowException(){;}
}
```

Queue Implementation by Linked List (1)

```
public class Queue implements QueueMethods
{
    public ListNode head, tail;

    public Queue()
    {
        head = null;
        tail = null;
    }

    public boolean isEmpty()
    {
        return ((head == null) ? true: false);
    }

    public boolean isFull()
    {
        return false;
    }
```

Both head and tail references will be maintained, to save traversal time.

The ListNode is the same used by the stack.

Since the structure is dynamic, it can never be ‘full’.

Queue Implementation by Linked List (2)

```
public void enqueue (Object item) throws
                                  QueueOverflowException
{
   ListNode temp = new ListNode (item);

   if (tail == null)
   {
      tail = temp;
      head = temp;
   }
   else
   {
      tail.next = temp;
      tail = temp;
   }
}
```

Insert data at the tail, making use of the reference to it

Queue Implementation by Linked List (3)

```
    public void dequeue() throws QueueUnderflowException
    {
       if (head == null)
          throw new QueueUnderflowException();
       else
          head = head.next;
       if (head == null)
          tail = null;
    }

    public Object peek() throws QueueUnderflowException
    {
       if (head == null)
          throw new QueueUnderflowException();
       else
          return head.data;
    }
}
```

Remove the head from the list.

Maintain the tail reference.

Queue Driver Program (1)

Not dealing with any exceptions.

```
public class QueueDriver
{
    public static void main(String [] args) throws Exception
    {
        Queue myQueue = new Queue();

        for (int i = 1; i <= 5; i++)
        {
            System.out.println("Adding integer : " + i +
                                   " to the queue”);
            myQueue.enqueue(new Integer(i));
        }

        System.out.println("Head of the queue is : " +

((Integer)myQueue.peek()).intValue());
```

Populate the queue with 5 integers.

Print the head of the queue.

Queue Driver Program (2)

```
        System.out.println("Removing 3 elements from queue");

        for (int i = 0; i < 3; i++)
           myQueue.dequeue();

        System.out.println("Head of the queue is : " +

((Integer)myQueue.peek()).intValue());
    }
}
```

Remove 3 elements from the queue.

Print out the new head of the queue.

Applications of Queue

- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :
  - Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
  - In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
  - Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

END.

WAIRAGU G.R.