MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

CSE_2161
DATA STRUCTURES - LAB MANUAL

SECOND YEAR

(Effective from 2022 batch)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MANIPAL INSTITUTE OF TECHNOLOGY
Bengaluru,
MAHE, MANIPAL
KARNATAKA - 560064

CONTENTS

COURSE OBJECTIVES AND OUTCOMES

Course objectives:

This course will enable students to

- To get practical experience in design, develop, implement and analyze linear data structures.
- To get practical experience in design, develop, implement and analyze nonlinear data structures.

Course outcomes:

After studying this course, students will be able to:
1. Apply recursion concepts to problems with arrays, functions, structures and pointers.
2. Implement applications using stacks and queues.
3. Solve problems using linked lists and trees.

Evaluation plan

- Internal Assessment Marks: 60%
  - ✓ Students must work out the C programs in code blocks only.
  - ✓ Students shall submit the lab code along with results obtained under every program as per the schedule. Eg: At the end of lab 2, both lab 1 & lab 2 programs should be submitted in the file.
  - ✓ Marks will be awarded for each lab submission, ie. 10M for lab 1&2. Marks will be reduced if students miss out programs.
  - ✓ In this way there will be 3 such evaluation with total of 60M, along with one internal exam.
- End semester assessment of 2-hour duration: 40%
  - ✓ A complete C program using functions will be asked. The given program must be worked out in code blocks and compiled. The **C program code** should be submitted. The name of file should be like **regno.c Eg: 20100345.c**

- ✓ All laboratory experiments (TWELVE nos) are to be included for practical examination.
- ✓ Strictly follow the instructions as printed on the cover page of answer script
- ✓ Marks distribution: Procedure + Conduction:15+25 (40)
- ✓ Change of experiment is not allowed.

Note:

1. **Students shall do all the programming exercises in CODE BLOCKS only. The URL for downloading and procedure is given below.**
2. **An additional lab exercise on Structures / Pointers (self-study) is kept at the end for student's reference.**

Procedure for CODE BLOCKS downloading and installing.

Code:Blocks is a free C/C++ and Fortran IDE built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable.

URL: https://www.codeblocks.org/downloads/binaries/
https://www.codeblocks.org/downloads/

In this URL you may be able to download for the following operating systems.

- **Windows XP / Vista / 7 / 8.x / 10**
- **Linux 32 and 64-bit**
- **Mac OS X**

Please look into the below YouTube link which details how to download and install code blocks for windows 10.

YouTube: https://www.youtube.com/watch?v=GWJqsmitR2I

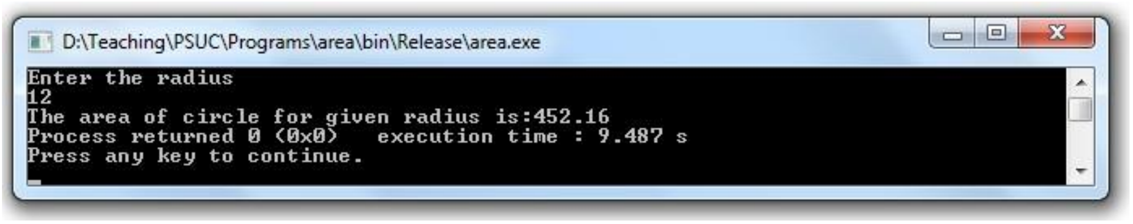
Sample Lab Observation Note Preparation

Title: SIMPLE C PROGRAMS

1. Program to find area of the circle. (Hint: Area=3.14*r*r)

```
//program to find area of circle #include<stdio.h>
int main()
{
int radius; float area;
printf("Enter the radius\n"); scanf("%d", &radius); area=3.14*radius*radius;
printf("My name is abcd"); Name should be printed in every prog.
printf("The area of circle for given radius is: %f", area); return 0;
}
```

Sample input and output: Screen shot of result – must be included.



D:\Teaching\PSUC\Programs\area\bin\Release\area.exe

```
Enter the radius
12
The area of circle for given radius is:452.16
Process returned 0 (0x0)   execution time : 9.487 s
Press any key to continue.
```

LAB NO.: 1

INTRODUCTION

Objectives

In this lab, student will be able to:

A. Learn how to use pointers.
B. To sort the given names using array of pointers.

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Declaration and initialization of pointers

```
int x=10;
int *y; // Declaration of Pointer variable
y=&x; // Storing address of x variable in y pointer variable
```

Usage of pointers

```
int a=3;
int *b;
b=&a;
printf("Value at address %u is %d", b, *b);
```

The output of above code will be something like given below.

```
Value at address 605764 is 3
```

Array of pointers:

"Array of pointers" is an array of the pointer variables. It is also known as pointer arrays.

Syntax:

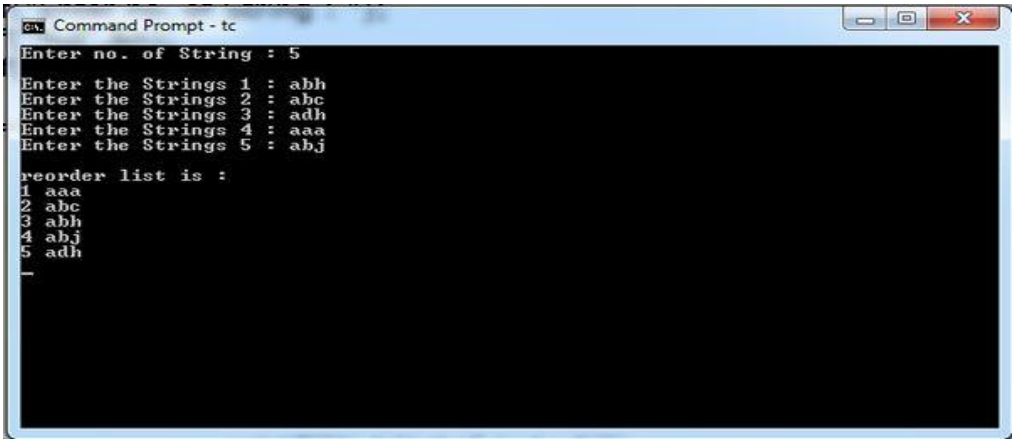
```
int *var_name[array_size];
```

Declaration of an array of pointers:

```
int *ptr[3];
```

ALGORITHM

1. Start
2. Read n for number of names to be sorted
3. Read the names
4. Set loop i=0 to n and j=1 to n
5. Compare the first name with second name if is greater than 0
6. Interchange the first name and the second name using third variable
7. Repeat until all the names are sorted
8. Print the sorted names
9. Stop



```
Enter no. of String : 5

Enter the Strings 1 : abh
Enter the Strings 2 : abc
Enter the Strings 3 : adh
Enter the Strings 4 : aaa
Enter the Strings 5 : abj

reorder list is :
1 aaa
2 abc
3 abh
4 abj
5 adh
_
```

LAB NO.: 2

Recursion and Memory Allocation Functions

In this lab, student will be able to:

A. Learn how to use memory allocation functions
B. Learn how to use recursion

A. Memory allocation functions:

C malloc() method

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.
Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

C calloc() method

1. “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

here, n is the no. of elements and element-size is the size of each element.

C free() method

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```

C realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

```
Syntax:
ptr = realloc(ptr, newSize);
```

<u>B) Algorithm</u> to reverse of a given number using recursion.

Recursion in C

In C, When a function calls a copy of itself then the process is known as Recursion. To put it short, when a function calls itself then this technique is known as Recursion.

If a function calls itself then the function is known as recursive function.

Example:

```
int fib(int num)
{
if (num==1 || num==2)
return 1;
else
return (fib(num-1)+fib(num-2));
```

<u>Iterative method</u>

```
Input:  num
(1) Initialize rev_num = 0
(2) Loop while num > 0
    (a) Multiply rev_num by 10 and add remainder of num
        divide by 10 to rev_num
            rev_num = rev_num*10 + num%10;
    (b) Divide num by 10
(3) Return rev_num
```

Recursive method

```
 reverse( num, rev)
{
   if(num==0)
      return rev;
   else
      return reverse (num/10, rev*10 + num%10);
}
```

Output:

```
Enter any number: 49212
Reverse of input number is: 21294
```

LAB NO.: 3

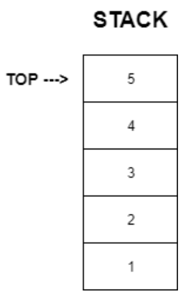
Stack and its Operations

In this lab, student will be able to:

(A) Learn how to implement following operations on the stack using array.
   (i) PUSH

(ii) POP
(iii) PEEK

A Stack is a data structure following the LIFO (Last In, First Out) principle.



PUSH Operation

Push operation refers to inserting an element in the stack. Since there's only one position at which the new element can be inserted—Top of the stack, the new element is inserted at the top of the stack.

POP Operation

Pop operation refers to the removal of an element. Again, since we only have access to the element at the top of the stack, there's only one element that we can remove. We just remove the top of the stack. Note: We can also choose to return the value of the popped element back, its completely at the choice of the programmer to implement this.

PEEK Operation

Peek operation allows the user to see the element on the top of the stack. The stack is not modified in any manner in this operation.

ALGORITHMS:

PUSH(item)

```
Step 1: Read an element to be pushed on to stack item
Step 2: check overflow condition of stack before inserting element into stack Top=max-1
Step 3: update the top pointer and insert an element into stack
Top=top+1
S[top] <-item
```

POP(item)

```
Step1: check underflow condition of stack before deleting element from stack top=-1
Step2:Display deleted element pointed by top
Deleted element<- s[top]
Step3: Decrement top pointer by 1
top<-top-1
```

Peek-

```
1)   if top==NULL then print “empty stack”
2)   go to step 3
3)   return stack[top];
4)   end
```

OUTPUT:

```
Enter the size of STACK[MAX=100]:10

        STACK OPERATIONS USING ARRAY
       --------------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:12

 Enter the Choice:1
 Enter a value to be pushed:24

 Enter the Choice:1
 Enter a value to be pushed:98

 Enter the Choice:3

 The elements in STACK

98
24
```

```
12
 Press Next Choice
 Enter the Choice:2

	The popped elements is 98
 Enter the Choice:3

 The elements in STACK

24
12
 Press Next Choice
 Enter the Choice:4

	EXIT POINT
```

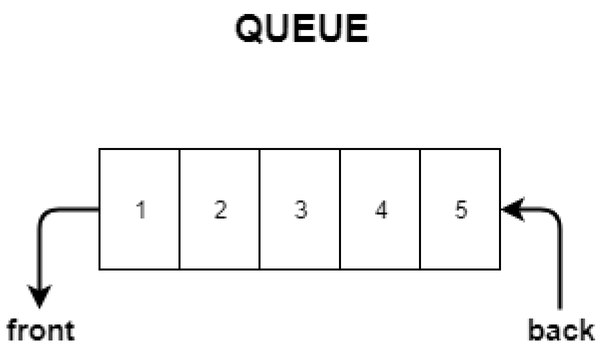
LAB NO.: 4

Queue and its Operations

In this lab, student will be able to:

A) Learn how to implement following operations on the Queue using array.
   (i) Enqueue
   (ii) Dequeue
   (iii) Display

***Queue** is a data structure following the FIFO(First In, First Out) principle.*



Enqueue Operation

Enqueue means inserting an element in the queue. In a normal queue at a ticket counter, where does a new person go and stand to become a part of the queue? The person goes and stands in the back. Similarly, a new element in a queue is inserted at the back of the queue.

Dequeue Operation

Dequeue means removing an element from the queue. Since queue follows the FIFO principle we need to remove the element of the queue which was inserted at first. Naturally, the element inserted first will be at the front of the queue so we will remove the front element and let the element behind it be the new front element.

ALGORITHMS

Enqueue():

Step 1 - Check whether queue is FULL. (rear == SIZE-1)
Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3 - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Dequeue():

Step 1 - Check whether queue is EMPTY. (front == rear)
Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and

terminate the function.
Step 3 - If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Display():

Step 1 - Check whether queue is EMPTY. (front == rear)
Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
Step 4 - Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear)

<u>OUTPUT:</u>

```
Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:1

 Enter no 1:10

Enter the Choice:1

 Enter no 2:54

Enter the Choice:1

 Enter no 3:98

Enter the Choice:1

 Enter no 4:234

Enter the Choice:3

 Queue Elements are:
 10
54
98
```

```
234

Enter the Choice:2

 Deleted Element is 10
Enter the Choice:3

 Queue Elements are:
 54
98
234

Enter the Choice:4
```

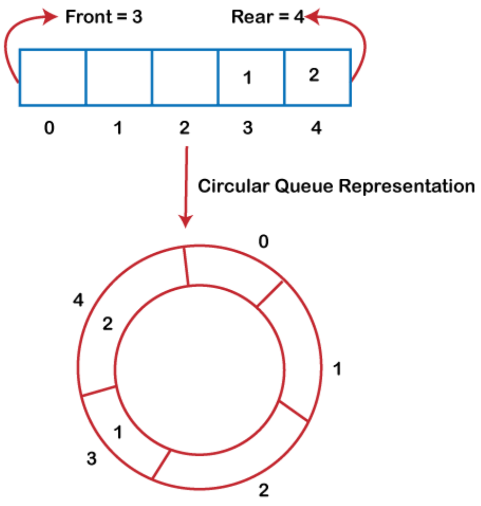
LAB NO.: 5

Circular Queue and its Operations

In this lab, student will be able to:

(A) Learn how to implement following operations on the Queue using array.
  (i) Enqueue
  (ii) Dequeue

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



ALGORITHMS:

The circular queue work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of FRONT to 0
- circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by REAR

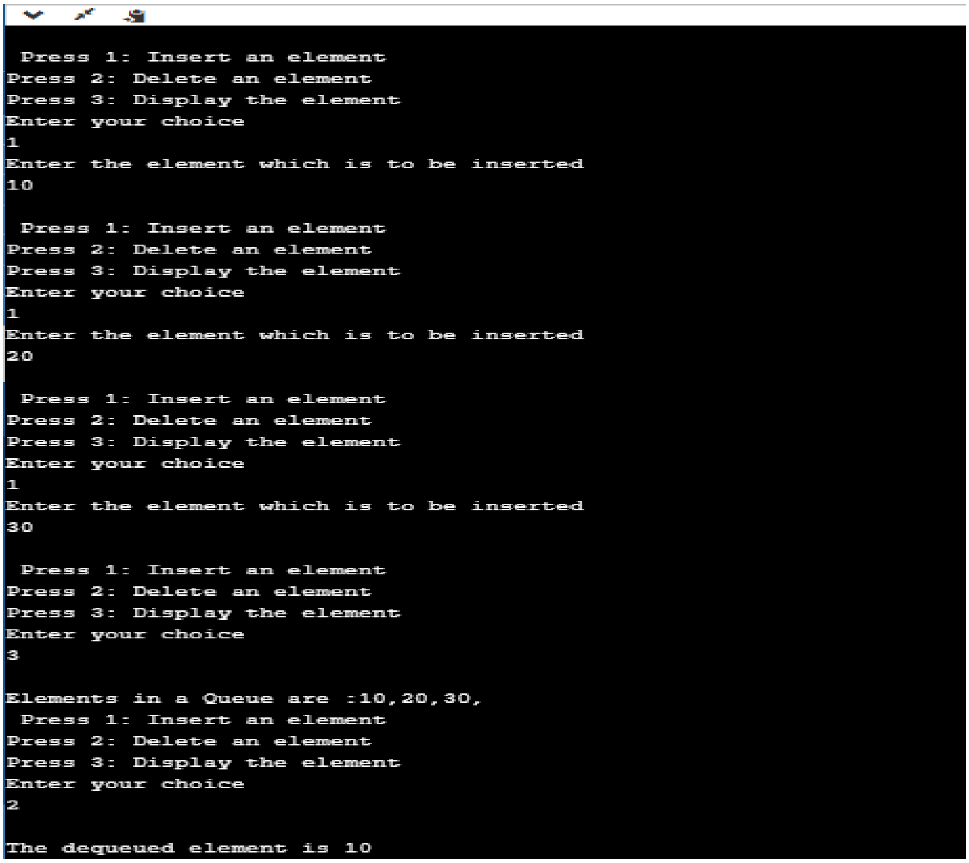
2. Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- circularly increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

However, the check for full queue has a new additional case:

- Case 1: FRONT = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

Output:



```
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

LAB NO.: 6

Singly Linked List and its Operations

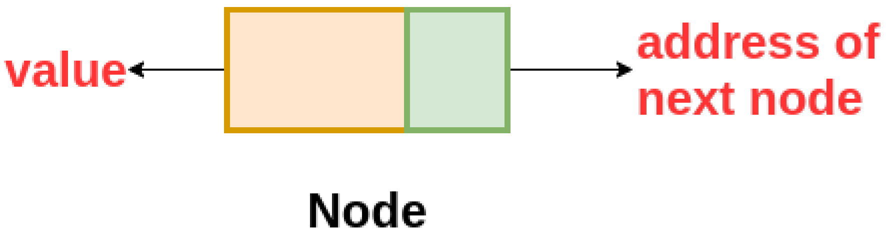
In this lab, student will be able to:

(A) Learn how to perform. Following operations on Singly linked list:

i. Insertion
ii. Deletion
iii. Searching
iv. Traversal

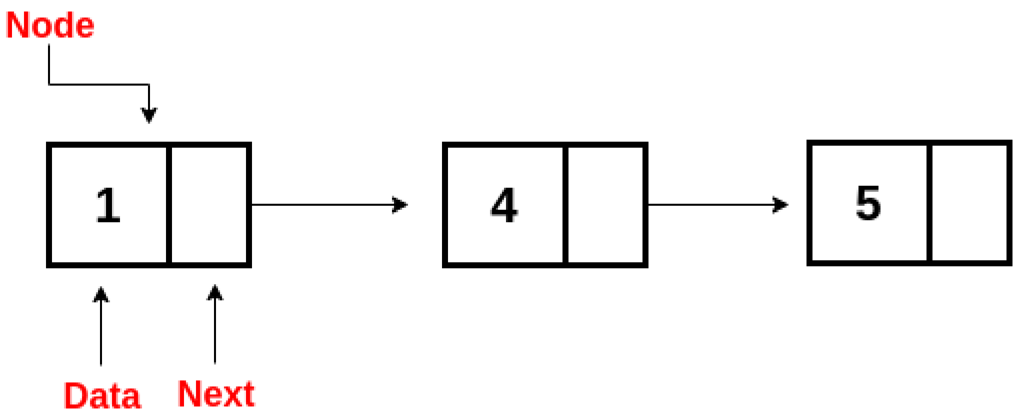
A **Singly-linked list** is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node.

The structure of the node in the Singly Linked List is



Node {
    int data *// variable to store the data of the node*
    Node next *// variable to store the address of the next node*
}

The nodes are connected to each other in this form where the value of the next variable of the last node is NULL i.e. ***next* = *NULL***, which indicates the end of the linked list.



ALGORITHMS:

Insertion at beginning

```
1) Allocate memory for new node.
2) Set new_node→data = val
3) Set new_node→next = start
4) Set start = new _node
5) End.
```

Insertion at end

```
3) Set new_node→next = null
4) Set ptr = start
5) Set step 6
   while
   Ptr→next!=
   NULL
6) Set ptr = ptr→next

   [END OF LOOP]
7) Set ptr→next = new_node
8) Exit
```

Search

```
1) Initialise set ptr = start
2) Repeat step 3 while (ptr)!=NULL
3) If val =
   ptr→data Set
   pos = ptr
   Gotostep 5
4) Set pos = NULL(-1)
5) Exit
```

Delete

```
1) Set ptr = Start
2) Set start = start →next
3) Free ptr
4) Exit
```

Traversal

1. Create a temporary variable for traversing. Assign reference of head node to it, say temp = head.
2. Repeat below step till temp != NULL.
3. temp->data contains the current node data. You can print it or can perform some calculation on it.
4. Once done, move to next node using temp = temp->next;.
5. Go back to 2nd step.

Output:

```
*********Main Menu*********
Choose one option from the following list ...
===============================================
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
1
Enter value
1
Node inserted
```

LAB NO.: 7

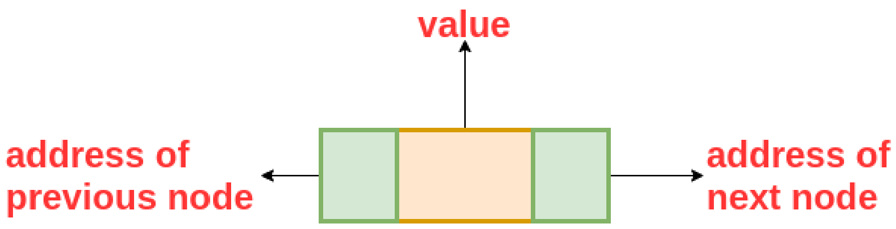
Doubly Linked List and its Operations

In this lab, student will be able to:

(A) Learn how to perform. Following operations on doubly linked list:

v. Insertion
vi. Deletion
vii. Searching

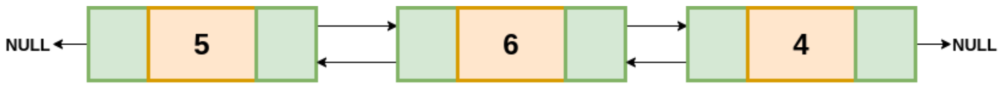
A **Doubly Linked List** contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes.The following is the structure of the node in the Doubly Linked List(DLL):



Node in DLL

```
DLLNode {
    int val // variable to store the data of the node
    DLLNode prev // variable to store the address of the previous node
    DLLNode next // variable to store the address of the next node
}
```

The nodes are connected to each other in this form where the first node has **prev = NULL** and the last node has **next = NULL**.



Doubly Linked List

<u>ALGORITHMS:</u>

Insertion

```
1)  Allocate  memory for new_ node.
2)  Set new_node→data  = val
3)  Set new_node→next  = start
4)  Set  start  = new _node
5)  End.
```

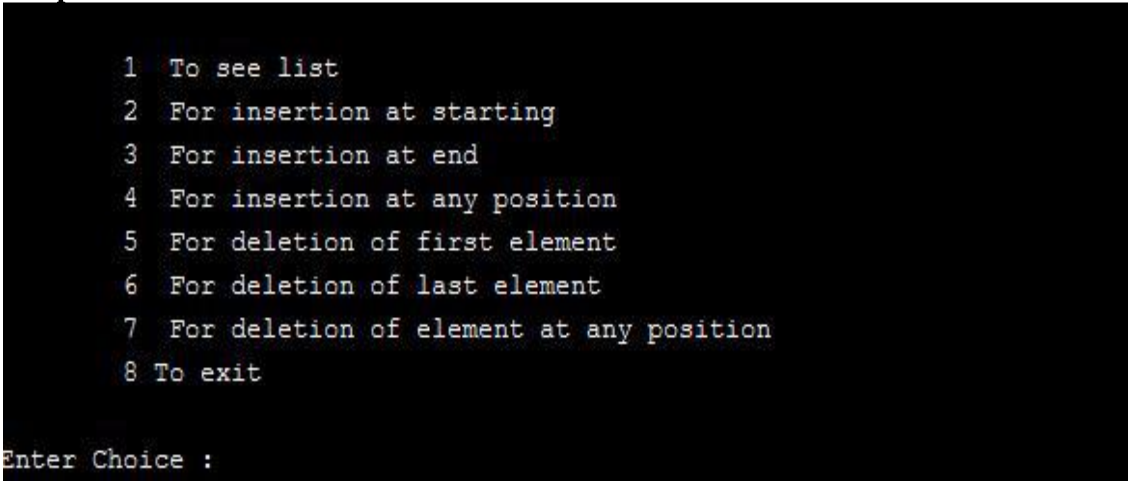
Deletion

```
1)  Set ptr   = Start
2)  Set  start = start  → next
3)  Free ptr
    Exit
```

Searching

```
1)Initialise set  ptr = start
2) Repeat step 3 while (ptr)!=NULL
3)If  val   = ptr→data
4) Set  pos = ptr
        Gotostep  5
5)Set  pos = NULL(-1)
Exit
```

Output:



```
        1  To see list
        2  For insertion at starting
        3  For insertion at end
        4  For insertion at any position
        5  For deletion of first element
        6  For deletion of last element
        7  For deletion of element at any position
        8 To exit

Enter Choice :
```

LAB NO.: 8

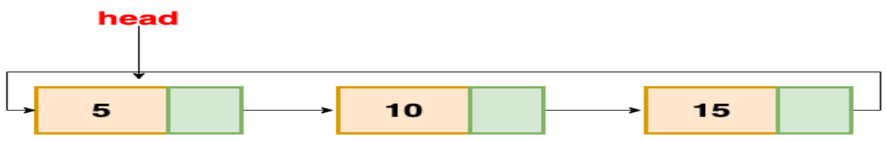
Circular Linked List and its Operations

In this lab, student will be able to:

(B) Learn how to perform. Following operations on circular linked list:

- (i) Insertion
- (ii) Deletion
- (iii) Searching

- A **circular linked list** is either a singly or doubly linked list in which there are no ***NULL*** values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List. In the case of a singly linked list, the next of the last node contains the address of the first node and in case of a doubly-linked list, the next of last node contains the address of the first node and prev of the first node contains the address of the last node.

- 

Circular Linked List

ALGORITHMS

Insertion in circular linked list at the beginning

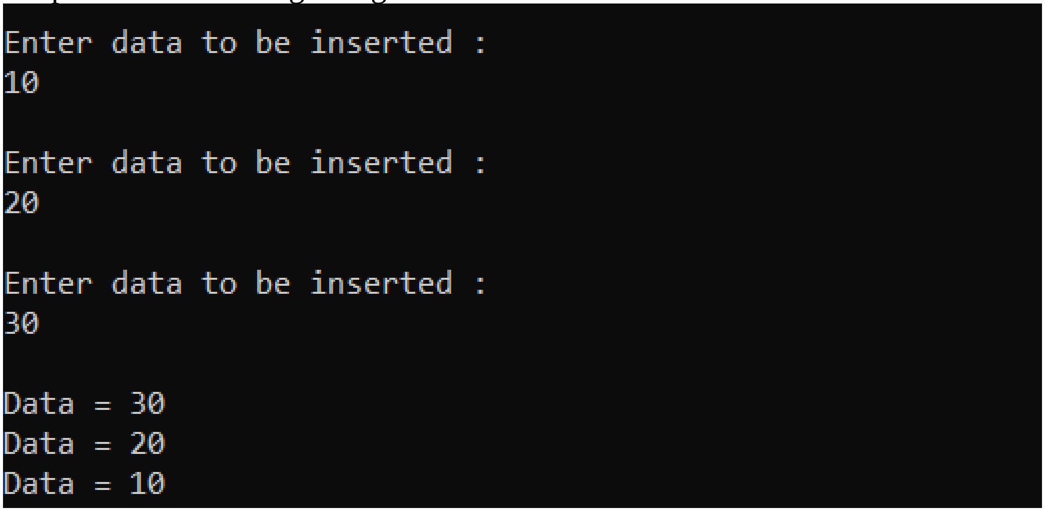
```
Step 1: IF PTR = NULL
  Write OVERFLOW
 Go to Step 11
 [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD
Step 9: SET TEMP → NEXT = NEW_NODE
Step 10: SET HEAD = NEW_NODE
Step 11: EXIT
```

Deletion at the end in circular linked list

```
Step 1: IF HEAD = NULL
  Write UNDERFLOW
   Go to Step 8
  [END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = HEAD
```

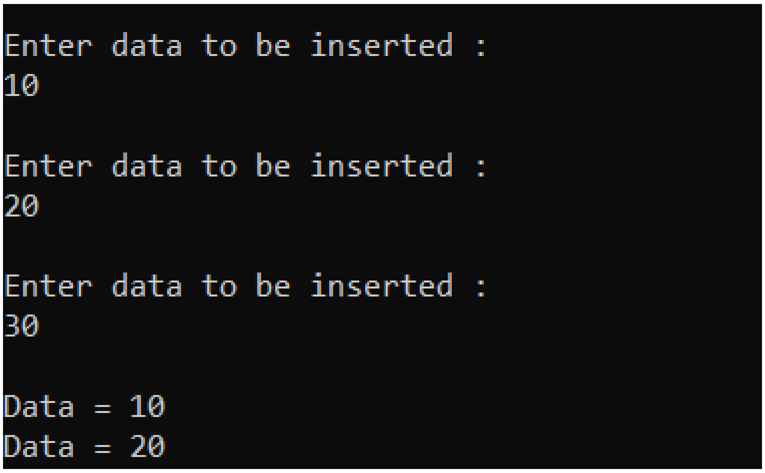
Step 7: FREE PTR
Step 8: EXIT

Output Insert at the beginning



```
Enter data to be inserted :
10

Enter data to be inserted :
20

Enter data to be inserted :
30

Data = 30
Data = 20
Data = 10
```

Deletion at the end



```
Enter data to be inserted :
10

Enter data to be inserted :
20

Enter data to be inserted :
30

Data = 10
Data = 20
```

LAB NO.: 9

Stacks and Queues Using Linked List

In this lab, student will be able to:

(B) Learn how to implement operations of Queue (Enqueue and Dequeue) and Stack (PUSH and POP) using Linked List.

Algorithm for Implementing a Stack using Linked List:

1. **PUSH() Operation:**

```
Step 1: Start
Step 2: Create a node new and declare variable top
Step 3: Set new data part to be Null // The first node is created, having null value and top pointing to it
Step 4: Read the node to be inserted.
Step 5: Check if the node is Null, then print "Insufficient Memory"
Step 6: If node is not Null, assign the item to data part of new and assign top to link part of new and also point stack head to new.
```

2. POP() Operation:

Step 1: Start
Step 2: Check if the top is Null, then print "Stack Underflow."
Step 3: If top is not Null, assign the top's link part to ptr and assign ptr to stack_head's link part.
Step 4: Stop

3. PEEK() Operation:

Step 1: Start
Step 2: Print or store the node pointed by top variable
Step 3: Stop

Algorithm for implementing operations on queue using Linked List:

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.
Step 2 - Define a 'Node' structure with two members data and next.
Step 3 - Define two Node pointers 'front' and 'rear' and set both to NULL.
Step 4 - Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1 - Create a newNode with given value and set 'newNode → next' to NULL.
Step 2 - Check whether queue is Empty (rear == NULL)
Step 3 - If it is Empty then, set front = newNode and rear = newNode.
Step 4 - If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1 - Check whether queue is Empty (front == NULL).
Step 2 - If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
Step 4 - Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is Empty (front == NULL).
Step 2 - If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
Step 5 - Finally! Display 'temp → data ---> NULL'.

Output:
```
Queue :10->20->30->NULL
After dequeue the new Queue :20->30->NULL
After dequeue the new Queue :30->NULL
```

LAB NO.: 10

Binary Search Tree and its operations

Objective:

After this lab the students will be able to understand the following operations in BST:
a) Insert an element into a binary search tree
b) Delete an element from a binary search tree
c) Search for a key element in a binary search tree

A **Binary Search Tree (BST)** is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Algorithms for insertion of a node in BST:

1. Create a new BST node and assign values to it.

2. insert(node, key)

```
i) If root == NULL,
   return the new node to the calling function.
ii) if root=>data < key
   call the insert function with root=>right and assign the return value in root=>right.
  root->right = insert(root=>right,key)
iii) if root=>data > key
   call the insert function with root->left and assign the return value in root=>left.
   root=>left = insert(root=>left,key)
```

3. Finally, return the original root pointer to the calling function.

Algorithms for deletion of a node in BST:

To delete the given node from the binary search tree(BST), we should follow the below rules.

1. Leaf Node

If the node is leaf (both left and right will be NULL), remove the node directly and free its memory.

2.Node with Right Child

If the node has only right child (left will be NULL), make the node points to the right node and free the node.

3.Node with Left Child

If the node has only left child (right will be NULL), make the node points to the left node and free the node.

4.Node has both left and right child

If the node has both left and right child,

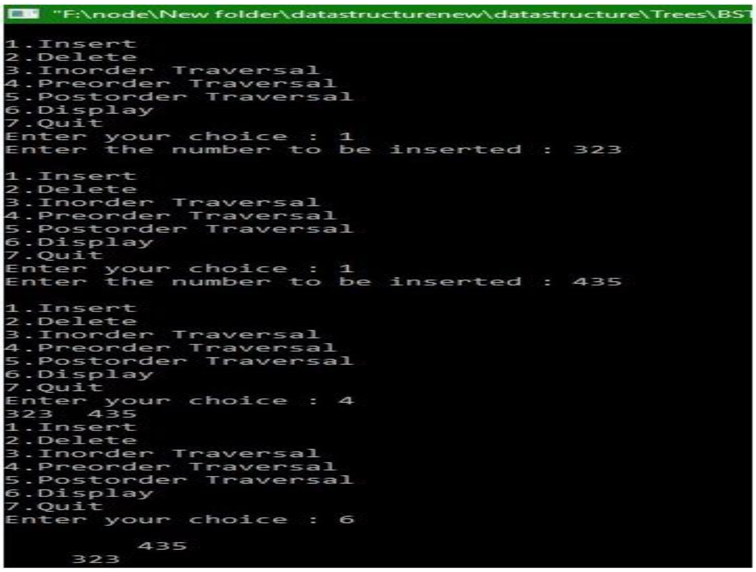
(i) .find the smallest node in the right subtree. say min

(ii) .make node->data = min

(iii) .Again delete the min node.

Algorithm for searching an element in BST:

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.
   Output



```
"F:\node\New folder\datastructurenew\datastructure\Trees\BST
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 323
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 1
Enter the number to be inserted : 435
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 4
323  435
1.Insert
2.Delete
3.Inorder Traversal
4.Preorder Traversal
5.Postorder Traversal
6.Display
7.Quit
Enter your choice : 6

        435
    323
```

LAB NO.: 11

Binary Tree Traversal Algorithms

Objective:

After this lab, the students will be able to perform following traversal algorithms on binary tree:

(i) Preorder
(ii) Inorder
(iii) Postorder

Algorithm:

Preorder (node)

```
1) repeat step 2 to step 4 while tree!=NULL
2) write "TREE→data"
3) preorder(Tree→left)
4) Preorder(Tree→Right)
5) End
```

Postorder (node)

```
1) repeat step 2 to step 4 while tree!=NULL
2) postorder(Tree→left)
3) Postorder(Tree→Right)
```

```
4)    Write"TREE→data"
5)    End
```

Inorder (node)

```
1)    repeat step 2 to step 4 while tree!=NULL
2)    inorder(Tree→left)
3)    write "TREE→data"
4)    inorder(Tree    Right)
5)    End
```

Output:

```
Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 1
Enter element to insert: 5

Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 1
Enter element to insert: 3

Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 1
Enter element to insert: 4

Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 1
Enter element to insert: 6
Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 1
Enter element to insert: 2

Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 2
2   3   4   5   6
Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 3
5   3   2   4   6
Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 4
2   4   3   6   5
Enter your choice:
1. Insert
2. Traverse via infix
3. Traverse via prefix
4. Traverse via postfix
5. Exit
Choice: 5
Memory Cleared
PROGRAM TERMINATED
```

LAB NO.: 12

Finding a Minimum and Maximum Element in Binary Search Tree

Objective:

After this lab, the students will be able to perform following operations in binary search tree:

(ii) Finding a Minimum element

(iii) Finding a Maximum element

Binary Search Tree is a node-based binary tree data structure which has the following properties:

The left subtree of a node contains only nodes with keys lesser than the node’s key.
The right subtree of a node contains only nodes with keys greater than the node’s key.
The left and right subtree each must also be a binary search tree.

Approch for finding minimum element:

Traverse the node from root to left recursively until left is NULL.
The node whose left is NULL is the node with minimum value.

Approch for finding maximum element:

Traverse the node from root to right recursively until right is NULL.
The node whose right is NULL is the node with maximum value.

Output:

```
Minimum value in BST is 1
Maximum value in BST is 14
```

Lab Exam:

- End semester assessment of 2-hour duration: 40%
  - ✓ A complete C program using functions will be asked. The given program must be worked out in code blocks and compiled. The **C program code** should be submitted. The name of file should be like **regno.c Eg: 20100345.c**
- ✓ All laboratory experiments (TWELVE nos) are to be included for practical examination.
- ✓ Strictly follow the instructions as printed on the cover page of answer script
- ✓ Marks distribution: Procedure + Conduction:15+25 (40)
- ✓ Change of experiment is not allowed.

REFERENCES

1. Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C,(3e), Cengage Learning India Pvt. Ltd, India, 2007.
2. Ellis Horowitz, Sartaj Sahni, Susan Anderson and Freed, Fundamentals of Data Structures in C, (2e), Silicon Press, 2007.
3. Richard F. Gilberg, Behrouz A. Forouzan, Data structures, A Pseudocode Approach with C, (2e), Cengage Learning India Pvt. Ltd, India , 2009.
4. 4. Tenenbaum Aaron M., Langsam Yedidyah, Augenstein Moshe J., Data structures using C, Pearson Prentice Hall of India Ltd., 2007.
5. 5. Debasis Samanta, Classic Data Structures, (2e), PHI Learning Pvt. Ltd., India, 2010.

DATA STRUCTURE AND C LANGUAGE QUICK REFERENCE

Array
Stores things in order. Has quick lookups by index.

Dynamic Array
An array that automatically grows as you add more items.

Linked List
Also stores things in order. Faster insertions and deletions than arrays, but slower lookups (you have to "walk down" the whole list).

Queue
Like the line outside a busy restaurant. "First come, first served."

Stack
Like a stack of dirty plates in the sink. The first one you take off the top is the last one you put down.

Hash Table
Like an array, except instead of indices you can set arbitrary keys for each value.

Tree
Good for storing hierarchies. Each node can have "child" nodes.

Binary Search Tree
Everything in the left subtree is smaller than the current node, everything in the right subtree is larger. O(\lg{n})*O*(lg*n*) lookups, but only if the tree is balanced!

Graph
Good for storing networks, geography, social relationships, etc.

Heap
A binary tree where the smallest value is always at the top. Use it to implement a priority queue.

Priority Queue
A queue where items are ordered by priority.

C LANGUAGE REFERENCE

PREPROCESSOR

```
                    // Comment to end of line
                    /* Multi-line comment */
#include <stdio.h>         // Insert standard header file
#include "myfile.h"            // Insert file in current directory
#define X some text        // Replace X with some text
#define F(a,b) a+b         // Replace F(1,2) with 1+2
#define X \
  some text        // Line continuation
#undef X                   // Remove definition
#if defined(X)             // Condional compilation (#ifdef X)
#else                      // Optional (#ifndef X or #if !defined(X))
#endif                     // Required after #if, #ifdef
```

LITERALS

```
255, 0377, 0xff            // Integers (decimal, octal, hex)
2147463647L, 0x7fffffffl // Long (32-bit) integers123.0,
1.23e2                     // double (real) numbers
'a', '\141', '\x61'                  // Character (literal, octal, hex)
'\n', '\\', '\'', '\"',              // Newline, backslash, single quote, double quote
"string\n"                 // Array of characters ending with newline and \0
"hello" "world"            // Concatenated strings
true, false                // bool constants 1 and 0
```

DECLARATIONS

```
int x;                     // Declare x to be an integer (value undefined)
int x=255;                 // Declare and initialize x to 255
short s; long 1;           // Usually 16 or 32 bit integer (int may be either)
char c= 'a';               // Usually 8 bit character
unsigned char u=255; signed char m=-1; // char might be either
unsigned long x=0xffffffffL; // short, int, long are signed
float f; double d;                   // Single or double precision real (never unsigned)
```

```
bool b=true;                    // true or false, may also use int (1 or 0)
int a, b, c;                    // Multiple declarations
int a[10];                      // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};                // Initialized array (or a[3]={0,1,2}; )
int a[2][3]={{1,2,3},{4,5,6}};  // Array of array of ints
char s[]= “hello”;              // String (6 elements including ‘\0’)
int* p;                         // p is a pointer to (address of) int
char* s= “hello”;               // s points to unnamed array containing "hello"
void* p=NULL;                   // Address of untyped memory (NULL is 0)
int& r=x;                       // r is a reference to (alias of) int x
enum weekend {SAT, SUN}; // weekend is a type with values SAT and SUN
enum weekend day;               // day is a variable of type weekend
enum weekend {SAT=0,SUN=1}; // Explicit representation as int
enum {SAT,SUN} day;             // Anonymous enum
typedef String char*;           // String s; means char* s;
const int c=3;                  // Constants must be initialized, cannot assign
const int* p=a;                 // Contents of p (elements of a) are constant
int* const p=a;                 // p (but not contents) are constant
const int* const p=a;           // Both p and its contents are constant
const int& cr=x;                // cr cannot be assigned to change x
```

<u>STORAGE CLASSES</u>

```
int x;                  // Auto (memory exists only while in scope)
static int x;           // Global lifetime even if local scope
extern int x;           // Information only, declared elsewhere
```

<u>STATEMENTS</u>

```
x=y;                    // Every expression is a statement
int x;                  // Declarations are statements
;               // Empty statement
{               // A block is a single statement
   int x;               // Scope of x is from declaration to end of
block
   a;                   // In C, declarations must precede statements
}
if (x) a;                 // If x is true (not 0), evaluate a
else if (y) b;            // If not x and y (optional, may be repeated)
else c;                   // If not x and not y (optional)
while (x) a;              // Repeat 0 or more times while x is true
for (x; y; z) a;          // Equivalent to:  x; while(y) {a; z;}
do a; while (x);          // Equivalent to: a; while(x) a;
switch (x) {              // x must be int
  case X1: a;     // If x == X1 (must be a const), jump here
  case X2: b;     // Else if x == X2, jump here
  default: c;     // Else jump here (optional)
}
break;                    // Jump out of while, do, for loop, or switch
continue;                 // Jump to bottom of while, do, or for loop
return x;                 // Return x from function to caller
try { a; }
catch (T t) { b; }        // If a throws T, then jump here
catch (...)  { c; }       // If a throws something else, jump here
```

FUNCTIONS

```
int f(int x, int);        // f is a function taking 2 ints and returning int
void f();                 // f is a procedure taking no arguments
void f(int a=0);          // f() is equivalent to f(0)
f();                      // Default return type is int
inline f();               // Optimize for speed
f( ) { statements; }      // Function definition (must be global)
```

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

int main() { statements... }   or

int main(int argc, char* argv[]) { statements... }

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

EXPRESSIONS

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation which is undefined. There are no runtime checks for arrays out of bounds, invalid pointers etc.

T::X	// Name X defined in class T
N::X	// Name X defined in namespace N
::X	// Global name X
t.x	// Member x of struct or class t
p → x	// Member x of struct or class pointed to by p
a[i]	// i'th element of array a
f(x, y)	// Call to function f with arguments x and y
T(x, y)	// Object of class T initialized with x and y
x++	// Add 1 to x, evaluates to original x (postfix)
x--	// Subtract 1 from x, evaluates to original x
sizeof x	// Number of bytes used to represent object x
sizeof(T)	// Number of bytes to represent type T
++x	// Add 1 to x, evaluates to new value (prefix)
--x	// Subtract 1 from x, evaluates to new value
~x	// Bitwise complement of x
!x	// true if x is 0, else false (1 or 0 in C)
-x	// Unary minus
+x	// Unary plus (default)
&x	// Address of x
*p	// Contents of address p (*&x equals x)
x * y	// Multiply
x / y	// Divide (integers round toward 0)
x % y	// Modulo (result has sign of x)
x + y	// Add, or &x[y]
x – y	// Subtract, or number of elements from *x to *y
x << y	// x shifted y bits to left (x * pow(2, y))
x >> y	// x shifted y bits to right (x / pow(2, y))
x < y	// Less than

x <= y	// Less than or equal to
x > y	// Greater than
x >= y	// Greater than or equal to
x == y	// Equals
x != y	// Not equals
x & y	// Bitwise and (3 & 6 is 2)
x ^ y	// Bitwise exclusive or (3 ^ 6 is 5)
x | y	// Bitwise or (3 | 6 is 7)
x && y	// x and then y (evaluates y only if x (not 0))
x || r	// x or else y (evaluates y only if x is false(0))
x = y	// Assign y to x, returns new value of x
x += y	// x = x + y, also -= *= /= <<= >>= &= |= ^=
x ? y : z	// y if x is true (nonzero), else z
throw x	// Throw exception, aborts if not caught
x, y	// evaluates x and y, returns y (seldom used)

STDIO.H.H. STDIO.H

cin >> x >> y;	// Read words x and y (any type) from stdin
cout << "x=" << 3 << endl;	// Write line to stdout
cerr « x « y « flush;	// Write to stderr and flush
c = cin.get();	// c = getchar();
cin.get(c);	// Read char
cin.getline(s, n, '\n');	// Read line into char s[n] to '\n', (default)
if (cin)	// Good state (not EOY)?
	// To read/write any type T:

STRING (Variable sized character array)

string s1, s2= "hello";	//Create strings
sl.size(), s2.size();	// Number of characters: 0, 5
sl += s2 + ' ' + "world";	// Concatenation
sl == "hello world";	// Comparison, also <, >, !=, etc.
s1[0];	// 'h'
sl.substr(m, n);	// Substring of size n starting at sl[m]

```
sl.c_str();                     // Convert to const char*
getline(cin, s);                // Read line ending in ‘\n’

asin(x); acos(x); atan(x);      // Inverses
atan2(y, x);                    // atan(y/x)
sinh(x); cosh(x); tanh(x);      // Hyperbolic
exp(x); log(x); log10(x);       // e to the x, log base e, log base 10
pow(x, y); sqrt(x);             // x to the y, square root
ceil(x); floor(x);              // Round up or down (as a double)
fabs(x); fmod(x, y);            // Absolute value, x mod y
```
