A Handbook on Computer Science

4

Programming and Data Structures

CONTENTS

Programming in C

1

- **Variable:** A variable in simple terms is a storage place which has some memory allocated to it.
  - (*i*) **Variable declaration** refers to the part where a variable is first declared or introduced before its first use.
  - (*ii*) **Variable definition** is the part where the variable is assigned a memory location and a value.
  - (*iii*) Mostly variable declaration and definition are done together.
- **const (Constant Variable):** Constant variables are variables which when initialized, can't change their value (the value assigned to them is un-modifiable).
  - (*i*) Constant variables need to be initialized during their declaration itself.
  - (*ii*) const keyword is also used with pointers.

1. **int *ptr; (Pointer to Variable)** We can change the value of ptr and we can also change the value of object ptr is pointing to Pointer and value pointed by pointer both are stored in read-write area.
2. **const int *ptr; or int const *ptr; :** We can change pointer to point to any other integer variable, but cannot change value of object pointed using pointer ptr.
   - (*i*) Pointer is stored in read-write area.
   - (*ii*) Object pointed may be in read only or read write area.
3. **int *const ptr; :** Constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.
4. **const int *const ptr; :** Constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable.

- **Extern:** Extern indicate that the variable is already defined don't create memory again, use previous memory.
  - (*i*) By default, the declaration and definition of a C function have "extern" prepended with them.
  - (*ii*) Declaration can be done any number of times but definition only once.

(*iii*) "extern" keyword is used to extend the visibility of variables/ functions.

(*iv*) When extern is used with a variable, it's only declared not defined.

(*v*) When an extern variable is declared with initialization, it is taken as definition of the variable as well.

- **Static:** Static variables preserve the value of their last use in their scope.
  (*i*) These are initialized only once and exist till the termination of the program (memory is allocated only once compile time itself).
  (*ii*) Their scope is local to the function to which they were defined.
  (*iii*) Global static variables can be accessed anywhere in the program.
  (*iv*) By default, they are assigned the value 0 by the compiler.
  (*v*) By default values : if it has pointer type, it is initialized to a NULL pointer; if it has arithmetic type, it is initialized to (positive or unsigned) zero.
- **Void:** void is a special data type (means an empty data type).
  (*i*) A void pointer is a pointer that has no associated data type with it.
  (*ii*) A void pointer can hold address of any type and can be typcasted to any type.
  (*iii*) **malloc( ) and calloc( ) return void * type** and this allows these functions to be used to allocate memory of any data type.
  (*iv*) void pointers cannot be dereferenced.
- **Typedef:** typedef is used to give a new name to an already existing or even a custom data type (like a structure).

Scope rules in C

- **Global Scope:** Can be accessed anywhere in a program.
- **Local Scope:** Can be accessed inside only declared function.
- **Block Scope:** A Block is a set of statements enclosed within left and right braces {and}. Blocks may be nested in C (a block may contain other blocks inside it). A variable declared in a block is accessible in the block and all inner blocks of that block, but not accessible outside the block.

Storage Classes

- **Auto:** Default storage class for all the variables declared inside a function. They are assigned a garbage value by default when they are declared.

Static: Static variables have a property of preserving their value even after they are out of their scope. Static variables are allocated memory in data segment, not stack segment. In C, static variables can only be initialized using constant literals.

Register: The only difference between auto and register storage is that the compiler tries to store these variables in the register of the processor if a free register is available. **This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.** If a free register is not available, these are then stored in the memory only.

- Using **'&'** with a register variable may result in compiler giving an error or warning, because when we say a variable is a register means that, it may be stored in a register instead of memory and accessing address of a register is invalid.
- Register keyword can be used with pointer variables.
- Register can not be used with static.
- There can be unlimited number of register variables in a program (but the point is compiler may put some variables in register and not the others).

Memory Layout

- **Text segment** (Also known as a **code segment,** one of the sections of a program in memory [read only area], which contains executable instructions).
- **Initialized data segment** (Also known as **Data Segment** [read write area], a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer).
- **Uninitialized data segment** (contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code).
- **Stack** (Stack, where automatic variables are stored, along with information that is saved each time a function is called).
- **Heap** (Heap is the segment where dynamic memory allocation usually takes place and Heap area is shared by all shared libraries and dynamically loaded modules in a process).

Operators in C

- **Arithmetic Operators:** ***(i)*** Addition ('+'), ***(ii)*** Subtraction ('–'), ***(iii)*** Multiplication ('*'), ***(iv)*** Division ('/') and ***(v)*** Modulus ('%').

- **Relational Operators:** (*i*) Equal checking '= =', (*ii*) Not equal to '!=', (*iii*) Grater than '>', (*iv*) Less than '<', (*v*) Grater than equal to '>=' and (*vi*) Less than equal to '<='.
- **Logical Operators:** (*i*) Logical AND ('&&'), (*ii*) Logical OR ('||') and (*iii*) Logical NOT ('!').

Short-Circuiting in Logical Operators

In case of logical AND, the second operand is not evaluated if first operand is false while in case of logical OR, the second operand is not evaluated if first operand is true.

S. No.	Operator	Associativity
1.	( ), [ ] , → , .	Left to right
2.	!, ~, ++, – –, + (unary), – (unary), *, &, sizeof	Right to left
3.	*, / , %	Left to right
4.	+ , –	Left to right
5.	<< , >>	Left to right
6.	<, <=, >, >=	Left to right
7.	==, !=	Left to right
8.	&	Left to right
9.	^	Left to right
10.	|	Left to right
11.	&&	Left to right
12.	||	Left to right
13.	? :	Right to left
14.	=, +=, –=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Right to left
15.	,	Left to right

(Highest) ↑ ↓ (Lowest)

- **Comma has the least precedence among all operators and should be used. There is no chaining of comparison operators in C.**
- **Sizeof( ) function:**
  (*i*) To find out number of elements in a array.
  (*ii*) To allocate block of memory dynamically.
- **Modulus (%) on Negative Numbers:** Sign of left operand is appended to result in case of modulus operator in C.

- **'char s[ ] = "madeeasy"'** creates a character array. We can modify array element but not base address of array, while **'char *s = "madeeasy"'** creates a string literal which stored in read only part of memory so we cannot change string value but can change their base address.

Switch Case

1. The expression used in switch must be integral type (int, char and enum). Any other type of expression is not allowed.
2. All the statements following a matching case execute until a break statement is reached.
3. The default block can be placed anywhere. The position of default doesn't matter, it is still executed if no match found.
4. The integral expressions used in labels must be a constant expressions.
5. The statements written above cases are never executed After the switch statement, the control transfers to the matching case, the statements written before case are not executed.
6. Two case labels cannot have same value.

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where Pointer acts as dangling pointer:

(*i*) De-allocation of memory

(*ii*) Function Call

(*iii*) Void pointer

NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

- **Important Points:**

(*i*) **NULL vs Uninitialized pointer:** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

(*ii*) **NULL vs Void Pointer:** Null pointer is a value, while void pointer is a type

Wild Pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

(*i*) Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

(*ii*) If a pointer p points to a known variable then it's not a wild pointer.

(*iii*) If we want pointer to a value (or set of values) without having a variable for the value, we should explicitly allocate memory and put the value in allocated memory.

Enumeration (or enum)

- Enumeration (or enum) is a user defined data type in C.
- Used to assign names to integral constants.
- Two enum names can have same value.
- Not explicitly assign of values to enum names results in, the compiler by default assigns values starting from 0.
- We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.
- The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.
- All enum constants must be unique in their scope. For example, the following program fails in compilation.

Memory Leak

Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

Strlen ( )

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

arameter Passing

	Actuals used	Actuals changed	Formals changed	Similar technique
Call by value	Yes	No	Yes	–
Call by reference	Yes	Yes	Yes	–
Call by result	No	Yes	Yes	Call by reference
Call by restore (value-result)	Yes	Yes	Yes	Call by reference
Call by need	Yes	No	Yes	Call by value
Call by text	Yes	Yes	–	Call by reference
Call by name	Yes	Yes	–	Call by reference
Call by constant	Yes	No	No	–

Note:

- "Call by reference" and "call by name" parameter passing gives different result while passing address of an array element as a parameter.
- "Call by reference" and "call by value result" parameter passing gives different result in presence of loops.

Scope

Scope of a program variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

Static Scoping

The method of binding names to non local variables called static scoping.

- Scope of variable can be statically determined prior to execution.
- Easier to read, more reliable, and executes faster.

Dynamic Scoping

It is based on the calling sequence of subprograms.

- Scope can be determined at runtime.
- More flexibility, but expense of readability, reliability and efficiency.

■■■

Functions and Recursion

2

Recursion

Recursion is a function which calls itself either directly or indirectly.

- **Factorial function:**

```
int fact (int n)
{
        if (n == 0) return 1;
        else return n*fact (n – 1);
}
```

- **Fibonacci sequence:**

```
int fib (int n)
{
        if (n < 2) return n;
        else return fib (n – 1) + fib (n – 2);
}
```

(*i*) The number of function calls in fib(n) = 2 × fib(n+1)–1

(*ii*) The number of addition in fib(n) = fib(n+1)–1

(*iii*) In fib(n) after (n+1) function calls first addition will be performed.

- **Tower of Hanoi:** (A : source, B : temporary, C : destination)

```
TOH (A, B, C, n)
{
        if (n – 1) C.PUSH (POP (A));
        else
        {
                TOH (A, B, C, n – 1);
                C.PUSH (POP (A));
                TOH (B, C, A, n – 1);
        }
}
```

(*i*) Number of function calls in TOH(n) = $2^{n+1} - 1$

(*ii*) Number of moves in TOH(n) = $2^n - 1$

- **GCD of two numbers (iteration)**

```
f(a, b)
{
        while (b ≠ 0)
        {
                t = b;
                b = a mod b;
                a = t;
        }
        return a;
}
```

- **GCD of two numbers (recursive function)**

```
f(a, b)
{
        if (a = b) return a;
        else if (a > b) return f(a – b, b);
        else f(a, b – a); // if (a < b)
}
```

- **Finding the second maximum of three (distinct) numbers:**

```
int trial (int a, int b, int c)
{
        if ((a > = b) && (c < b)) return b;
        else if (a > = b) return trial (a, c, b);
        else return trial (b, a, c);
}
```

Declarations and Notations

- **int*p;** — p can be a pointer to an integer.
- **int*p[10];** — p is a 10 element array of pointers to integer.
- **int (*p)[10];** — p is a pointer to a 10-element integer array.
- **int *p();** — p is a function that returns a pointer to an integer.
- **int p(char*a);** — p is a function that takes an argument as pointer to a character and returns an integer.
- **int*p(char *a);** — p is a function that takes an argument as pointer to a character and returns a pointer to an integer.
- **int (*p)(char *a);** — p is a pointer to a function that takes an argument as pointer to a character and returns an integer.

- `int (*p(char *a))[10];` p is a function that takes an argument as pointer to a character and returns a pointer to a 10 element integer array.
- `int p(char (*a)[]);` p is a function that takes an argument as pointer to a character array and returns an integer.
- `int p(char *a[]);` p is a function that takes an argument as array of pointers to characters and returns an integer.
- `int *p(char a[]);` p is a function that takes an argument as character array and returns a pointer to an integer.
- `int *p(char (*a) []);` p is a function that takes an argument as pointer to a character array returns a pointer to an integer.
- `int *p(char *a[]);` p is a function that takes an argument as array of pointers to characters and returns a pointer to an integer .
- `int (*p)(char (*a)[]);` p is a pointer to a function that takes an argument as pointer to a character array and returns an integer.
- `int *(*p)(char (*a)[]);` p is a pointer to a function that takes an argument as pointer to a character array and returns a pointer to an integer.
- `int *(*p)(char *a[]);` p is a pointer to a function that takes an argument as array of pointers to characters and returns a pointer to an integer.
- `int (*p[10])();` p is a 10 element array of pointers to functions, each function returns an integer.
- `int (*p[10])(char a);` p is a 10 element array of pointers to functions each functions takes an argument as character, and returns an integer.
- `int *(*p[10])(char a);` p is a 10 element array of pointers to functions, each function takes an argument as character, and returns a pointer to an integer.
- `int *(*p[10])(char *a );` p is a 10 element array of pointers to functions, each function takes an argument as pointer to a character, and returns a pointer to an integer.

■■■

Abstract Data Types, Arrays and Linked Lists

3

Introduction

Data Structures based on the manipulation criteria:

1. **Primitive Data structures:** Integers, real numbers, characters, boolean values.
2. **Non-primitive data structures:** Arrays, stacks, queues, records, files, linked lists, tress, graphs, etc.

Data structures based on the storage criteria:

1. **Linear Data structures:** Arrays, records, stacks, queues, etc.
2. **Non-linear data structures:** Trees, Graphs, etc.

Linked Lists

- Linked list is a collection of nodes (contains data and pointer).
- **Properties:** Follow dynamic allocation and non-contiguous in nature.

Operations on Single Linked Set

Insertion

- Number of pointers modified to insert a new node at end = 2

```
node → next = newnode;
newnode → next = null;
```

- Number of pointers modified to insert a new node at beginning = 2

```
newnode → next = root;
root = newnode;
```

- Number of pointers modified to insert a new node in mid = 2

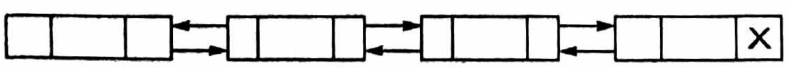
```
newnode → next = node → next;
node → next = newnode;
```

Deletion

- Number of pointers modified to insert a new node at beginning = 1

```
head → next = head → next → next;
```

Operations on Double Linked Set



```
struct node
```

```
{
    int data;
    struct node * prev;
    struct node * next;
}
```

Insertion

- Number of pointers are modified to insert a node at beginning = 4

```
newnode → next = root;
newnode → prev = null;
root → prev = newnode;
root = newnode;
```

- Number of pointers are modified to insert a node at the end = 3

```
newnode → next = null;
newnode → prev = node;
node → next = newnode;
```

- Number of pointers are modified to insert a node in between = 4

```
newnode → next = node → next;
newnode → prev = node;
(node → next) → prev = newnode;
node → next = newnode;
```

Deletion

- Number of pointers are modified to delete a node in middle = 2

```
(node → prev) → next = node → next;
(node → next) → prev = node → prev; free (node)
```

Array

Array is a collection of homogeneous elements stored in contiguous memory location.

Properties: Static in nature, Compile time early binding and user friendly.

1-D Array

Let '*A*' be an array of *n* elements. Address of an element ***A[i]***:

Base(*A*) + (*i* – start index) × size of element.

2-D Array

Let *A*[*m*][*n*] be a 2-*D* array with *m* rows and *n* columns.

- Address of an element ***A[i][j]*** in Row Major order:

Base (A) + (j – start index) × size of element +
(i – start index) × size of element × n

Address of an element A[i][j] in Column Major order:

Base (A) + (i – start index) × size of elements +
(j – start index) × m × size of elements

Array Operations

Insertion

- [best case] At end takes constant time i.e. $\Omega(1)$
- [worst case] At beginning takes $O(n)$ time
- [Average case] In middle takes $\theta(n)$ time

Deletion

- [best case] At end takes constant time i.e. $\Omega(1)$
- [worst case] At beginning takes $O(n)$ time
- [Average case] In middle takes $\theta(n)$ time

Traversal

Direct access: $O(1)$

Triangular Matrices

- Lower Triangular $[\triangle]_{n\times n}$:

  (i) Size $= n+\frac{n^2-n}{2}=\frac{n(n+1)}{2}$

  (ii) Row Major Order (RMO): $(j-1)+\frac{i(i-1)}{2}$

  (iii) Column Major Order (CMO): $a[i][j]=(i-j)+\left[(j-1)n-\frac{(j-1)(j-2)}{2}\right]$

- Upper Triangular $[\triangledown]_{n\times n}$:

  (i) Size $= \frac{n(n+1)}{2}$

  (ii) RMO: $a[i][j]=(j-i)+\left[(i-1)n-\frac{(i-1)(i-2)}{2}\right]$

  (iii) CMO: $a[i][j]=\left[(i-1)+\frac{j(j-1)}{2}\right]$

- **Strictly lower triangular** $[\diagdown]_{n\times n}$

  (*i*) Size = $\dfrac{n^2-n}{2}$

  (*ii*) RMO: $a[i][j] = (j-1) + \dfrac{(i-1)(i-2)}{2}$

  (*iii*) CMO: $a[i][j] = (i-1) + \dfrac{(j-1)(j-2)}{2}$

- **Tridiagonal** $[\diagdown\!\diagdown\!\diagdown]_{n\times n}$:

  (*i*) Size = $3n-2$

  (*ii*) Row: $2i + j - 3$

  (*iii*) Column: $i + 2j - 3$

■■■

Stacks and Queues

4

Stack

A stack is an ordered list in which all insertion and deletions are made at one end. Top is a pointer pointing to the top most element of the stack.

Applications of Stack

1. Recursion
2. Post fix notation (Evaluation of expression)
3. Conversion of arithmetic expression from infix to polish (prefix/postfix)
4. Tower of hanoi
5. Fibonacci series
6. Permutation
7. Balancing of symbols
8. Subroutines

Stack Operations

Operations on stack are: push( ), pop( ), Is empty( ), Is full( ), peep( ) and change( ).

- **Push operation on stack:** Let N is the maximum stack size and x is the element to be inserted on to stack S.

```
push(S, Top, N, x)
{
    if (Top == N – 1)
    {
        printf("stack is overflow");
        exit(1);
    }
    Top ++;
    S [Top] = x;
}
```

- **Pop operation on stack:**

```
pop (S, Top, N)
{
    int y;
    if (Top==–1)
```

```
    {
        printf ("under flow");
        exit(1);
    }
    y = S[Top];
    Top – –;
    return (y);
}
```

Queue

It is first-in first-out (FIFO). The fist item inserted is the first to be removed. One side insertion (Rear), other side deletion (front).

- **Queue Operations:** enqueue( ), Dequeue( ), Is empty( ) and Is full( ).
- **Double ended queue:** Insertion and deletion can be on both sides.
- **Input restricted queue:** Restriction only on i/p not on deletion.
- **Output restricted queue:** Deletion on one side and no restriction of insertion.
- **Priority queue:** Ascending priority queue/Descending priority queue.
- A queue can be implemented using two stacks.

Enqueue Operation for QUEUE

- QUEUE [0: n-1] is an array, 'front' stores the subscript value of the first element of the queue, 'rear' stores the subscript value of the last element of the queue, 'front' and 'rear' are initialized to –1 when the queue is empty, and 'element' is the new element to be inserted into the queue.

```
Enqueue (QUEUE, n, front, rear, element)
{
    If (((front == 0) and (rear == n–1)) or (rear ==front – 1))
    {
        Print "Overflow"
        exit( );
    }
    if (front == –1)
        front = rear = 0;
    else if (rear == n – 1)
        rear = 0;
    else
```

```
        rear = rear + 1;
    QUEUE [rear] = element;
    exit( );
}
```

queue Operation for QUEUE

```
Dequeue (QUEUE, n, front, rear)
{
    if (front == –1)
    {
        Print "Underflow"
        exit( );
    }
    element = QUEUE [front];
    if (front == rear)
        front = rear = –1;
    else if (front == n – 1)
        front = 0;
    else
        front = front + 1;
    exit( );
}
```

plications of Queue

Multiprogramming
Resource sharing
BFS
CPU scheduling
Radix sort
Real time system
Storing operands in evaluation of prefix expressions.

que (Double Ended Queue)

s a linear data structure in which insertions and deletions can be made at th the ends. There are four possible operations: Insert at the left, Delete the left, Insert at the right and Delete at the right.

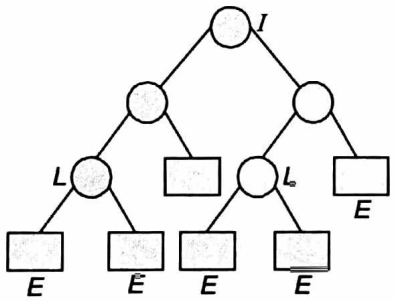
■■■

Trees, BSTs and Binary Heaps

5

Tree

- Tree is defined as $T(V, E)$ where V = set of vertices and E = set of edges.
- In a tree there is exactly one edge between a pair of vertices and there are no cycles and self loops.
- The degree of a node in the tree is defined as the maximum number of children that node can hold.
- A tree can never be empty, but binary tree may be empty.
- Internal node (I): A node having minimum of 1 child is known as internal node.
- Leaf node (L): A node having no children.
- External node (E): The replacement of null leaves are external nodes.



- **Properties:**
  - (*i*) For 'n' nodes there are $(n-1)$ edges.
  - (*ii*) Let n_0 = number of nodes with degree 0.
    n_2 = number of nodes with degree 2.
    $n_0 = n_2 + 1$
  - (*iii*) (In heap array) If child is at location i(index starts from 0) then parent is at location $\lfloor (i-1)/2 \rfloor$.
    If parent is at location i(index starts at 0) then left child is at location $2i + 1$ and Right child is at location $2i+2$.
  - (*iv*) (In heap array) If parent is at location i(indexing at 1) then Left child is at location $2i$ and Right child is at location $2i+1$.
    If child is at location i(indexing at 1) then parent is at location $\lfloor i/2 \rfloor$.

(v) In a linked representation of binary tree, if there are 'n' nodes then number of NULL links = n +1.

- **Complete Binary Tree** is a tree in which nodes are inserted/filled from left to right. All the leaves may not be at the same level.
- **Full Binary Tree** is a tree in which every node has exactly 2 children and all the leaves are at the same level.
- All full binary trees are complete binary trees but vice versa is not true.
- **Strict Binary Tree** is a tree in which each node has exactly 0 or 2 children.
- A complete n-ary tree is one in which every node has '0' or 'n' sons. If 'x' is the number of internal nodes of a complete n-any tree, the number of leaves in it is: $x(n-1)+1$.
- Let $T(n)$ be the number of different binary search trees on n-distinct elements, then $T(n) = \sum_{K=1}^{n} T(k-1)\,T(n-k+1)$.
- In binary tree, Rank (or) Index of any node:

  Rank (i) = (Number of nodes in left subtree of i) + 1
- In a m-ary tree (degree $\leq m$), if n_i is number of nodes of degree 'i' ($i = 0, 1, \ldots, m$) then n_0(leaf nodes) $= 1 + \sum_{i=2}^{m} (i-1)n_i$
- Maximum size of array to store a binary tree with 'n' nodes = $2^n - 1$.
- Minimum size of array to store a binary tree with 'n' nodes = $2^{\lceil \log(n-1) \rceil} - 1$.
- Maximum height possible for a binary tree with 'n' nodes = n
- Minimum height possible for a binary tree with 'n' nodes = $\log_2 n$
- Maximum number of nodes in a binary tree of height 'h' = $2^{h+1} - 1$.
- Total number of binary trees possible with n nodes = ${}^{2n}C_n / (n+1)$
- For non-empty binary tree, if n is number of nodes and e is the number of edges, then $n = e + 1$.

Binary Tree Traversals

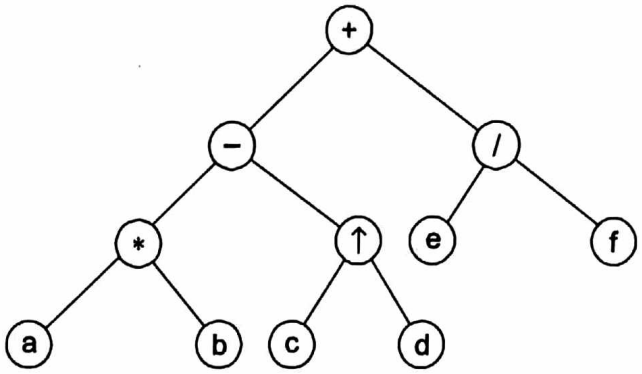
Traversing a tree means a visiting each node in a specified order.

Traversal Techniques

1. Preorder (Root, Left, Right)

2. Inorder (Left, Root, Right)
3. Postorder (Left, Right, Root)

Example:



Preorder Traversal: + − * ab ↑ cd / ef

Inorder Traversal: a * b− c ↑ d + e / f

Postorder Traversal: ab * cd ↑ − ef / +

Level Order Traversal: + − / * ↑ ef abcd

Binary Search Tree (BST)

- Left child < Root < Right Child
- Recursion of left child from the root gives the minimum element and Recursion of right child from the root gives the maximum element.
- Inorder traversal of BST is called "Binary Sort". Sorted order of data results in ascending order.
-

	Average Case	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Build		$O(n^2)$

- **"Deletion of a node" in BST:**
  - (*i*) If node has no child, simply delete that node.
  - (*ii*) If node has 1 child, delete that node and replace it with child.
  - (*iii*) If node has 2 children, delete that node and replace it with inorder successor or predecessor of that node.

Heap Tree

- Heap tree is a complete binary tree with heap property (min heap/max heap).
- There are two types of heap trees:
  - (*i*) **Min heap:** At any subtree the root value always smaller than its children.
  - (*ii*) **Max heap**: At any subtree the root value always greater than its children.

- **Heap Sort:** By deleting the root node from heap tree and placing the deleted node in output, until all the nodes are deleted.
- It gives either ascending order (min heap) or descending order (max heap).

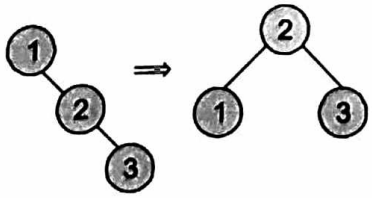
AVL Search Tree

- Devised by **Adelson Velski Landis**
- Height balanced binary search tree
- Balance factor (b_f) is :

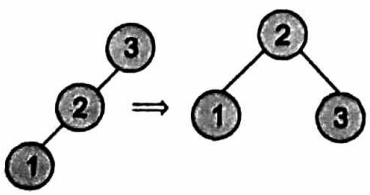
$$|b_f| = |h_L - h_R| \le 1$$

$$b_f = -1 \text{ or } 0 \text{ or } 1$$

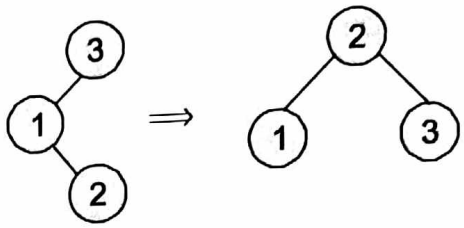
- Rotation is a technique used in the AVL tree to balance the height.
- **Four types of rotations are:**
  - (*i*) **Right-Right [RR] Rotation (Single Rotation):**



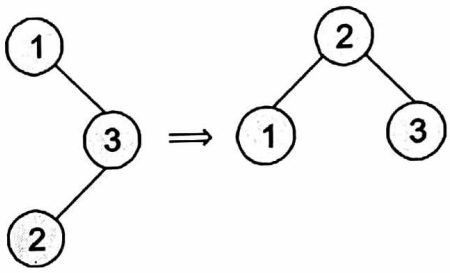
  - (*ii*) **Left-Left [LL] Rotation (Single Rotation):**



(*iii*) Left-Right [LR] Rotation (Double Rotation):



(*iv*) Right-Left [RL] Rotation (Double Rotation):



- Minimum number of nodes in a AVL tree of height '*h*'

$$N_{min}(H) = \begin{cases} 1 & ; \quad L = 0 \\ 2 & ; \quad L = 1 \\ 1 + N(H-1) + N(H-2) & ; \quad L > 1 \end{cases}$$

Time and Space Complexity with each Data Structure

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log (n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Time and Space Complexity with each Data Structure

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log (n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)