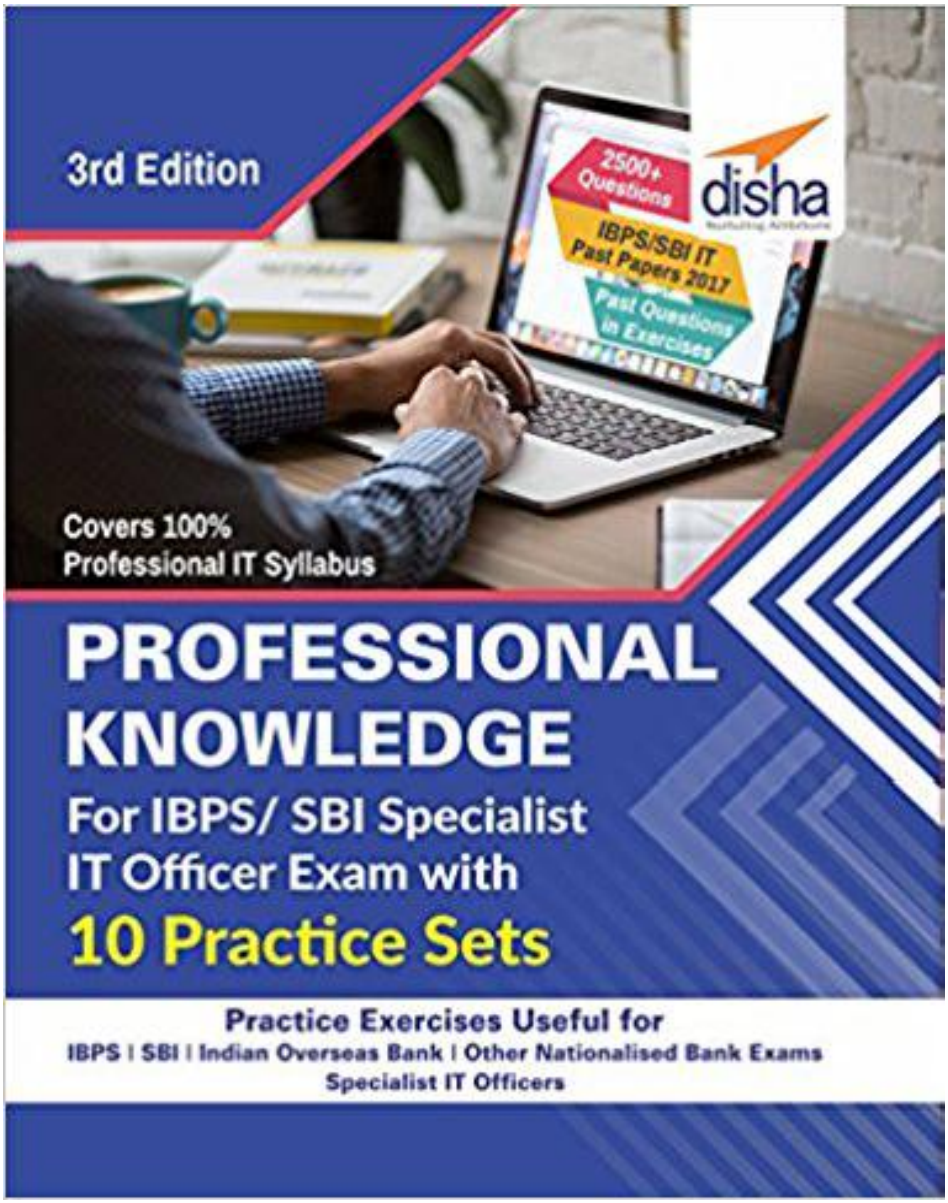
Data Structures

This chapter "Data Structures" **is taken from our book:**



ISBN : 9789386320971

CHAPTER 3

DATA STRUCTURES

DATA TYPE

It is representation of information using a set of value and set of operations required for deriving further results and consider an example A day's rain is expressed in discrete form as millimeter. This value can be subjected to a set of operation such as add to derive the total rain in a year, Division to derive average rain in a year. Unstructured or scalar: Integer, float, char and Pointer, homogenous: Array, string, enum, structure and Union, heterogeneous : ADT like list, queue, stack, tree and graph.

DATA STRUCTURE

It is way of organizing value with help of existing data types, ex: Accumulation of rain.

Data for one year and apply some operation to derive statistical results. Data of 365 days need integer to store 365 values in the list- one dimension and 10 different regions require storing - 2D. It is a aggregation of different type of data by which the stored data can be made more explanatory. Hence, the Data structure is require through knowledge of data types available in a Programming Language.

Data structure can be also defined as; it is the mathematical model which helps to store and retrieve the data efficiently from primary memory. It helps to consistently maintain the data as well as the implement-n functions of interest for data.

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out access	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes (If the tree remains balanced)	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes (Tree always remains balanced)	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes (Tree always remains balanced) (Similar trees good for disk storage)	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex

ARRAY AND ARRAY TYPE

An array is a collection of homogeneous data elements described by a single name.

ONE DIMENSIONAL ARRAY

One-dimensional array or linear array is a set of 'n' finite numbers of homogenous data elements such as:

1. The elements of the array are referenced respectively by an index set consisting of 'n' consecutive numbers.
2. The elements of the array are stored respectively in successive memory locations. 'n' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as
   A[0], A[1], A[2] ..... A[n–1]
   The number 'n' in A[n] is called a subscript or an index and A [n] is called a subscripted variable.

MULTI DIMENSIONAL ARRAY

The elements of an array can be of any data type, including an array of arrays is called a multidimensional array. In this case, since we have 2 subscripts, this is a two-dimensional array. In a two-dimensional array, it is convenient to think of the first subscript as being the row, and the 2nd subscript as being the column. Two loops are required. Similarly the array of 'n' dimensionals would require 'n' loops.

SPARSE ARRAY

A sparse array is an array where most of elements have the value zero "0" and only a few non-zero values. One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called sparse matrix.

STRING AND STRING REPRESENTATION

STRING

A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string. A string is defined as a sequence of characters.

STRING REPRESENTATION

Strings are stored or represented in memory by using following three types of structures:

- Fixed length structures
- Variable length structures with fixed maximum
- Linear structures

STACKS

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

A stack is a dynamic, constantly changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place.

The last element inserted into the stack is the first element deleted-last in first out list (LIFO). After several insertions and deletions, it is possible to have the same frame again.

RECURSION AND ITERATION

No.	Iteration	Recursion
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.
2	Iteration involves four clear-cut steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory, intilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

EXPRESSION

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation
2. Prefix notation
3. Postfix notation

The infix notation is what we come across in our general mathematics, where the operator is written in between the operands. For example: The expression to add two numbers A and B is written in infix notation as :

A + B

Note that the operator '+' is written in between the operands A and B. The prefix notation is a notation, in which the operator is written before the operands. As the operator '+' is written before the operands A and B, this notation is called prefix.

The postix notation the operator(s) are written after the operands, so it is called the postix notation (post means after), it is also known as suffix notation or reverse polish notation, The above expression if written in postfix expression looks like:

A B +

Converting infix to postfix expression

The method of converting infix expression A + B * C to postifx form is :

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

STACK OPERATION

The primitive operations performed on the stack are as follows:

Push : The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add

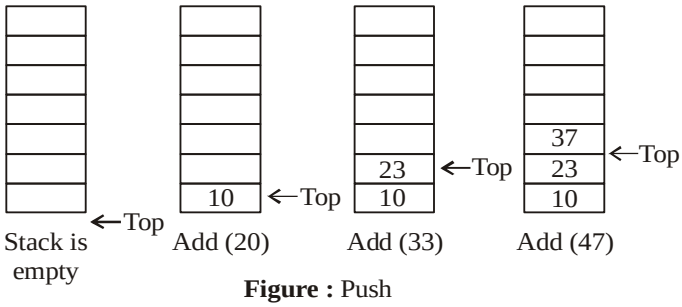


Figure : Push

the new element at the top. After every push operation the top is incremented by by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

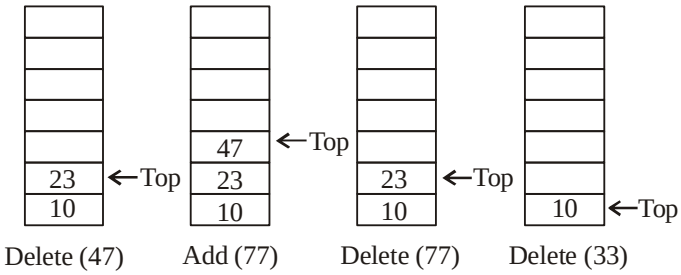


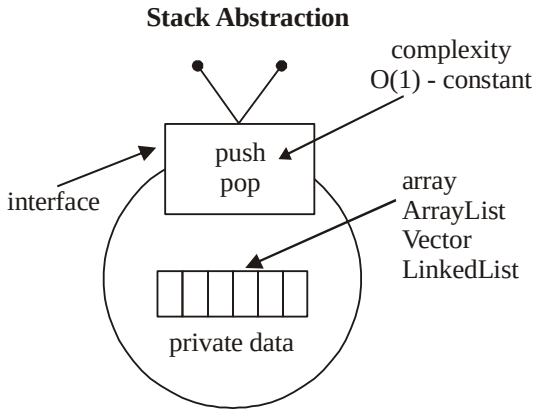
Figure : Pop

Pop : The process of deleting (or removing) and element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

STACK IMPLEMENTATION

In the standard library of classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality.

The following picture demonstrates the idea of implementation by composition.

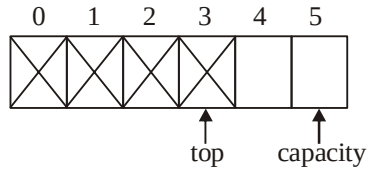


Class creation by composition

Another implementation requirement (in addition to the above interface) is that all stack operations must run in **constant time O(1).** Constant time means that there is some constant k such that an operation takes k nanoseconds of computational time regardless of the stack size.

Array-based implementation

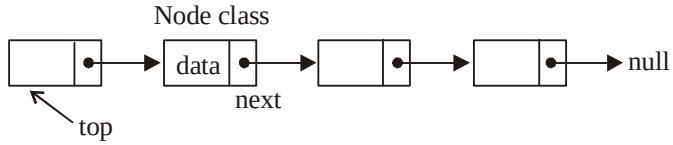
In an array-based implementation we maintain the following fields: an array A of a default size (≥1), the variable top that refers to the top element in the stack and the *capacity* that refers to the array size. The variable top changes from –1 to capacity –1. We say that a stack is empty when top = –1, and the stack is full when top = capacity –1.



In a fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception. In a dynamic stack abstraction *when* top reaches *capacity*, we double up the stack size.

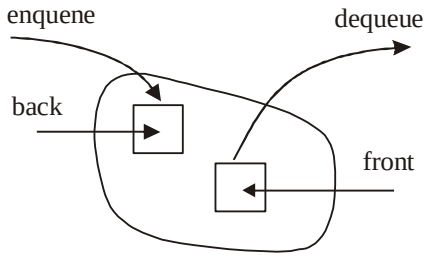
Linked List-based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



QUEUES

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.



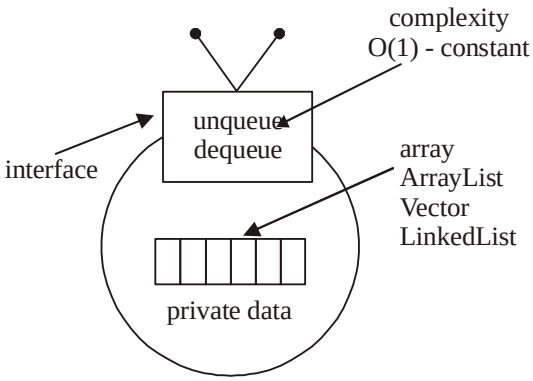
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Implementation

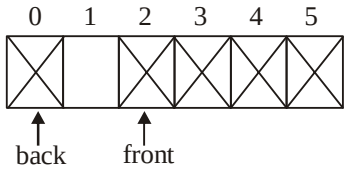
In the standard library of classes, the data type queue is an adapter class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type of the underlying data structure, a queue must implement the same functionality.

Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.

Queue Abstraction



class creation by composition



Finally, when back reaches *front*, the queue is full. There are two choices to handle a full queue: (a) throw an exception; (b) double the array size.

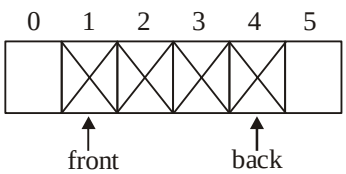
The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8% 5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array_size". In addition to the back and front indexes, we maintain another index: cur - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

Types of Queue

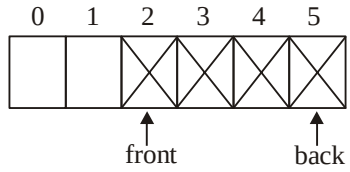
There are three major variations in a simple queue. They are

1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue.

Circular Queue : Circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full. Suppose if we have a Queue of n elements then after adding the element at the last index i.e. (n -1) th, as queue is starting with 0 index, the next element will be inserted at the very first location of the queue which was not possible in the simple linear queue. That's why linear queue leads to wastage of the memory, and this flaw of linear queue is overcome by circular queue, So, in all we can say that the circular queue is a queue in which first element come right after the last element that means a circular queue has a starting point but no end point. Given an array A of a default size (≥1) with two references back and front, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until front. Such a model is called a wrap around queue or a circular queue.

Double Ended Queue (de-queue) : A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). i.e.; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

Priority Queues : Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

LINKED LISTS

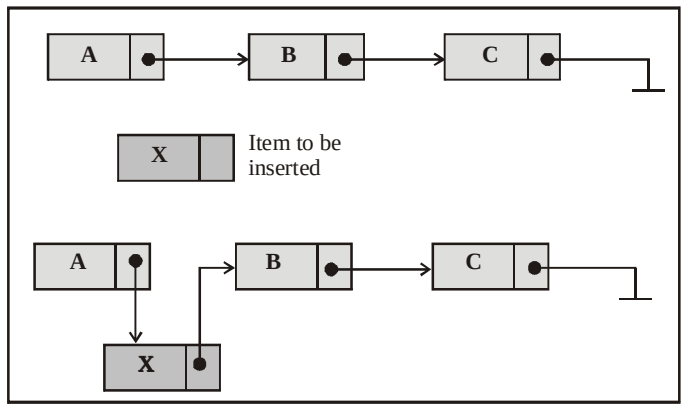
A linked list is a collection of structures ordered not by their physical placement in memory but by logical links that are stored as part of the data in the structure itself. It is not necessary that it should be stored in the adjacent memory locations. Every structure has a data field and an address field. The Address field contains the address of its . Successive elements are connected by pointers. Last element points to NULL. It can grow or shrink in size during execution of a program. It can be made just as long as required.

For creation:

- It is used to create a linked list.
- Once a linked list is created with one mode, insertion operation can be used to add more elements in a mode.

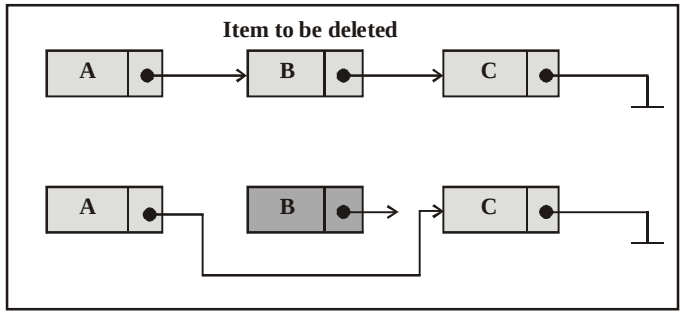
For insertion:

- A record is created holding the new item.
- The next pointer of the new record is set to link it to the item which is to follow it in the list.
- The next pointer of the item which is to precede it must be modified to point to the new item.
- After every push operation the top is incremented by one



For deletion:

– The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.



TYPES OF LINKED LIST

Singly Linked List

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure.

Doubly Linked List

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every node in the doubly linked list has three fields: LeftPointer, RightPointer and DATA. LPoint will point to the node in the left side (or previous node) that is LPoint will hold the address of the previous node. RPoint will point to the node in the right side (or next node) that is RPoint will hold the address of the next node. DATA will store the information of the node.

Circular Linked List

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.

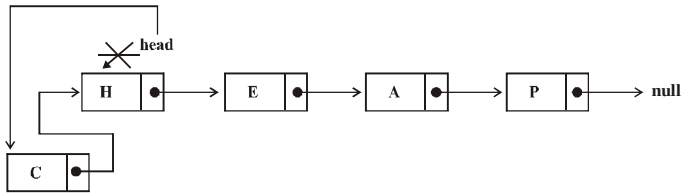
Algorithm to reverse direction (single linked list):

A node data structure has two fields. To keep a variable *first Node*, this always points to the first node in the list, or is null for an empty list.

LINKED LIST OPERATIONS

(a) addFirst

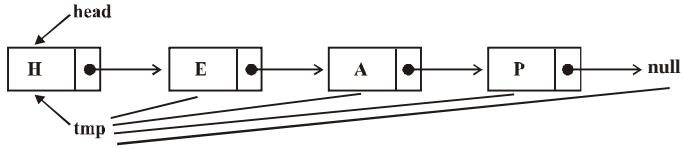
The method creates a node and prepends it at the beginning of the list.




```
public void addFirst(AnyType item)
{
   head = new Node<AnyType>(item, head);
}
```

Traversing

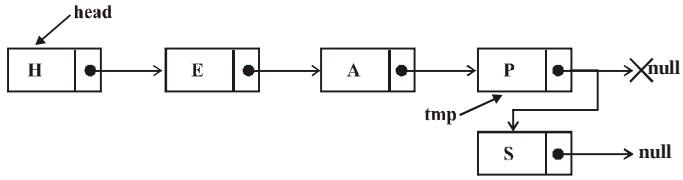
Start with the head and access each node until you reach null. Do not change the head reference.




```
Node tmp = head;
while(tmp != null) tmp = tmp→next in
```

(b) addLast

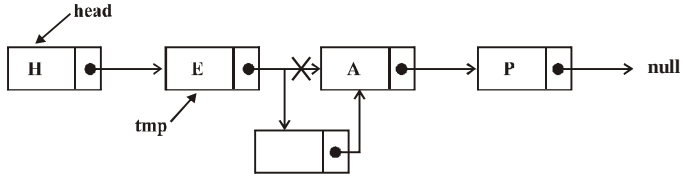
The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node




```
public void addLast(AnyType item)
{ i
   f(head == null) addFirst(item);
    else
 {
   Node<AnyType> tmp = head;
   while(tmp→next in != null) tmp = tmp→next in
    tmp→next in = new Node<AnyType>(item, null);
   }
}
```

Inserting "after"

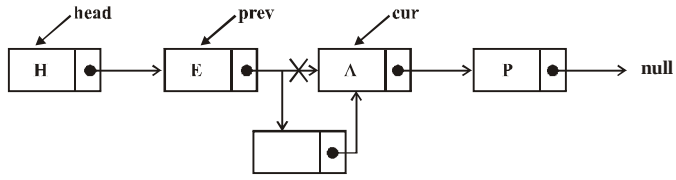
Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":




```
public void insertAfter(AnyType key, AnyType
toInsert)
{ Node<AnyType> tmp = head;
  while(tmp != null && !tmp.data.equals(key)) tmp =
tmp→next in;
  if(tmp != null)
    tmp→next in = new Node<AnyType>(toInsert,
tmp→next in); }
```

Inserting "before"

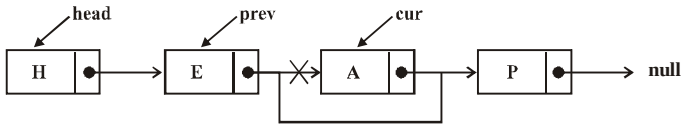
Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references *prev* and *cur*. When we move along the list we shift these two references, keeping *prev* one step before *cur*. We continue until cur reaches the node before which we need to make an insertion. If cur reaches null, we don't insert, otherwise we insert a new node between *prev* and *cur*.

Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



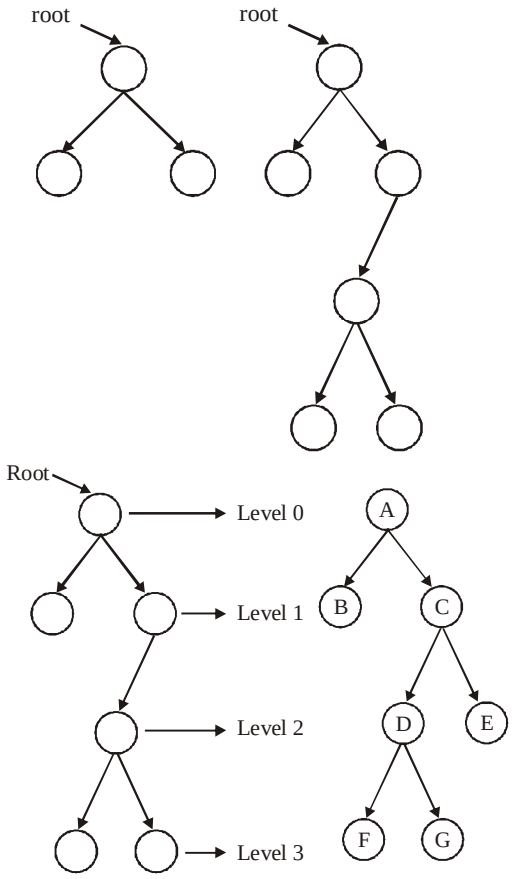
The algorithm is similar to insert "before" algorithm. It is convinient to use two references *prev* and *cur*. When we move along the list we shift these two references, keeping *prev* one step before *cur*. We continue until *cur* reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

```
public void remove(AnyType key)
{
   if(head == null) throw new RuntimeException("cannot
delete");
   if( head.data.equals(key) )
 {
   head = head.next; return; }
    Node<AnyType> cur  = head;
    Node<AnyType> prev = null;
   while(cur != null && !cur.data.equals(key) )
    {
   prev = cur;
      cur = cur.next;
   }
  if(cur == null) throw new RuntimeException("cannot
delete");
    //delete cur node
   prev.next = cur.next;
}
```

TREES

- A **tree** is a data structure that is made of nodes and pointers, much like a linked list. The difference between them lies in how they are organized:
- In a linked list each node is connected to one "successor" node (via next pointer), that is, it is linear.
- In a tree, the nodes can have several next pointers and thus are not linear.
- The top node in the tree is called the **root** and all other nodes branch off from this one.
- Every node in the tree can have some number of children. Each child node can in turn be the parent node to its children and so on.
- A common example of a tree structure is the binary tree. A **binary tree** is a tree that is limited such that each node has only two children.
- If n1 is the root of a binary tree and n2 is the root of its left or right tree, then n1 is the **parent** of n2 and n2 is the **left** or **right child** of n1.
- A node that has no children is called a **leaf**.
- The nodes are **siblings** if they are left and right children of the same parent.
- The level of a node in a binary tree:
  The root of the tree has level 0
  The level of any other node in the tree is one more than the level of its parent.



TYPES OF BINARY TREE

1. Strictly Binary Tree

- If every non leaf node in a binary tree has non empty left and right sub trees, the tree is termed as strictly binary tree
- Every non leaf node has degree 2.
- A strictly binary tree with n leaves has (2n-1) nodes.
- Thus a strictly binary tree has odd number of nodes.

2. Complete Binary Tree

- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l at each level l between 0 and d.
- Thus the total number of nodes in complete binary tree are $2^{d+1}-1$ where leaf nodes are 2^d and non leaf are 2^d-1.
- Because a complete binary tree is also a strictly binary tree , thus if it has n leaves then it has 2n-1 nodes. Also follows from this is the previous assertion that if $n=2^d$ then total nodes are

$$2*2^d-1=2^{d+1}-1$$

3. complete Binary Tree (CBT)

- A binary tree of depth d is an almost complete binary tree if:
  - At any node in the tree with a right descendent at level d, node must have a left son and every left descendent of node is either a leaf at level d or has two sons.
    i.e. the tree must be left filled.

- Note : CBT property says that if there are n nodes in the tree then leaf node have numbers [(n/2)+1][(n/2)+2].....n
- CBT with n leaves has 2n nodes if it is not a SBT
- An CBT which is also an SBT has 2n-1 nodes for n leaves
- Also an CBT is an SBT if the number of node are odd
- An CBT of depth d is intermediate between the complete binary tree of depth, t d-1 that contains 2^{d-1} nodes, and the complete binary tree of depth d, which contains 2^{d+1}-1 nodes
- CBT property says that if there are in node in the tree then leaf note are numbered from $\left(\frac{n}{2+1}, \frac{n}{2+2}....n\right)$

Implementation

- A binary tree has a natural implementation in linked storage. A separate pointer is used to point the tree (e.g. root)
  root = NULL; // empty tree
- Each node of a binary tree has both left and right subtrees which can be reached with pointers:

```
struct tree_node { int data;
struct tree_node *left_child;
struct tree_node *right_child;};
```

TRAVERSAL OF BINARY TREES

Linked lists are traversed from first to last sequentially. However, there is no such natural linear order for the nodes of a tree. Different orderings are possible for traversing a binary tree. Three of these traversal orderings are:

- Preorder traversal (also known as depth – first order)
- Inorder traversal (a.k.a. symmetric order)
- Postorder traversal (a.k.a end order)

These names are chosen according to the step at which the root node is visited.

- With **preorder** traversal the node is visited *before* its left and right subtrees,
- With **inorder** traversal the root is visited *between* the subtrees,
- With **postorder** traversal the root is visited *after* both subtrees.

Binary Tree Traversal >> Preorder Traversal

Preorder traversal of a binary tree consists of following three recursive operations.

a. Visit the root.
b. Traverse the left sub-tree in preorder.
c. Traverse the left sub-tree in preorder.

```
void doPreOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null){
printf("%d", p->key);
doPreOrder(p->left);
doPreOrder(p->right);
}
}
```

Binary Tree Traversal >> Inorder Traversal

Inorder traversal of a binary tree consists of following three recursive operations.

a. Traverse the left sub-tree in inorder.
b. Visit the root.
c. Traverse the left sub-tree in inorder.

```
void doInOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null){
doInOrder(p->left);
printf("%d", p->key);
doInOrder(p->right);
}
}
```

Binary Tree Traversal >> Postorder Traversal

Postorder traversal of a binary tree consists of following three recursive operations.

a. Traverse the left sub-tree in postorder.
b. Traverse the left sub-tree in postorder.
c. Visit the root.

```
void doPostOrder(nodeptr &tree){
nodeptr p = tree;
if(p!= null) {
doPostOrder(p->left);
doPostOrder(p->right);
printf("%d", p->key);
}
}
```

BINARY SEARCH TREE

BST is a binary tree. For each node in a BST, the left subtree is smaller than it; and the right subtree is greater than it. A **binary search tree** (**BST**), sometimes also called an **ordered** or **sorted binary 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 less 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 must each also be a binary search tree.
- There must be no duplicate nodes.

Implementation

We implement a binary search tree using a private inner class BSTNode. In order to support the *binary search tree property*, we require that data stored in each node is Comparable:

```
public class BST <AnyType extends Comparable<AnyType>>
{
   private Node<AnyType> root;

   private class Node<AnyType>
   {
      private AnyType data;
      private Node<AnyType> left, right;

      public Node(AnyType data)
      {
         left = right = null;
         this.data = data;
      }
   }
   ...
```

Insertion

The insertion procedure is quite similar to searching. We start at the root and recursively go down the tree searching for a location in a BST to insert a new node. If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace a NULL reference.

Searching

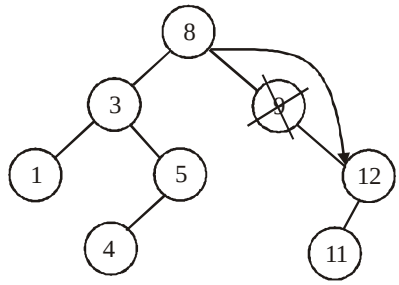
Searching in a BST always starts at the root. We compare a data stored at the root with the key we are searching for (let us call it as **to Search**). If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise - to the right child. The recursive structure of a BST yields a recursive algorithm.

Deletion

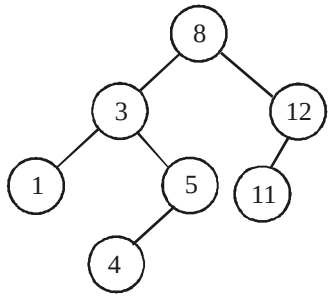
Deletion is somewhat more tricky than insertion. There are several cases to consider. A node to be deleted (let us call it as **to Delete**)

is not in a tree; is a leaf;
has only one child; has two children.

If **to Delete** is not in the tree, there is nothing to delete. If **to Delete** node has only one child the procedure of deletion is identical to deleting a node from a linked list - we just bypass that node being deleted

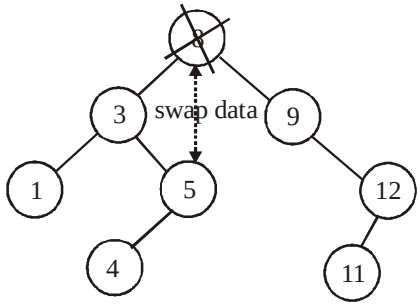


before deletion

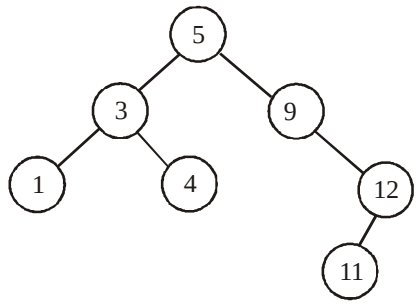


after deletion

Deletion of an internal node with two children is less straightforward. If we delete such a node, we split a tree into two subtrees and therefore, some children of the internal node won't be accessible after deletion. In the picture below we delete 8:



before deletion



after deletion

Deletion starategy is the following: replace the node being deleted with the largest node in the left subtree and then delete that largest node. By symmetry, the node being deleted can be swapped with the smallest node is the right subtree.

Non-Recursive Traversals

Depth-first traversals can be easily implemented recursively.A non-recursive implementation is a bit more difficult. In this section we implement a pre-order traversal as a tree iterator

```
public Iterator<AnyType> iterator()
{
 return new PreOrderIterator();
}
```

where the PreOrderIterator class is implemented as an inner private class of the BST class

```
private class PreOrderIterator implements
Iterator<AnyType>
{
 ...
}
```

THREADED BINARY TREE

Traversing a binary tree is a common operation and it would be helpful to find more efficient method for implementing the traversal. Moreover, half of the entries in the Lchild and Rchild field will contain NULL pointer. These fields may be used more efficiently by replacing the NULL entries by special pointers which point to nodes higher in the tree. Such types of special pointers are called threads and binary three with such pointers are called threaded binary tree.

B+ tree : B+ tree is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.

SPANNING TREE

A spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertexlies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T.

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

GRAPH

A graph consists of set of vertices and a set of edges. Graphs representations have found application in almost all subjects like geography, engineering and solving games and puzzles. A vertex can also have a weight, sometimes also called a cost.

A graph G consist of

1. Set of vertices V (called nodes), (V = {v1, v2, v3, v4.....}) and
2. Set of edges E (i.e., E {e1, e2, e3.....cm}

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge 'e' in E is identified with a unique pair (a, b) of nodes in V, denoted by e = [a, b]

A graph whose pairs are ordered is called a directed graph, or just a digraph.

If a graph is not ordered, it is called an unordered graph, or just a graph.

SHORTEST PATH

A path from a source vertex a to b is said to be shortest path if there is no other path from a to b with lower weights. There are many instances, to find the shortest path for travelling from one place to another. That is to find which route can reach as quick as possible or a route for which the travelling cost in minimum.

HASHING

Hashing for storing data in such a way the data can be inserted and retrieved very quickly. Hashing uses a data structure called hash table. The searching time of the each searching technique, depends ons the comparison i.e., n comparisons required for an array A with n elements. To increase the efficiency, i.e., to reduce the searching time, we need to avoid unnecessary comparisons.

HASH FUNCTION

The basic idea of hash function is the transformation of the key into the corresponding location in the hash table. A Hash function H can be defined as a function that takes key as input and transforms it into a hash table index. Hash functions are of two types:

1. Distribution – Independent function
2. Distribution – Dependent function

We are dealing with Distribution - Independent function. Following are the most popular Distribution - Independent has functions:

1. Division method
2. Mid square method
3. Folding method.

Division Method : TABLE is an array of database file where the employee details are stored. Choose a number m, which is larger than the number of keys k. i.e., m is greater than the total number of records in the TABLE. The number m is usually chosen to be prime number to minimize the collision. The hash function H is defined by

$$H(k) = k \pmod m$$

Where H (k) is the hash address (or index of the array) and here k (mod m) means the remainder when k is divided by m.

Mid Square Method : The key k is squared. Then the hash function H is defined by

$$H(k) = k^2 = 1$$

Where 1 is obtained by digits from both the end of k^2 starting form left. Same number of digits must be used for all of the keys.

Folding Method : The key K, k_1, k_2.... k_r is partitioned into number of parts. The parts have same number of digits as the required hash address, except possibly for the last part. Then the parts are added together, ignoring the last carry. That is

$$H(k) = k_1 + k_2 + ...+ k_r$$

SORTING

Sorting is used to arrange names and numbers in meaningful ways. Let A be a list of n element A1, A2,..... An in memory. Sorting is list A refers to the operation of rearranging the contents of A so that they are in increasing (or decreasing) order (numerically or lexicographically): A1 < A2 < A3 < ..... < An.

Sorting can be performed in many ways. Over a time several methods are being developed to sort data(s). Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, Binary sort, Shell sort and Radix sort.

- **BUBBLE SORT**

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Time Complexity

The time complexity for bubble sort is calculated in terms of the number of comparisons f (n) (or of number of loops); here two loops (outer loop and inner loop) iterates (or repeated) the comparisons. The number of times the outer loop iterates is determined by the number of elements in the list which is asked to sort (say it is n). The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop

$$f(n) = (n-1) + (n-2) + ... + 2 + 1$$
$$= n(n-1) = O(n^2).$$

- **SELECTION SORT**

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Time Complexity

Time complexity of a selection sort is calculated in terms of the number of comparisons f (n). In the first pass it makes n – 1 comparisons; the second pass makes n – 2 comparisons and so on. The outer for loop iterates for (n – 1) times. But the inner loop iterates for n*(n – 1) tims to complete the sorting.

$$f(n) = (n-1) + (n-2) + ... + 2 + 1$$
$$= (n(n-1)/2$$
$$= O(n^2)$$

- **INSERTION SORT**

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first, if the first element is greater than second; place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place. And place the third value to first place and so on.

Time Complexity

In the insertion sort algorithm (n – 1) times the loop will execute for comparisons and interchanging the numbers. The inner while loop iterates maximum of ((n – 1) × (n – 1))/2 times of computing the sorting. Worst case time complexity : O (n^2)
Best case time complexity = O (n)
Average time complexity = O (n^2)
Space complexity = O (1)

- **SHELL SORT**

Shell Sort is introduced to improve the efficiency of simple insertion sort. Shell Sort is also called diminishing increment sort. IN this method, sub-array, contain kth element of the original array, are sorted. The complexity of shell sort is f (n) = O(n (log n)^2).

- **QUICK SORT**

Quick sort is one of the widely used sorting techniques and it is also called the partition exchange sort. Quick sort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases :

- The partition phase and
- The Sort phase.
- Complexity analysis of quick sort
- Worst case time complexity : O(n^2)
- Best case time complexity : O(n log n)
- Average time complexity : O (n log n)
- Space complexity : O (n log n)

- **MERGE SORT**

Merge sort works using the principle that if you have two sorted lists, you can merge them together to form another sorted list. Consequently, sorting a large list can be thought of as a problem of sorting two smaller list and then merging those two lists together.

- Worst case time complexity : O (n log n)
- Best case time complexity : O(n log n)
- Average time complexity : O (n log n)
- Space complexity : O (n)

- **HEAP SORT**

A heap is a partially sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less than either of its children. Additionally, a heap is a “complete tree” — a complete tree is one in which there are no gaps between leaves. For instance, a tree with a root node that has only one child must have its child as the left node. More precisely, a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.

- Worst case time complexity : O(n log n)
- Best case time complexity : O(n log n)
- Average time complexity : O (n log n)
- Space complexity : O (n)