gradeup

Data Structure Part-2

Prep Smart. Stay Safe. go Gradeup

Data Structure Part-2

Content:

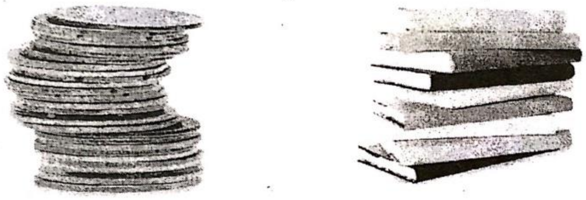
STACK

PRESENTATION OF CONTENTS

Stacks

Stacks and Queue are two data structures that allow insertions and deletions operations only at the beginning or the end of the list, not in the middle.

A stack is a linear data structure in which items may by added or removed only at one end named as the top of the stack. Everyday example of such a structure are very common viz. a stack of dishes, a stack of books, a stack of coins and a stack of cloths etc. as shown in below figure.

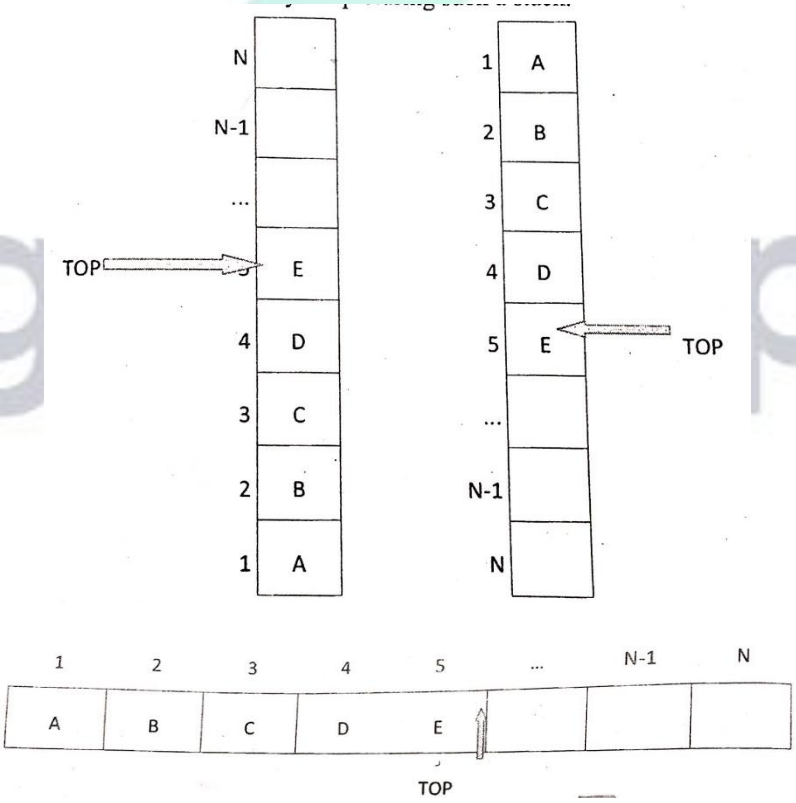


Stacks are also called list-in first-out (LIFO) lists. This means, that elements which are inserted list will be removed first. Other names generally used for stacks are “piles” and “push-down lists”. Stack has many important applications in the field of computer science. Terminology which is used for two basic operation associated with stacks are:

a. “Push”: It is the term used to insert an element into a stack.
b. “Pop”: It is the term used to delete an element from a stack.

Example: Suppose that 5 elements are pushed onto an empty stack A, B, C, D, E

Below figure shows three ways of picturing such a stack



Operations on Stacks

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) and item from a stack are implemented by the procedures called PUSH and POP respectively.

In the implementation of these operations, TOP and MAX are assumed as global variables; hence these are not required as arguments in the algorithms, which in turn may be named as PUSH (STACK, ITEM) and POP (STACK, ITEM) respectively.

Implementation of Stacks Arrays

Insertion: When we are adding a new element, first, we must test whether there is a free space in the stack for the new item; if not, then we have the condition known as overflow. If this condition is not there, then the value of TOP is changed before the insertion in PUSH. After changing the value of TOP, insertion is done.

Deletion: In executing the procedure POP, we must first test whether there is an element in the stack to be deleted; if not; then we have the condition known as underflow. The item to be deleted is first stored in some variable, then the value of TOP is changed after the deletion in POP.

It would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed.

Though array and stack are totally different data structures, a array can be used to store the elements of a stack. One can declare the array with the maximum size large enough to manage a stack.

Applications

The stacks are used in numerous applications and some of them are as follows:

- ✓ Arithmetic expression evaluation
- ✓ Undo operation of a document editor or a similar environment
- ✓ Implementation of recursive procedures
- ✓ Backtracking
- ✓ Keeping track of page i.e. visited history of a web user
- ✓ Quicksort

We will be discussing some of these applications in detail.

Arithmetic Expression: Polish Notation

Let AE be an arithmetic expression involving constants and operations. This section gives an algorithm which finds the value of AE by using Reverse Polish (Postfix) Notation. We will see now, that a stack is an essential tool in this algorithm. We will be using the following levels of precedence in our AE.

Highest	:	Parentheses()
Next Highest	:	Exponentiation (^)
Next highest	:	Multiplication (*) and division (/)
Lowest	:	Addition (+) and subtraction (-)

Example: Let us evaluate the following parenthesis - free arithmetic expression:

$$2 \wedge 3 + 5 * 2 \wedge 2 - 12 / 6$$

First we evaluate the exponentiations to obtain

$$8 + 5 * 4 - 12 / 6$$

Then we evaluate the division and multiplication to obtain 8 + 20 - 2. Lastly, we evaluate the addition and subtraction to obtain the final result, 26. We Observe that the expression is traversed three times, each time corresponding to a level of precedence of the operations.

Now if the expression is written using parentheses:

$$2 \wedge 3 + (5 * 2) \wedge 2 - 12 / 6$$

Now first we evaluate the parentheses to obtain

$$2 \wedge 3 + (10) \wedge 2 - 12 / 6$$

Then we evaluate the exponentiation to obtain 8 + 100 – 2. Last, we evaluate the addition and subtraction to obtain the final results as 106. It can easily be observed that by inserting a parentheses, the result has changed drastically. To avoid such confusion and problem, computer first convert the expression as a parentheses free expression, which may be a postfix expression or prefix expression and then evaluates it. Therefore, now we will be studying the concept of these notations and use of stack in them.

Polish Notation:

In Mathematics, the operator symbol is placed between its two operands. For example,

$$A + B \qquad C - D \qquad E * F \qquad G / H$$

This is called infix notation. With this notation, we must distinguish between

$$(A + B)*C \qquad \text{and} \qquad A + (B *C)$$

By using either parentheses or operator – precedence convention such as the usual precedence levels discussed above. Accordingly, the order of the operands and operators in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

We can convert an infix expression into polish notation as follows:

(A + B) ∗ C = [+AB] ∗ C = ∗ + ABC

A + (B ∗ C) = A + [∗BC] = A∗BC

(A + B) / (C - D) = [+AB] / [- CD] = /+AB - CD

Fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operands and operations in the expression. No need of parentheses when writing expressions in Polish notation.

But when an expression in written in a program in computer, it is infix expression that is used. Computer usually evaluates an arithmetic expression which is written in infix notation in two steps. Firstly, it converts the expression to postfix notation, and after that it evaluates the postfix expression. The postfix notation is also known as Reverse Polish Notation (RPN). Some of the example to convert an infix expression into postfix expression are as follows:

(A + B) ∗ C = [+AB] ∗ C = ∗ + AB+C∗

A + (B ∗ C) = A + [∗BC] = ABC∗+

(A + B) / (C - D) = [+AB] / [- CD] = AB + CD-/

Converting an Infix Expressions into Postfix Expressions

Let AE be an arithmetic expression written in infix notation. The following algorithm converts an infix expression AE into its equivalent postfix expression PE. Algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression PE will be constructed from left to right using the operands from AE and the operators which are removed from STACK. One begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of AE. The algorithm is completed when STACK is empty.

Example: let us discuss the algorithm with the help of an example. Consider the following arithmetic infix expression

AE: A + (B - C * (D / E ^ F))

We will convert AE into its equivalent postfix expression PE using above algorithm. First we push "(" onto STACK, and then we add")" to the end of AE to obtain:

AE: A + (B - C * (D / E ^ F)))

Below table shows the status of STACK and of the string PE as each element of AE is scanned.

Symbol Scanned		Stack	Expression PE
1	A	(	A
2	+	(+	A
3	(	(+(	A
4	B	(+(	A B
5	-	(+(-	A B
6	C	(+(-	A B C
7	*	(+(-*	A B C
8	(	(+(-*(	A B C
9	D	(+(-*(	A B C D
10	/	(+(-*(/	A B C D
11	E	(+(-*(/	A B C D E
12	^	(+(-*(/^	A B C D E
13	F	(+(-*(/^	A B C D E F
14	)	(+(-*	A B C D E F ^/
15	)	(+	A B C D E F^/*-
16	)		A B C D E F ^ / * - +

Example: Let us consider the following arithmetic expression AE written in infix notation:

AE: 1 + 2 - 3 ∗ (4/2)

The equivalent postfix expression PE becomes:

PE: 1, 2, +, 3, 4, 2, /, ∗, -

We will now evaluate PE using EVAL algorithm. First we add a sentinel right parenthesis at the end of PE to obtain

P: 1,	2,	+,	3,	4,	2,	/,	∗,	-,	)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

Below figure shows the contents of STACK as each element of PE is scanned. Final number in STACK, -3, which is assigned to the RESULT when the sentinel “(” is scanned, is the value of PE.

Symbol Scanned		**Stack**
1	1	1
2	2	1, 2
3	+	3
4	3	3, 3
5	4	3, 3, 4
6	2	3, 3, 4, 2
7	/	3, 3, 2
8	*	3, 6
9	-	-3
10	)	

QUEUE

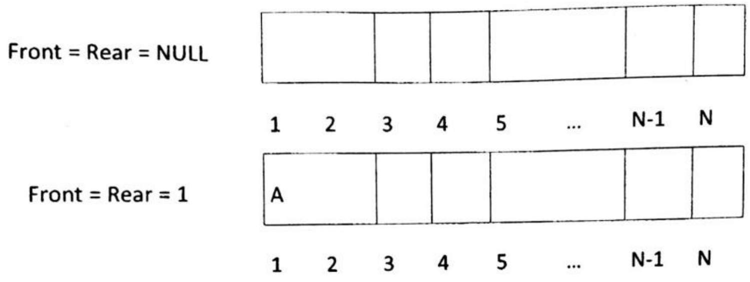
Queues are data structures that allow insertions and deletions operations only at the beginning or the end of the list, not in the middle.

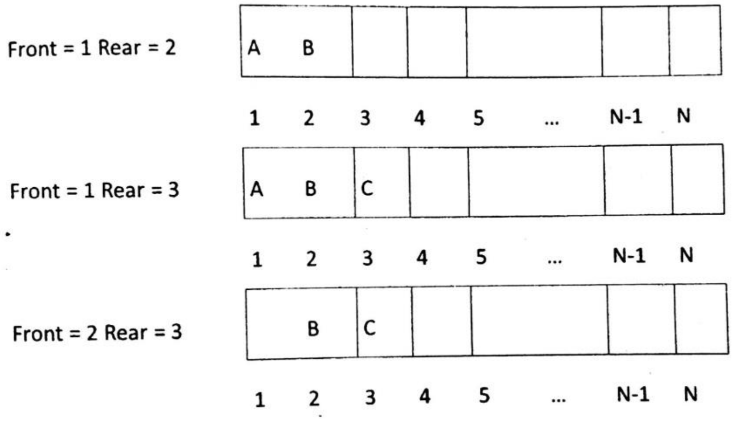
It is a linear structure in which element may be inserted at one end which is called the rear, and deleted at the other end called the front. In below figures a queue of people waiting for their turn. Queues are also called First-In-First-Out (FIFO) list. An important example of a queue is computer science occurs in a timesharing system, in which programs with the same priority from a queue while waiting to be executed.

Computer Representation of Queue

Queues may be represented in the computer in two ways i.e. by mean of linear arrays and one-way linked lists. First we will be considering representation of queues with the help of arrays

Queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front(firs) element of the queue; and REAR(last), containing the location of the rear element of the queue. The condition FRONT = 0 will indicate that the queue is empty.

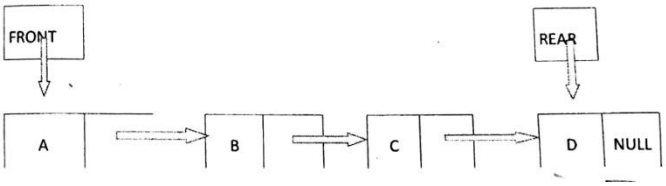




As indicated in the above figure, when there is no element, then front and rear both are equal to NULL. If an element is inserted, the rear changes to 2, and after another insertion rear becomes 3. Now if an element is deleted than the front changes to 2.

Now we will be considering representation of queues with the help of linked lists.

In this case the representation is vary much similar to the one-way linked list, except that the address of the starting node is saved as FRONT and the address of the last node is termed as REAR. Also this linked list is restricted linked – list in the sense that insertions can take place only at the REAR and deletion can only take place at FRONT. The linked representation is shown in below figure.



OPERATIONS ON QUEUE

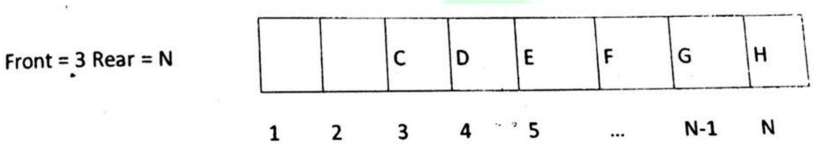
Insertion

Above figure indicated the some of the ways in which elements will be inserted in the queue and the way new elements will be deleted from the queue. Whenever element is added to queue, the value of REAR is increased by 1; this can be implemented by assignment.

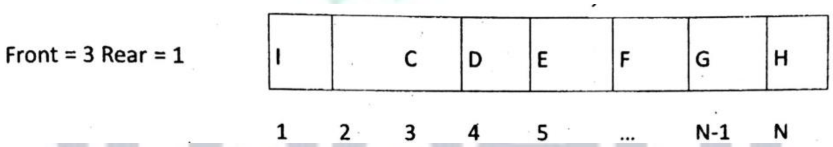
REAR = REAR + 1

Generally QUEUE is maintains as circular array in computer science, that is, QUEUE[1] comes after QUEUE[N] in the array. With this assumption, if we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Instead of increasing REAR to N + 1, we reset REAR = 1 and then assign,

QUEUE [REAR] = ITEM



This operation can be shown by below figure

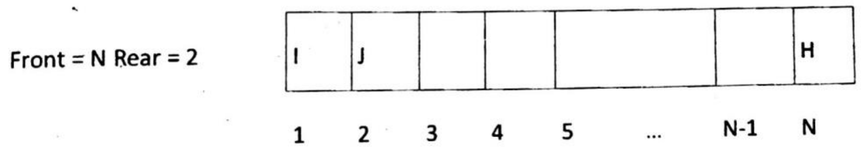


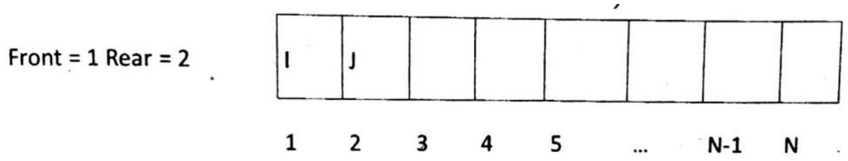
Deletion

As shown in above figure, whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

FRONT = FRONT + 1

As QUEUE is assumed to be circular, that is, that QUEUE[1] comes after queue[N] in the array. With this assumption, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1 as shown in below figure.





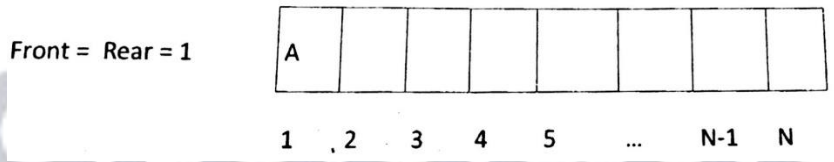
Suppose that the queue contains only one element, i.e., suppose that

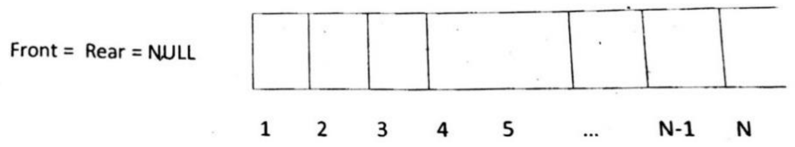
FRONT= REAR = 1

And suppose that the element is deleted. Then we assign

FRONT = NUL and REAR = NULL

To indicate that the queue is empty and this operation can be depicted by below figure





Dequeue

It is a linear list in which elements can be added or removed at either end but not in the middle. The term deque refers to the name double ended queue

There are basically two variations of a deque – namely, an input – restricted deque and an output restricted deque – which are intermediate between a deque and a queue. Input restricted deque is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an output –

restricted deque is a deque, which allows deletions at only one end of the list buy allows insertions at both end of the list.

PRIORITY QUEUES

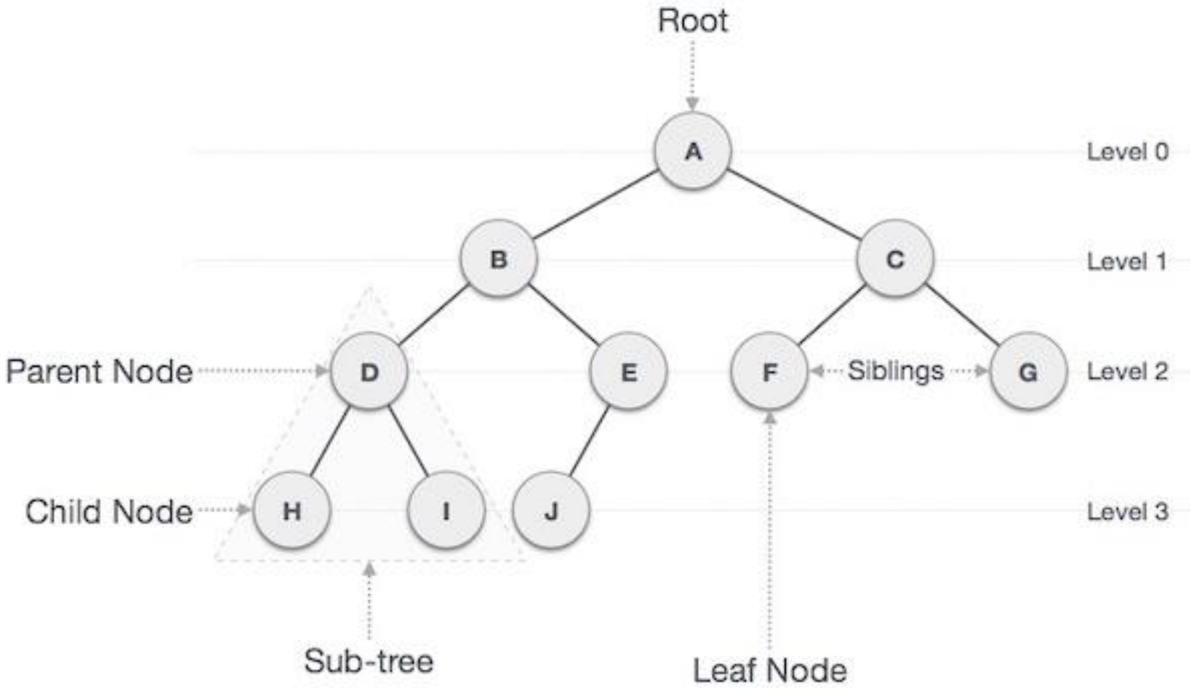
It is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

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

There are various ways for maintaining a priority queue in computer memory. The two main ways are using one-way list and array representation.

Tree:

Tree represents the nodes connected by edges. Binary Tree is a special data structure used for the data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has benefits of both a linked list and an ordered array as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



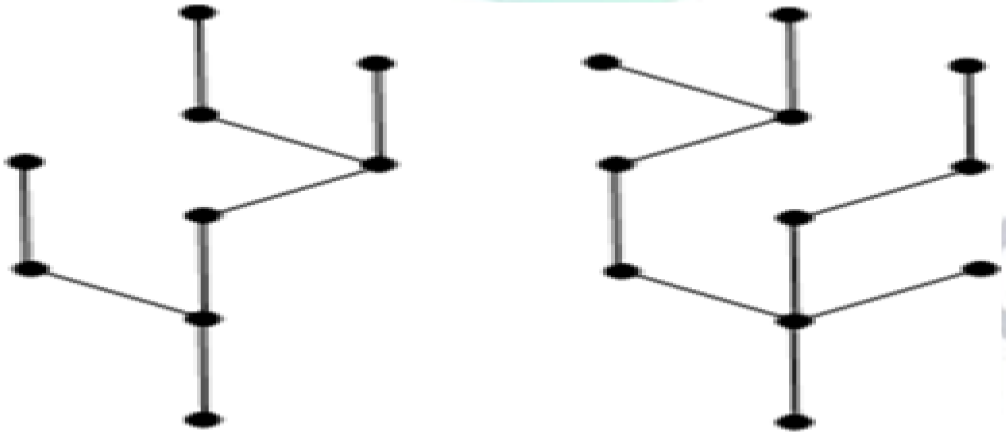
Important Terms:

- **Path** − It refers to the sequence of nodes along the edges of a tree.
- **Root** − Node which is at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except root node has one edge upward to a node called parent.
- **Child** − Node below a given node connected by its edge downward is called its child node.
- **Leaf** − Node which does not have any child node is called the leaf node.
- **Subtree** - It is represents the descendants of a node.
- **Visiting** − Visiting refers to checking value of a node when control is on the node.
- **Traversing** − It means passing through nodes in a specific order.

- **Levels** - It is represents the generation of a node. If root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − It represents a value of a node based on which a search operation is to be carried out for a node.

Forest:

In graph theory, a **forest** is **an undirected, disconnected, acyclic graph**. In other words, a disjoint collection of trees is known as forest. Each component of a forest is tree.

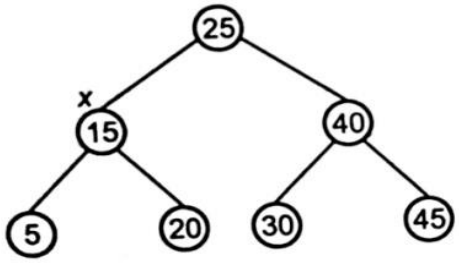


BINARY SEARCH TREE

BINARY-SEARCH-TREE PROPERTY

Let x be a node in a binary search tree.

- ➢ If y is a node in the left sub tree of x, then key [y] ≤ key [x]
- ➢ If y is a node in the right sub tree of x, then key [x] ≤ key [y]



In this tree key [x] = 15

- If y is a node in the left sub tree of x then key [y] = 5.

i.e., key [y] ≤ key [x]

- If y is a node in the right sub tree of x then key [y] = 20

i.e., key [x] ≤ key [y]

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm derives its name from the fact that the key of the root of a sub tree is printed between the values in its left sub tree and those in its right sub tree. (similarly, a pre order tree walk prints root before the values in either sub tree, and a post order tree walk prints root after the values in its sub trees.

BINARY-SEARCH-TREE PROPERTY vs HEAP PROPERTY

In a heap, a nodes key is greater than equal to both of its children's keys. In binary search tree, a node's key is greater than or equal to its child's key but less than or equal to right child's key. Furthermore, this applies to entire sub tree is the binary search tree case. It is very important to node that the heap property does not help print the nodes in sorted order because this property does not tell us in which sub tree the next item is. If the heap property could be used to print the keys in sorted in O(n) times, this would contradict our known lower bound on comparison sorting.

Last station implies that since sorting n elements takes Ώ (n lg n) time in the worst case in the comparison-based algorithm for constructing a Binary Search Tree from arbitrary list n elements takes Ώ(n lg n) time in the worst case.

We can show the validity of this argument (in case you are thinking of beating Ώ (n lg n) bound)as follows: let c(n) be the worst-case running time for constructing binary tree of a set of n elements. Given an n-node BST, the INORDER walk in the tree output the keys in sorted order (shown above).since the worst-case running time of any computation based sorting algorithm is Ώ(n lg n), we have

$$c(n) + O(n) = \text{Ώ}(n \lg n)$$

therefore, $$c(n) = \text{Ώ}(n \lg n)$$

QUERYING A BINARY SEARCH TREE

The most common operation performed on a binary search tree is searching for a key stored in the tree, beside the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. These operations run in O(h) times where h is the height of the tree i.e., h is the number of links root node to the deepest node.

SEARCHING

TREE-SEARCH (x, k) algorithm searches the tree root at x for a node whose key value equals k. it returns a pointer to the node if it exists otherwise NIL.

This algorithm runs in O(h) time where h is the height of the tree.

MINIMUM AND MAXIMUM

An element in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered. The given following procedure returns a pointer to the maximum element in the sub tree rooted at a given node x.

TREE-MINIMUM (x)

```
While left[x] ≠ NIL
    do x ← left [x]
Return x
```

TREE-MAXIMUM (x)

```
While right [x] ≠ NIL
    do x ← right [x]
Return x
```

Both of these procedure runs in O(h) time on a tree of height h.

SUCCESSOR AND PREDECESSOR

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than key[x]. the structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the free.

TREE SUCCESSOR (x)

```
If right [x] ≠ NIL
    Then return TREE-MINIMUM (right[x])
```

```
3.  y ← p [x]
4.  while y ≠ NIL and x = right [y]
5.      do x ← y
6.      y ← p [y]
7.  return y
```

the code for TREE-SUCCESSOR is broken into two cases. If the right sub tree of node x is nonempty, then the successor of x is just the left-most node in the right sub tree, which is found in line 2 by calling TREE-MINIMUM (right [x]). On the other hand, if right sub tree of the node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. to find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; line 3-7 of TREE-SUCCESSOR accomplish this.

Running time of TREE-SUCCESSOR on a tree of height h is O(h).

gradeup
SUPER
Select your exam
Upcoming
Banking
SSC
Teaching
JEE
GATE
Medical
UPSC
Defence
NET
SBI Clerk
JEE Main 2020
CTET
RRB NTPC
NEET 2020
NIELIT
SSC CPO
CLAT
UPPSC PCS
SSC CGL
KVS
CDS
SSC CHSL
UPTET
SEBI Grade A
Get Started with Gradeup!
Sign in with Google
OR
Continue
Continue with Email