

Data Structures

At the heart of virtually every computer program are its algorithms and its data structures. It is hard to separate these two items, for data structures are meaningless without algorithms to create and manipulate them, and algorithms are usually trivial unless there are data structures on which to operate.

This category concentrates on four of the most basic structures: stacks, queues, binary search trees, and priority queries. Questions will cover these data structures and implicit algorithms, not on implementation language details. A *stack* is usually used to save information that will need to be processed later. Items are processed in a “last-in, first-out” (LIFO) order. A *queue* is usually used to process items in the order in which requests are generated; a new item is not processed until all items currently on the queue are processed. This is also known as “first-in, first-out” (FIFO) order. A *binary search tree* is used when one is storing a set of items and needs to be able to efficiently process the operations of insertion, deletion and query (i.e. find out if a particular item is part of the set and if not, which item in the set is close to the item in question). A *priority queue* is used like a binary search tree, except one cannot delete an arbitrary item, nor can one make an arbitrary query. One can only delete the smallest element of the set, and can only find out what is the smallest element of the set.

A stack supports two operations: PUSH and POP. A command of the form “PUSH(A)” puts the key A at the top of the stack; the command “POP (X)” removes the top item from the stack and stores its value into variable X. If the stack was empty (because nothing had ever been pushed on it, or if all elements has been popped off of it), then X is given the special value of NIL. An analogy to this is a stack of books on a desk: a new book is placed on the top of the stack (pushed) and a book is removed from the top also (popped). Some textbooks call this data structure a “push-down stack” or a “LIFO stack”.

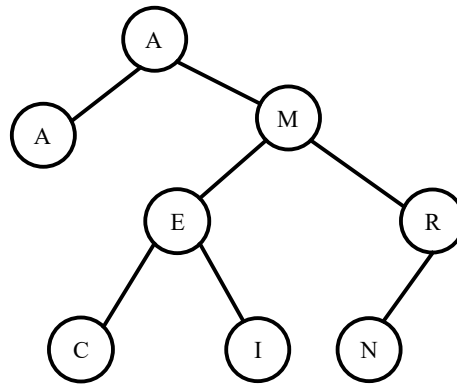
Queues operate just like stacks, except items are removed from the bottom instead of the top. A good physical analogy of this is the way a train conductor or newspaper boy uses a coin machine to give change: new coins are added to the tops of the piles, and change is given from the bottom of each. Some textbooks refer to this data structure as a “FIFO stack”.

Consider the following sequence of 14 operations:

PUSH(A), PUSH(M), PUSH(E), POP(X), PUSH(R),
POP(X), PUSH(I), POP(X), POP(X), POP(X),
POP(X), PUSH(C), PUSH(A), PUSH(N)

If these operations are applied to a stack, then the values of the pops are: E, R, I, M, A and NIL. After all of the operations, there are three items still on the stack: the N is at the top (it will be the next to be popped, if nothing else is pushed before the pop command), and C is at the bottom. If, instead of using a stack we used a queue, then the values popped would be: A, M, E, R, I and NIL. There would be three items still on the queue: N at the top and C on the bottom. Since items are removed from the bottom of a queue, C would be the next item to be popped regardless of any additional pushes.

A binary search tree is composed of nodes having three parts: information (or a key), a pointer to a left child, and a pointer to a right child. It has the property that the key at every node is always greater than or equal to the key of its left child, and less than the key of its right child. The following tree is built from the keys A, M, E, R, I, C, A, N in that order:



The *root* of the resulting tree is the node containing the key A; note that duplicate keys are inserted into the tree as if they were less than their equal key. The tree has a depth (sometimes called height) of 3 because the deepest node is 3 nodes below the root. Nodes with no children are called *leaf* nodes; there are four of them in the tree: A, C, I and N. An *external node* is the name given to a place where a new node could be attached to the tree. In the final tree above, there are 9 external nodes; these are not drawn. The tree has an *internal path length* of 15: the sum of the depths of all nodes. It has an *external path length* of 31: the sum of the depths of all external nodes. To insert the N (the last key inserted), 3 *comparisons* were needed: against the root A, the M and the R.

To perform an “inorder” traversal of the tree, recursively traverse the tree by first visiting the left child, then the root, then the right child. In the tree above, the nodes are visited in the following order: A, A, C, E, I, M, N and R. A “preorder” travel (root, left, right) visits in the following order: A, A, M, E, C, I, R and N. A “postorder” traversal (left, right, root) is: A, C, I, E, N, R, M, A. Inorder traversals are typically used to list the contents of the tree in order.

Binary search trees can support the operations: insert, delete and search. Moreover, it handles the operations efficiently: in a tree with, say, 1 million items, one can search for a particular value in about $\log_2 1000000 \approx 20$ steps. Items can be inserted or deleted in about as many steps, too. However, binary search trees can become unbalanced, if the keys being inserted are not pretty random. For example, consider the binary search tree resulting from inserting the keys A, E, I, O, U, Y. Sophisticated techniques are available to maintain balanced trees. Binary search trees are “dynamic” data structures: they can support an unlimited number of operations, and in any order.

To search for a node in a binary tree, the following algorithm (in pseudo-code) is used:

```

p = root
found = FALSE
loop while (p ≠ NIL) and (not found)
  if (x < p's key) then p = p's left child
  else if (x > p's key) then p = p's right child
  else found = TRUE
repeat
  
```

Deleting from a binary search tree is a bit more complicated. The algorithm we'll use is as follows:

```


$p$  = node to delete  

 $f$  = father of  $p$   

if ( $p$  has no children) then  

    delete  $p$   

else if ( $p$  has one child) then  

    make  $p$ 's child become  $f$ 's child  

    delete  $p$   

else ( $p$  has two children)  

     $l$  =  $p$ 's left child (it might also have children)  

     $r$  =  $p$ 's right child (it might also have children)  

    make  $l$  become  $f$ 's child instead of  $p$   

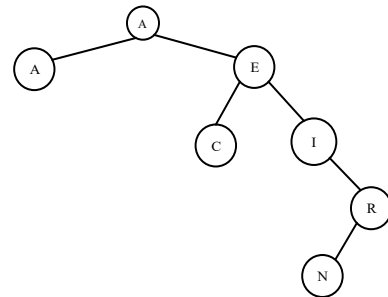
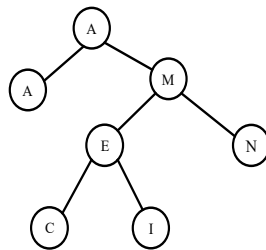
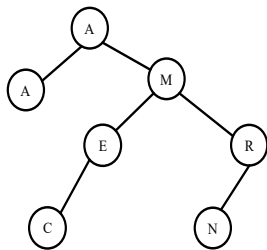
    stick  $r$  onto the  $l$  tree  

    delete  $p$   

end


```

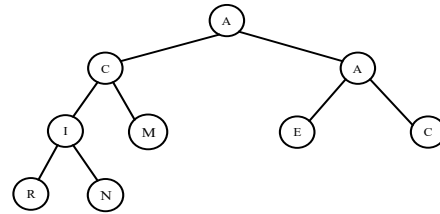
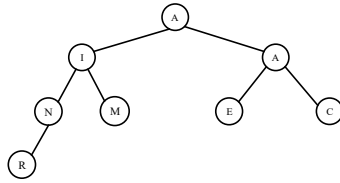
The following three diagrams illustrate the algorithm using the tree above. At the left, we delete I (no children); in the middle, we delete the R (one child); and at the right, we delete the M (two children).



A priority queue is quite similar to a binary search tree, but one can only delete the smallest item and “search” for the smallest. These operations can be done in a guaranteed time proportional to the log of the number of items. One popular way to implement a priority queue is using a “heap” data structure. A heap uses a binary tree (that is, a tree with two children) and maintains the following two properties: every node is larger than its two children (nothing is said about the relative magnitude of the two children), and the resulting tree contains no “holes”. That is, all levels of the tree are completely filled, except the bottom level, which is filled in from the left to the right.

You may want to stop for a moment to think about how you might make an efficient implementation of a priority queue.

The algorithm for insertion is not too difficult: put the new node at the bottom of the tree and then go up the tree, making exchanges with its parent, until the tree is valid. Consider inserting C into the following heap that has been built by inserting A, M, E, R, I, C, A, N:

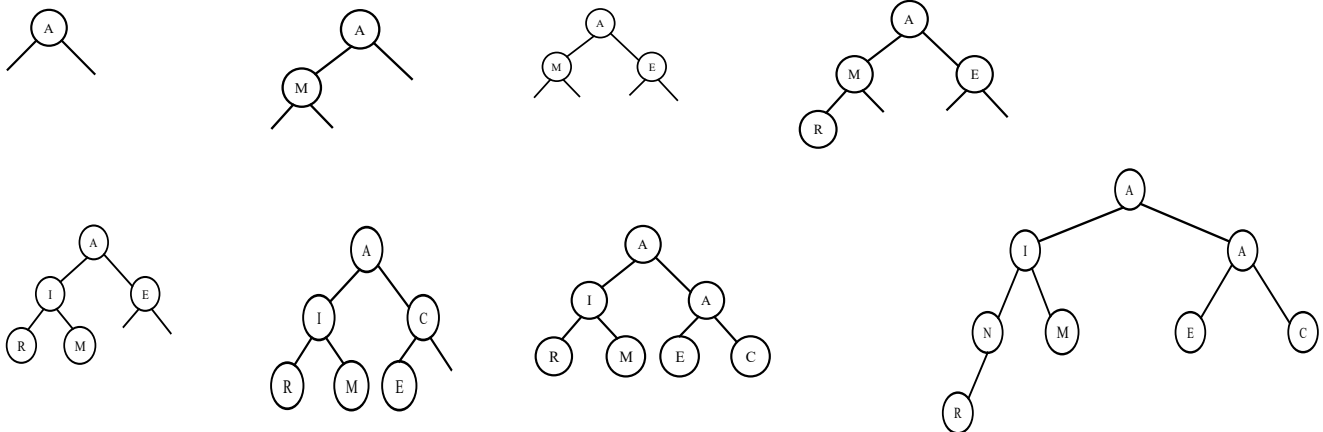


The smallest value is always the root. To delete it (and one can only delete the smallest value), one replaces it with the bottom-most and right-most element, and then walks down the tree making exchanges with the child in order to insure that the tree is valid. The following pseudo-code formalizes this notion:

```

b = bottom-most and right-most element
p = root of tree
p's key = b's key
delete b
loop while (p is larger than either child)
    exchange p with smaller child
    p = smaller child
repeat
  
```

BUILDING A HEAP FROM A, M, E, R, I, C, A, N:



References

Amberg, Wayne. *Data Structures from Arrays to Priority Queues*, Wadsworth (1985).

Bentley, Jon. "Thanks, Heaps" in *Programming Pearls*, *Communications of the ACM*, Vol . 28, No. 3, March 1985.

Sedgewick, Robert. *Algorithms*, Addison-Wesley (1983), Chapters 11 and 14.

Wirth, Niklaus. "Data Structures and Algorithms" in *Scientific American*, September 1984, pp. 60-79.

Sample Problems

The statements $push(p, q)$ and $pop(p, q)$ handle two parallel stacks. The push puts p on the top of one stack and the q on the top of the second stack. The pop command does a pop of the first stack and puts the value into p , and a pop of the second stack and puts the value into q . For example,

```
push(4, 6)
push(2, 5)
pop(a, b)
pop(c, d)
```

would result in $a=2$, $b=5$, $c=4$, $d=6$.

Consider the following operations on an initially empty stack:

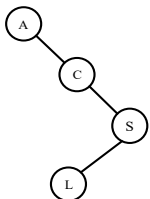
```
push(40, 10)
push(35, 3)
push(12, 20)
pop(a, b)
pop(a, b)
push(10, 23)
```

What would be the next item removed from the “first” stack?

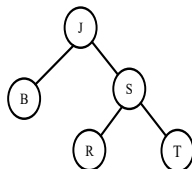
The first pop puts $a=12$ and $b=20$; the second pop set $a=35$ and $b=3$. The $push(10, 23)$ would cause the next pop to set $a=10$ and $b=23$. The wording of the problem indicates that a is the “first” stack and b the “second”.

Which of the following binary trees are valid binary search trees? (Empty nodes are not drawn.)

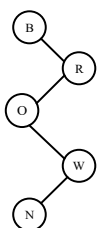
A



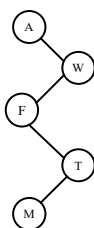
B



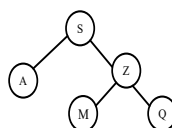
C



D

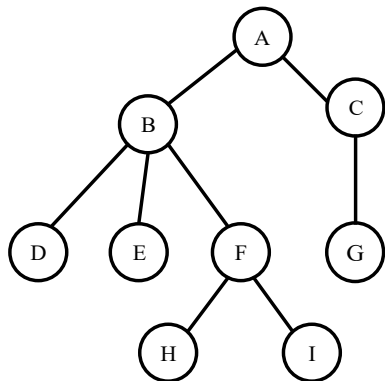


E



Be careful! A binary tree is a tree where each node has at most 2 children. A binary search tree is a binary tree with the additional property that the letter of each node is greater than the value of its left child, and less than the value of its right child. The valid trees are: (a), (b), and (d).

If one traversed the following tree in preorder (visit the root, and then each of the subtrees from left to right), in what order would nodes be visited? Be sure your answer is neat and clear!



Answer: A B D E F H I C G

A common mistake is not to recursively visit *all* nodes in each subtree.