

Note: Lecture Four materials adopts Section 10.1 - Stacks and queues, & Chapter 12 - Binary Search Trees in *Introduction to Algorithms*, Cormen, Leiserson, Rivest, and Stein, 2nd Edition, MIT Press .

Call Number: QA 76.6 C662 2001, Steacie Library at York; Online e-copy is also available through a valid York Passport ID.

1 Stacks (Textbook Section 4.2)

Stacks are dynamic sets in which the element removed from the set by Delete operation is prespecified as the one most recently inserted. In other words, the policy of last-in-first-out (LIFO) is implemented. The Insert operation on a stack is often called *Push*, and the Delete operations, which does not take an element argument, is called *Pop*.

Consider a stack of at most n element implemented as an array $S[1 \dots n]$, and an attribute $top[S]$ that indexes the most recently inserted element. The following code implement these operations.

StackEmpty(S)

```
1: if tops[S] = 0
2:   then return TRUE
3:   else return FALSE
```

Push(S, x)

```
1: top[S] <- top[S] + 1
2: S[top[S]] <- x
```

Pop(S)

```
1: if StackEmpty(S)
2:   then error "underflow"
3:   else top[S] <- top[S] - 1
4:   return S[top[S]+1]
```

It is obvious that each of these operations takes $O(1)$ time.

Exercise: How to implement StackFull(S)?

Example 1. Consider a stack S where the size of S is $n = 7$. S is initially empty (Figure 1 (a)). Stack S has four elements after $Push(S, 15)$, $Push(S, 6)$, $Push(S, 2)$, $Push(S, 9)$ (Figure 1 (b)).



Figure 1: A Stack Example

2 Queues (Textbook Section 4.3)

In a queue the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

The Insert operation on a queue is called *enqueue*, and the Delete operation is called *dequeue* (taking no element argument like *Pop*). The queue has a head and a tail. When an element is enqueued, it takes its place at the tail of the queue. The element dequeued is always the one at the head of the queue. We use an array $Q[1 \dots n]$ to implement a queue of at most $n - 1$ elements. The queue has

- an attribute $head[Q]$ that indexes, or points to, its head;
- an attribute $tail[Q]$ indexes the next location at which a newly arriving element will be inserted into the queue.

The elements in the queue are in locations $head[Q]$, $head[Q] + 1$, \dots , $tail[Q] - 1$, where we wrap around in the sense that location 1 immediately follows location n in a circular order. Note:

- Initially, we have $head[Q] = tail[Q] = 1$;
- When $head[Q] = tail[Q]$, the queue is empty. When the queue is empty, an attempt to dequeue an element causes the queue to underflow;
- When $head[Q] = tail[Q] + 1$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

Enqueue(Q , x)

```
1:  $Q[tail[Q]] \leftarrow x$ 
2: if  $tail[Q] == length[Q]$ 
3:   then  $tail[Q] \leftarrow 1$ 
4:   else  $tail[Q] \leftarrow tail[Q] + 1$ 
```

Dequeue(Q)

```
1:  $x \leftarrow Q[head[Q]]$ 
2: if  $head[Q] == length[Q]$ 
3:   then  $head[Q] \leftarrow 1$ 
4:   else  $head[Q] \leftarrow head[Q] + 1$ 
5: return  $x$ 
```

It is obvious that each of these operations takes $O(1)$ time.

Exercise: How to implement `QueueEmpty(S)` and `QueueFull(S)`?

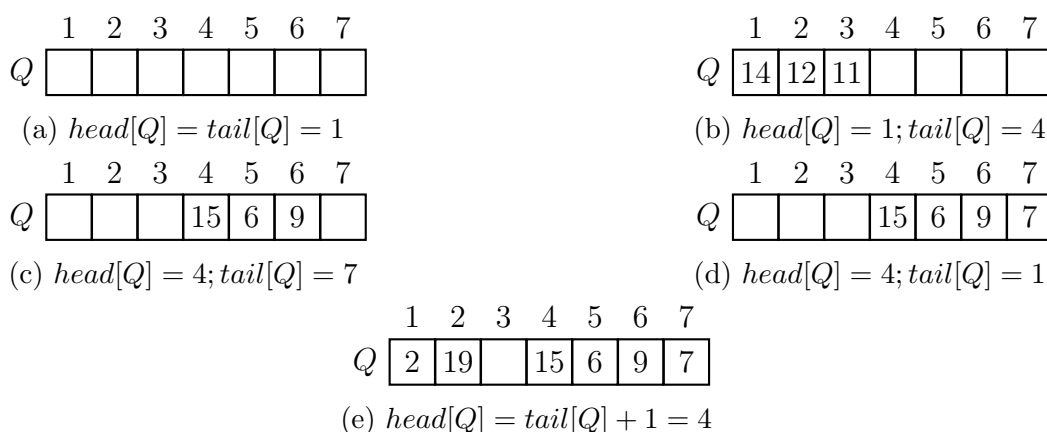


Figure 2: A Queue Example

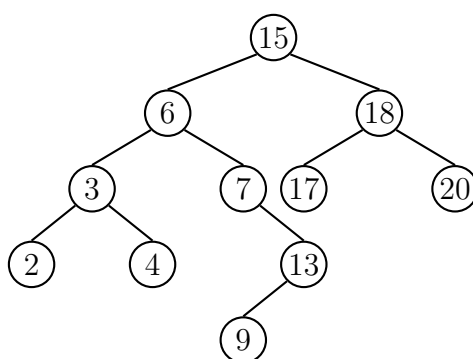


Figure 3: A Binary Search Tree Example

Example 2. Consider a stack Q where the size of S is $n - 1 = 6$. Q is initially empty (Figure 2 (a)). Queue Q has three elements after $Enqueue(Q, 14)$, $Enqueue(Q, 12)$, $Enqueue(Q, 11)$ (Figure 2 (b)). The Queue is empty again after performing $Dequeue(Q)$ for three times. Queue Q has three elements after $Enqueue(Q, 15)$, $Enqueue(Q, 6)$, $Enqueue(Q, 9)$ (Figure 2 (c)). Queue Q has four elements after $Enqueue(Q, 7)$ (Figure 2 (d)). Q is full after $Enqueue(Q, 2)$ and $Enqueue(Q, 19)$ (Figure 2 (e)).

3 Binary Search Trees (Textbook Chapter 5)

In addition to a key field and value/satellite data, each node in a binary search tree contains fields *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value *NIL*. The root node is the only node in the tree whose parent field is *NIL*.

The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**:

Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $key[y] \leq key[x]$. If y is a node in the right subtree of x , then $key[x] \leq key[y]$.

Example 3. Figure 3 is an example of binary search tree including 11 nodes.

The procedure *TreeInsert* inserts a new node z for which $key[z] = v$, $left[z] = NIL$, and $right[z] = NIL$, into a binary search tree T . It modifies T and some of the fields of z in such

a way that z is inserted into an appropriate position in the tree. `TreeInsert` begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x . After initialization, the while loop in lines 3 - 7 causes these two pointers to move down the tree, going left or right depending on the comparison of $\text{key}[z]$ with $\text{key}[x]$, until x is set to `NIL`. This `NIL` occupies the position where we wish to place the input item z . Lines 8 - 13 set the pointers that cause z to be inserted. `TreeInsert` runs in $O(h)$ time.

```
TreeInsert(T,z)
1:  y <- NIL
2:  x <- root[T]
3:  while x != NIL
4:      y <- x
5:      if key[z] < key[x]
6:          then x <- left[x]
7:          else x <- right[x]
8:  p[z] <- y
9:  if y = NIL
10:     then root[T] <- z
11:     else if key[z] < key[y]
12:         then left[y] <- z
13:         else right[y] <- z
```

Given a pointer to the root of the tree and a key k , `TreeSearch` returns a pointer to a node with key k if one exists; otherwise, it returns `NIL`. The nodes encountered form a path downward from the root of the tree, and thus the running time of `TreeSearch` is $O(h)$, where h is the height of the tree.

```
IterativeTreeSearch(x,k)
1: while x != NIL and k != key[x]
2:     if k < key[x]
3:         then x <- left[x]
4:         else x <- right[x]
5: return x
```

A minimum element can always be found by following left child pointers from the root until a `NIL` is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x . The binary-search-tree property guarantees that `TreeMinimum` is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $\text{key}[x]$, the minimum key in the subtree rooted at x is $\text{key}[x]$. If node x has a left subtree, then since no key in the right subtree is smaller than $\text{key}[x]$ and every key in the left subtree is not larger than $\text{key}[x]$, the minimum key in the subtree rooted at x can be found in the subtree rooted at $\text{left}[x]$. The pseudocode for `TreeMaximum` is symmetric. Both of these procedures run in $O(h)$ time since, as in `TreeSearch`, the sequence of nodes encountered forms a path downward from the root.

```
TreeMinimum(x)
1: while left[x] != NIL
2:     x <- left[x]
3: return x
```

```
TreeMaximum(x)
1: while right[x] != NIL
2:     x <- right[x]
3: return x
```

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 tree. The procedure is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which is found in line 2 by calling `TreeMinimum(right[x])`. On the other hand, if the right subtree of 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 . The running time of `TreeSuccessor` on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree.

`TreeSuccessor(x)`

```

1: if right[x] != NIL
2:   then return TreeMinimum(right[x])
3: y <- p[x]
4: while y != NIL and x == right[y]
5:   x <- y
6:   y <- p[y]
7: return y

```

I am going to use the BST example in Figure 3 to explain why this algorithm works. Given x , there are two cases:

- (Case 1) x does not have a right-subtree. Then x must be the maximum node (right-most node) of the left subtree of some node z .¹

In the BST of Figure 3, the pair of x - z where x is in case 1 is summarized in the following table:

x	2	4	9	13	17
z	3	6	13	15	18

- (Case 2) x has a right-subtree. Then there must be a minimal node z in the tree rooted at `right[x]`. In the BST of Figure 3, the pair of x - z where x is in case 2 is summarized in the following table:

x	3	6	7	15	18
z	4	7	9	17	20

We now prove that for any pair “ x - z ” of such kind, z is actually the successor of x .

For Case 1, if all nodes rooted at z ONLY are considered, z is the successor of x , since from BST property, we have

$$\dots x z \dots$$

Suppose u is not a node rooted at z and u is the successor of x , then there must be a root v for u and z (possibly $v = u$). In this case, u is either smaller, or larger than all nodes rooted at z (including surely x). Hence, z is the successor of x for Case 1.

For Case 2, if we do not consider the tree rooted at `right[x]`, then this is reduced to Case 1 and we have say z_0 as the successor. However, z_0 must be greater than all nodes rooted at `right[x]`, including z the minimum of these nodes. Hence z is actually the successor of x for Case 2.

¹Exception! There is a special case within Case 1 where x is the maximal element in the tree T . For Figure 3, for example, when x is with key 20, x does not have a successor.