Ch. 10 – Elementary Data Structures

Stacks, Queues, Lists and Trees

Dynamic Sets

We introduce a family of data structures that support (some or all of) the operations:

SEARCH(S, k) – S a set, k a key.
INSERT(S, x) – S a set, x a pointer to an object.
DELETE(S, x)
MINIMUM(S)
MAXIMUM(S)
SUCCESSOR(S, x)
PREDECESSOR(S, x)

The question will be: what’s the time complexity? Ideally, we would like to support all in time $O(\log n)$. 
Dynamic Sets

Supports the operations:
- \textsc{stack-empty}(S)
- \textsc{push}(S, x)
- \textsc{pop}(S)

Each of the stack operations requires $O(1)$ time.

\textsc{stack-empty}(S)
\begin{verbatim}
1 if $S\.top == 0$
2 return TRUE
3 else return FALSE
\end{verbatim}

\textsc{push}(S, x)
\begin{verbatim}
1 $S\.top = S\.top + 1$
2 $S[S\.top] = x$
\end{verbatim}

\textsc{pop}(S)
\begin{verbatim}
1 if \textsc{stack-empty}(S)
2 error "underflow"
3 else $S\.top = S\.top - 1$
4 return $S[S\.top + 1]$
\end{verbatim}

Figure 10.1 An array implementation of a stack $S$. Stack elements appear only in the lightly shaded positions. (a) Stack $S$ has 4 elements. The top element is 9. (b) Stack $S$ after the calls \textsc{push}(S, 17) and \textsc{push}(S, 3). (c) Stack $S$ after the call \textsc{pop}(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.
Dynamic Sets

Queue

Supports the operations:

ENQUEUE(Q, x)
1. Q[Q.tail] = x
2. if Q.tail == Q.length
3. Q.tail = 1
4. else Q.tail = Q.tail + 1

DEQUEUE(Q)
1. x = Q[Q.head]
2. if Q.head == Q.length
3. Q.head = 1
4. else Q.head = Q.head + 1
5. return x

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes O(1) time.
Dynamic Sets

Lists

Figure 10.3 (a) A doubly linked list $L$ representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The `next` attribute of the tail and the `prev` attribute of the head are `NIL`, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of `LIST-INSERT(L, x)`, where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call `LIST-DELETE(L, x)`, where $x$ points to the object with key 4.

Dynamic Sets

Operations on Linked Lists

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x $\neq$ NIL and x.key $\neq$ k
3     x = x.next
4  return x

LIST-DELETE(L, x)
1  if x.prev $\neq$ NIL
2     x.prev.next = x.next
3  else L.head = x.next
4  if x.next $\neq$ NIL
5     x.next.prev = x.prev

LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head $\neq$ NIL
3     L.head.prev = x
4  L.head = x
5  x.prev = NIL
```
Dynamic Sets

Linked Lists and Sentinels

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. (a) An empty list. (b) The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. (c) The list after executing LIST-INSERT$(L, x)$, where $x.key = 25$. The new object becomes the head of the list. (d) The list after deleting the object with key 1. The new tail is the object with key 4.

Dynamic Sets

Ops on Linked Lists and Sentinels

LIST-SEARCH$(L, k)$

1. $x = L.nil.next$
2. while $x \neq L.nil$ and $x.key \neq k$
3. $x = x.next$
4. return $x$

LIST-INSERT$(L, x)$

1. $x.next = L.nil.next$
2. $L.nil.next.prev = x$
3. $L.nil.next = x$
4. $x.prev = L.nil$
Dynamic Sets

Pointers and Objects

Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays key, next, and prev. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable $L$ keeps the index of the head.

Dynamic Sets

Pointers and Objects

Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array $A$. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three attributes key, next, and prev correspond to the offsets 0, 1, and 2, respectively, within each object. A pointer to an object is the index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.
Dynamic Sets

Pointers and Objects

**Figure 10.7** The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. (a) The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. (b) The result of calling ALLOCATE-OBJECT() (which returns index 4), setting key[4] to 25, and calling LIST-INSERT(L, 4). The new free-list head is object 8, which had been next[4] on the free list. (c) After executing LIST-DELETE(L, 5), we call FREE-OBJECT(5). Object 5 becomes the new free-list head, with object 8 following it on the free list.

Dynamic Sets

Pointers and Objects

**Figure 10.8** Two linked lists, \( L_1 \) (lightly shaded) and \( L_2 \) (heavily shaded), and a free list (darkened) intertwined.
Dynamic Sets

Rooted Trees

Figure 10.9  The representation of a binary tree $T$. Each node $x$ has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The key attributes are not shown.

Dynamic Sets

Rooted Trees

Figure 10.10  The left-child, right-sibling representation of a tree $T$. Each node $x$ has attributes $x.p$ (top), $x.left-child$ (lower left), and $x.right-sibling$ (lower right). The key attributes are not shown.