Stacks, Queues and Linked Lists

Adnan Aziz

1 Dynamic sets

CLRS Part III, page 197
In mathematics, a set is a well-defined collection of elements (elements could be numbers, functions, geometric shapes); could be infinite.
Algorithms—operate on sets. Two special aspects of these sets is that they are *finite* and *dynamic*.
Often—only operations are *insert*, *delete*, *test membership*.

- Can get more complicated: *extract-min*

Typical implementation:

- elements are objects—given pointer, fields can be examined and manipulated

A common scenario is that one field is a "key". E.g., object may contain id, name, birthday, address; any of these could be the key.

- If the keys are all distinct, can view dynamic set as simply a set of keys.

Sometimes objects are drawn from a "totally ordered" set (e.g., the real numbers).
There are two prototypical operations: *queries* return information about the set, and *update* modify the set
Examples:

```
search(S,k)

insert(S,k)
```

```
delete(S,k)
minimum(S)
maximum(S)
successor(S,k)
predecessor(S,k)
```

Note that these operations can use these to enumerate the elements
Runtimes are measures in terms of size of the set, i.e., the number of elements.

2 Stacks and Queues

CLRS 10.1
Dynamic sets in which elements removed by delete is pre-specified

- stack—always delete most recently inserted element "LIFO"
- queue—always delete element longest in set "FIFO"

2.1 Stacks

insert —usually called "push"

delete —usually called "pop"

Can implement stack of at most n elements using an array `S[1..n]` (See Figure 10.1, CLRS)

- keep attribute `top[S]` which indexes the most recently inserted element (`top[S] = 0` $\Rightarrow$ stack is empty)
- *underflow*—try popping empty stack
- *overflow*—try pushing fill stack)

Pseudo-code: (ignore overflow; lv. for HW)

```
STACK-EMPTY(S)
  if top[S] = 0
    then return TRUE
    else return FALSE

PUSH(S,x)
  top[S] <- top[S] + 1
  S[top[S]] <- x

POP(S)
  if STACK-EMPTY(S)
    then error "underflow"
    else
      top[S] = top[S] - 1
      return S[top[S] + 1]
```

All the operations have $O(1)$ time complexity.

2.2 Queues

"FIFO"—**head**: element which has been in for longest, **tail**: location at which to insert

- *insert*—usually called "enqueue"
- *delete*—usually called "dequeue"

Can implement queue of at most $n-1$ elements using an array `Q[1..n]` (See Figure 10.2, CLRS)

- keep attribute `head[Q]` which indexes the head, and attribute `tail[Q]` which is the location at which to add the next element
- `head[Q] = tail[Q]` $\Rightarrow$ `Q` is empty
- `head[Q] = tail[Q]+1` $\Rightarrow$ `Q` is full

Implementations of enqueue, dequeue **without** error checking:

```
ENQUEUE(Q,x)
  Q[tail[Q]] <- x
  if tail[Q] = length[Q]
    then tail[q] = 1
  else tail[Q] <- tail[Q] + 1

DEQUEUE(Q)
  x <- Q[head[Q]]
  if HEAD[Q] = length[Q]
    then head[Q] <- 1
    else head[Q] <- head[Q] + 1
  return x
```

Runtimes? All $O(1)$
What is the big shortcoming with the array based implementation?

3 Linked Lists

CLRS 10.2

Conceptually: objects arranged in linear order. Differs from arrays in that in arrays index $+1$ gives next element; in linked list, we use a pointer.

- Will see: can implement all operations on a linked list.

Doubly linked list L: each element is an object with a key field, a $next$ field, and a $prev$ field. (Of course, there maybe other satellite data.)
It's important that you keep track of the difference between element and key!
Given an element x:

- next(x)—pointer to successor (NIL $\Rightarrow$ no successor; such an element is called the "tail")
- prev(x)—pointer to predecessor (NIL $\Rightarrow$ no predecessor; such an element is called the "head")

Variations:

- singly linked
- sorted
- circular list—prev of head is tail; next of tail is head (makes some functions easier to write)

We will stick to unsorted, doubly linked lists.
Example—see Figure 10.3, CLRS.

3.1 Searching in Linked Lists

Given list L, and a key k, return a pointer to the first object with key k (not present $\Rightarrow$ return *NIL*)

```
LIST-SEARCH(L,k)
  x <- head[L]
  while x != NIL and key[x] != k
    do x <- next[x]
  return x
```

Runtime complexity is $\Theta(n)$

3.2 Inserting into a Linked List

Given list L, element x (whose key field is already set), insert x into list.

- intuition—“splice” onto the front

```
LIST-INSERT(L,x)
  next[x] <- head[L]
  if head[L] != NIL
     then prev[head[L]] <- x
  head[L] <- x
  prev[x] <- NIL
```

Runtime? $\Theta(1)$

3.3 Deleting from a Linked List

Remove an element x from list L

- assume given pointer to x—we’ll “splice” out x
  - how to generalize to deleting element given only key? use `LIST-SEARCH` function

```
LIST-DELETE(L,x)
  if prev[x] != NIL
    then next[pred[x]] <- next[x]
    else head[L] <- next[x]
  if next[x] != NIL
    then pred[next[x]] <- prev[x]
```

3.4 Sentinels

Observe: code for delete is complicated by tests for boundary conditions. Can get around this by use of "sentinels."

- Not that helpful
  - clearer code
  - small speedup
  - more memory

Section 10.3, CLRS discusses how one can implement linked lists in a language which does not support pointers/heaps/memory management. We don't need to worry about this in `C++` but you may enjoy reading this section to get an idea of how `new, malloc, delete, free` work.