Stacks, queues and linked lists

Mikko Karvonen

mkarvonen@iua.upf.es
Elementary Data Structures

- Stacks, queues and linked lists are elementary data structures that are based on pointers between some of the elements. These pointers are links to the first or last element in the set, to an element’s previous or next element and so on.

- Stacks and queues are dynamic sets in which the element removed from the set by DELETE operation is pre-specified. In linked lists, you must select or search the element to be removed.

- In stacks, the most recently inserted element is deleted = last-in, first-out (LIFO)

- And in queues, the element that has been in the set for the longest time is deleted = first-in, first-out (FIFO)
Stacks (1/2)

- In stacks, the basic operations INSERT and DELETE are called PUSH and POP, respectively, to match the physical world. The book uses an analogy of a stack of plates in a cafeteria; the plate that has been put in the stack most recently has to be taken out first.

- A stack can be constructed using an array:

  array $S[1..n]$
  attribute $top[S]$ – indexes the element at the top

  so a stack is an array $S[1..top[S]]$. 
Stacks (2/2)

- Stack operations in pseudo-code:

  STACK-EMPTY(S)
  if top[S] = 0
    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]

  The running time for the operations is $O(1)$ (constant time).
Queues (1/2)

- In queues, the INSERT and DELETE operations are called ENQUEUE and DEQUEUE. As the POP operation of the stack, the DEQUEUE operation does not take any parameters, it just deletes the element from the head of the queue.

- Stacks have only one pointer (the top[S] attribute that points to the top) but queues have two: head[Q] points to the head and tail[Q] points to the end of the queue.

- A queue can be realized efficiently using a circular array Q that “wraps around”, i.e. so that when we are at the last index of an array the “next” element is at index 1. The only thing we have to take care of is updating the head[Q] and tail[Q] attributes.

- When head[Q] = tail[Q], the queue is empty.
  Initially, head[Q] = tail[Q] = 1

  When head[Q] = tail[Q] + 1, the queue is full.
Queue operations in pseudo-code:

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

These are also O(1) time operations.
Linked lists (1/6)

- Each element of a linked list is an object with a key value (the actual data) + “satellite data” such as next and prev, that are links to the next and previous objects in the list.

- **Singly linked lists** only have the next attribute, **doubly linked lists** have both next and prev.
Linked lists (2/6)

- Linked lists are easy to realize with Object Oriented (OO) languages due to their object nature, but can also be realized using arrays - more about this later.
- head[L] and tail[L] attributes point to the head and tail of the list

`head[L] = NIL` -> list empty

- A list can be sorted or unsorted. The sorted linear order of a list matches the keys’ linear order.
- Lists can also be circular, so that the first `prev` points to the tail and the last `next` points to the head.
Linked lists (3/6)

- List operations in pseudo-code:

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

  LIST-DELETE(L, x)
  if prev[x] != NIL
    then next[prev[x]] <- next[x]
  else head[L] <- next[x]
  if next[x] != NIL
    then prev[next[x]] <- prev[x]
LIST-SEARCH(L, k)  
  x <- head[L]  
  while x != NIL and key[x] != k  
      do x <- next[x]  
  return x

O(n) time (in worst-case scenario the algorithm has to go through the whole list!)

- LIST-INSERT and LIST-DELETE are both O(1), but LIST-DELETE with LIST-SEARCH is O(1) + O(n)…
By using *sentinels* (dummy objects) the list algorithms can be simplified a bit. The gain is mostly in clarity of code rather than speed.

When using sentinels, we introduce a new object `nil[L]` that represents NIL but has all the fields of the other list elements (such as `prev`, `next`).

In code, references to NIL are replaced by references to `nil[L]`. In the case of doubly linked lists, the sentinel changes the list into a circular doubly linked list, because the `nil[L]` object is sort of placed before `head[L]` and after `tail[L]`. Thus, `next[nil[L]]` points to the head and `prev[nil[L]]` points to the tail -> no need for `head[L]` anymore!
Linked lists (6/6)

- List operations in pseudo-code, revisited:

  LIST-DELETE'(L, x)
  next[prev[x]] <- next[x]
  prev[next[x]] <- prev[x]

  LIST-INSERT'(L, x)
  next[x] <- next[nil[L]]
  prev[next[nil[L]]] <- x
  next[nil[L]] <- x
  prev[x] <- nil[L]
Implementing pointers and objects (1/4)

- As mentioned before, linked lists, as well as stacks and queues, are pretty straightforward to construct in OO languages due to their object nature. But what about non-OO languages?

- A multiple-array representation of objects (in the case of linked lists)

  3 arrays: key, next, prev

  So an “object” can be represented as:
  
  `key[x], next[x], prev[x]`  
  
  \( (x = \text{index}) \)
Implementing pointers and objects (2/4)

- A single-array representation

Keys and prev & next pointers are in the same array A. We use offsets for the attributes:

<table>
<thead>
<tr>
<th>key</th>
<th>next</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

So,

- key[i] = A[i] (or A[i + 0]),
- next[i] = A[i + 1],
- prev[i] = A[i + 2]

and so on...
Implementing pointers and objects (3/4)

- We also need to take care of allocating and freeing objects.

  In multiple-array representation, we use an array with a length of $m$ and with $n \leq m$ elements.

  $n =$ elements in the set  
  $m-n =$ free objects (can be used to represent the elements inserted into the dynamic set in the future)

  The *free list* is actually a singly linked list which uses only the *next* array to find a place for a new element. A reference to the head is held in a global variable *free*. 
Implementing pointers and objects (4/4)

Allocating and freeing in pseudo-code:

*(needs to be run before LIST-INSERT)*

ALLOCATE-OBJECT()
  if free = NIL
    then error "out of space"
  else x <- free
  free <- next[x]
  return x

*(needs to be run after LIST-DELETE)*

FREE-OBJECT(x)
  next[x] <- free
  free <- x