FTP_Alg_Cap1_1

jungkyu.canci@hslu.ch

September the 16th 2024

Introduction: Algorithms

- An **algorithm** is a procedure that takes an **input** and with a sequence of computational steps produces an **output**.
- Usually inputs and outputs are values or sets of values.
- An algorithm can be viewed as a tool for solving computational problems.
- Example: Sort a sequence of number into non-decreasing order.

  Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

  Output: A reordering $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence, with the property $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

  For instance

  Input: $(34, 21, 45, 67, 45, 12)$

  Output: $(21, 21, 34, 45, 45, 67)$

- An algorithms is said to be **correct**, if for every input (given in the correct format) in a finite time it halts with a correct output. In this case we say that the algorithm **solves** the given computational problem.
- An incorrect algorithm might never stop (infinite run) or produce an incorrect output.
- An algorithm can be expressed in "words"(pseudo-code). But all steps should be clearly given and they should provide the computational procedure to be followed.

What kinds of problems are solved by algorithms?

Just a short list in comparison with all possibilities:

- Data analysis for human DNA
- Algorithms to manage large amount of data in internet.
- Managing data in e–commerce: privacy of personal information. Public-key-cryptography and digital signature.
- Optimization problems: how to maximize the profit and minimize costs.
- Finance, pricing of products: determining the best price of a product in order to maximize the final profit.

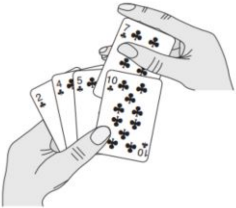
Insertion sort: definition

Recall of a basic algorithm: **insertion sort**.

Input A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output A reordering $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence, with the property $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

insertion sort is efficient for sorting a small set of "numbers" It works in the same way many people sort a hand of playing cards:

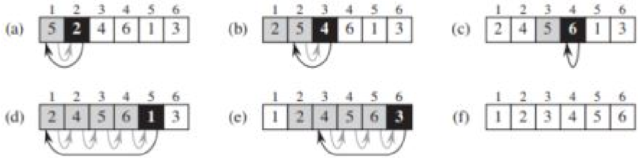


Insertion sort: pseudo code

In this course we will describe algorithms as programs written in a pseudocode, that could be written in a similar way in your preferred programming language (e.g. C, C++, Java, Python or Pascal).

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Example:



More on sorting algorithms?

`https://www.youtube.com/watch?v=kPRA0W1kECg`

How to tell if an algorithm is any good?

For example, is Quicksort better than Insertion sort? And by how much?

- Implement both algorithms, and compare: this is expensive and error prone
- Perform an analytical comparison via **Algorithm Analysis**

Algorithm Analysis

- Analyzing algorithms is the prediction of the resources that an algorithm requires.
- The analysis can be done in terms of memory space: maximum memory required at any time during the execution, in terms of "basic" memory blocks (e.g. number of bits, or number of integers, and so on).
- But usually we are interested in the computational (running) time of the algorithm.
- The running time represents the number of "basic" operations that the algorithm has to perform.
- Often for a given problem we consider several algorithms to solve it and we compare their running times.

Analysis of insertion sort

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

- $n = A.length$
- t_j denotes the number of times the **while** loop test in line 5 is executed for that value of j.
- A "usual" **for** or **while** loop, the test is executed one time more that the loop body.
- c_i are constants (= number of steps to execute each single command), so the contribution of each line to the running time is $c_i \cdot$ "times". Comments does not cost.

Analysis of insertion sort

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

- So $T(n)$, the running time, is given by

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j +$$

$$+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_8)n - (c_2 + c_4 + c_8) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

Analysis of insertion sort: best case and worst case

- The best case occurs when the array is already sorted. In this case $A[i] \leq key$ and so $t_j = 1$ (and $t_j - 1 = 0$). Thus

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

  In this case the running time is of the form $T(n) = an + b$ (i.e. a **linear function** of n)

- The worst case is when the array is in the decreasing order. In this case $A[j]$ hast to be compared with all entries in $A[1, \ldots, j-1]$, thus $t_j = j$.
  Recall:

$$\sum_{r=1}^{n} r = \frac{n(n+1)}{2}$$

  Thus

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \ , \ \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Analysis of insertion sort: best case and worst case

- In the worst case we have

$$
\begin{aligned}
T(n) = &(c_1 + c_2 + c_4 + c_8)n - (c_2 + c_4 + c_8) + \\
&+ c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) \\
= &(c_1 + c_2 + c_4 + c_8)n - (c_2 + c_4 + c_8) + \\
&+ c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2} \\
= &(c_1 + c_2 + c_4 + c_8)n - (c_2 + c_4 + c_8) + \\
&+ c_5 \left(\frac{n^2 + n - 2}{2} \right) + (c_6 + c_7) \frac{n^2 - n}{2} \\
= &a_2 n^2 + a_1 n + a_0
\end{aligned}
$$

$a_2 = \frac{c_5 + c_6 + c_7}{2}$, $a_1 = c_1 + c_2 + c_4 + c_8 + \frac{c_5 - c_6 - c_7}{2}$, $a_0 = -c_2 - c_4 - c_8 - c_5$

Analysis of insertion sort: computational complexity

- In the worst case we have

$$T(n) = a_2 n^2 + a_1 n + a_0$$
$$T(n) = O(n^2), \text{ as } n \to \infty$$

- Asymptotically, we say that $T(n)$ grows no faster than the function $f(n) = n^2$.
- Therefore, **Insertion sort** as a running time of $O(n^2)$.
- This result is intuitive too, given that Insertion sort has two nested loops, and that each can be repeated at most n times.

Asymptotic notation, functions, and running times

Let f and g be non negative real valued functions (defined usually on the interval $[0, \infty]$).

- $f(x) = O(g(x))$ as $x \to \infty$ if there exists a positive number M and a real number x_0 such that

$$f(x) \leq M \cdot g(x) \quad \forall x \geq x_0$$

- Example: $c = O(x^\alpha)$ for all non negative constants c and positive exponent α.
- $f(x) = \Omega(g(x))$ as $x \to \infty$ if there exists a positive number m and a real number x_0 such that

$$m \cdot g(x) \leq f(x) \quad \forall x \geq x_0$$

- Example: $x^\alpha = \Omega(1)$ for all positive exponent α.

Asymptotic notation, functions, and running times

- $f(x) = \Theta(g(x))$ as $x \to \infty$ if there exist two positive numbers m and M and a real number x_0 such that

$$m \cdot g(x) \leq f(x) \leq M \cdot g(x) \quad \forall x \geq x_0$$

- Examples: $3x^2 - x + 1 = \Theta(x^2)$. More general: let $P(x)$ be a polynomial of degree d, then $P(x) = \Theta(x^d)$.
- Remarks: we have $f(x) = O(g(x))$ if and only if $g(x) = \Omega(f(x))$. $f(x) = \Theta(x)$ if and only if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.
- Less used: $f(x) = o(g(x))$ for every positive constant c there exists a number x_0 (it may depend on c) such that $f(x) \leq c \cdot g(x)$ for all $x \geq x_0$.
  $f(x) = \omega(g(x))$ for every positive constant c there exists a number x_0 (it may depend on c) such that $c \cdot g(x) \leq f(x)$ for all $x \geq x_0$.

Asymptotic notation, functions, and running times

Examples: Let $T(n)$ be the running time of an algorithm

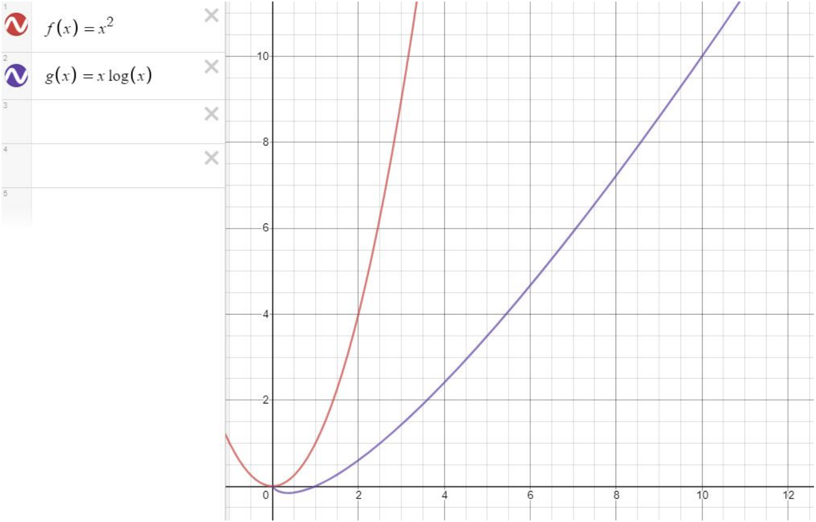
- $T(n) = O(n^{100})$ means that $T(n)$ grows asymptotically no faster than n^{100}
- $T(n) = \Omega(n^5)$ means that $T(n)$ grows asymptotically no slower than n^5
- $T(n) = \Theta(n^8)$ means that $T(n)$ grows asymptotically as fast as n^8

Examples:

- As we have seen insertion sort is $O(n^2)$ and $\Theta(n^2)$ in the worst case. More precisely in the best case is linear (i.e. $\Theta(n)$) and in the worst case is $\Theta(n^2)$. In the calculation of the complexity one has to consider the worst case.
- In Quick Sort (we will see later) the running time is on **average** $\Theta(n \log n)$ (and $\Theta(n^2)$ in the worst case).

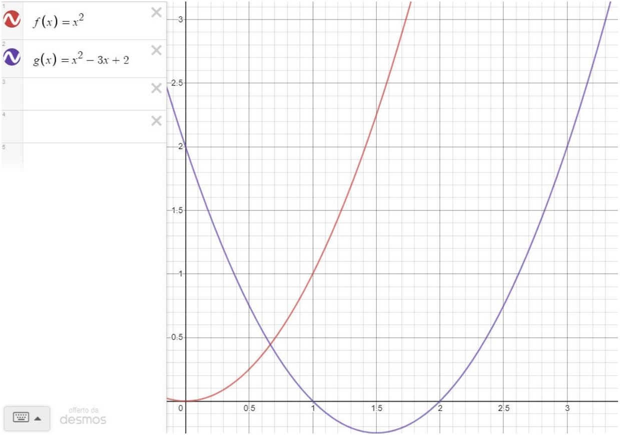
Which one is faster?

Insertion sort vs Quick sort



Asymptotic analysis

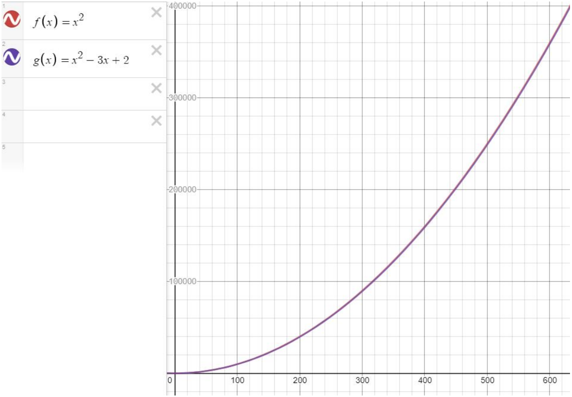
We compare the function $f(n) = n^2$ and $g(n) = n^2 - 3n + 2$ as $n \to \infty$



But n is small

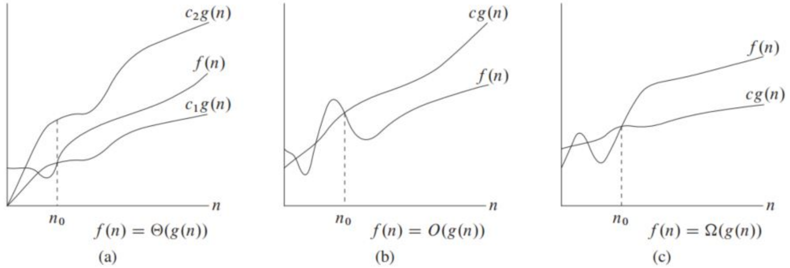
Asymptotic analysis

We re-scale the previous comparison with bigger x's



Asymptotic analysis

We give a graphic example of use of Θ, O and Ω.



In https://en.wikipedia.org/wiki/Big_O_notation Big–O functions listed in a increasing order.

Data Structures: Introduction (the following part from here to the end of the slides will be not examinated)

- Sets are important in computer science (as well as in mathematics) but algorithms could transform them. Therefore in computer science sets are **dynamic**.
- Algorithms may require several transformations on sets. These dynamic sets, on which these operations act are called *dictionaries*
- How to define a dynamic set? It depends upon the operations that must be supported.
- Usually a dynamic set is an object containing pointers to other objects.
- **keys** identify attribute of objects.
- Objects may contain **satellite data**, which can be used for the manipulation of the dynamic sets.

Operations on a dynamic set

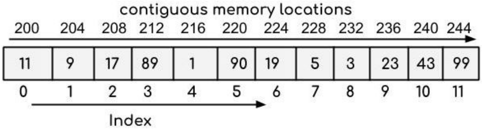
- There are two types of operations:
  - **queries** (they only return information on the sets)
  - **modifying operations** (they may change the sets)
- Some example of modifying operations:
  - INSERT(S, x): A given set S is augmented with the element pointed to by x
  - DELETE(S, x): For a given pointer x in a set S, the operation removes x from S. (pointer and not key)
- Some example of queries:
  - SEARCH(S, k): for a given set S and a key value k returns a pointer x such that $x.key = k$, or NIL is k is no element of S.
  - MINIMUM(s): in a totally ordered set, it returns a pointer with the element of S with the smallest key.
  - MAXIMUM(s): in a totally ordered set, it returns a pointer to the element of S with the largest key.

Operations on a dynamic set

- Other example of queries:
  - SUCCESSOR(S, x): for a given element x whose key is from a totally ordered set S, it returns a pointer to the next larger element in S, or NIL is x is already a maximum element of S.
  - PREDECESSOR(S, x): for a given element x whose key is from a totally ordered set S, it returns a pointer to the next smaller element in S, or NIL is x is already a minimum element of S.
- We can extend the above two queries so that they apply to sets with non–distinct keys.
- For a set on n keys, the normal presumption is that a call to MINIMUM followed to a iterated $n - 1$ times call to SUCCESSOR enumerates the elements in the set in sorted order.

Arrays

- An array (or data structure) in computer science is a data structure consisting of a set of elements (values or variables) identified by an index or a key. In an array the position of each element can be calculated from its index (which can be a multidimensional one) via a mathematical formula.
- The simple type of array are the uni-dimensional one, called also **linear array**. E.g.



- A two dimensional array can be represented with a matrix.

Arrays: complexity

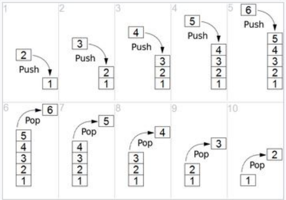
Complexity for each operation

- SEARCH(S, k): $O(1)$, direct access with index/key k
- INSERT(S, x): $O(n)$, we might need to re-allocate and copy the array
- DELETE(S, x): $O(n)$, same as above
- MINIMUM(S): $O(n)$, loop over the array
- MAXIMUM(S): $O(n)$, loop over the array
- SUCCESSOR(S, x): $O(n)$, because we need to search for x
- PREDECESSOR(S, x): $O(n)$, because we need to search for x

Stacks and Queues

Here we just present some elementary example of data structures. We start with stacks and queues.

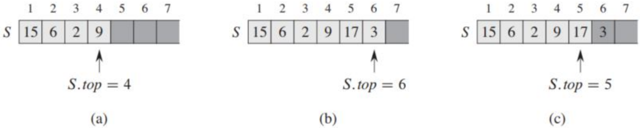
- Stacks and queues are dynamic sets where DELETE operates in a prepspecified manner.
- In a **stack** one delete the most recently inserted element. This specific operation is named **LIFO** (Last In First Out).
- In a **queue** one delete the element, which has been in the set for the longest time. This specific operation is named **FIFO** (First In First Out).



Simple representation of a stack runtime with *push* and *pop* operations.

Stacks

- PUSH is the name for INSERT in a stack.
- POP is the name for DELETE in a stack.
- The figure below show a stack over an array $S[1 \dots n]$. The attribute S.top indicates the most recently inserted element.



the stack consists of elements $S[1 \dots S.top]$

Stacks: standard operations

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE
```

$S.top = 0$ means that the stack is empty. STACK–EMPTY tests it.

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

We say that the stack **underflows** if we attempt to pop an empty stack, which is normally an error.

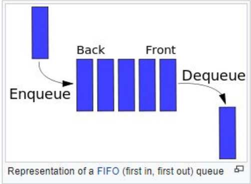
```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

We say that the stack **overflows** if we attempt to push an already full stack (in the pseudocode we don't worry about it) .

All above operations take $O(1)$ time.

Queues

- ENQUEUE is the INSERT operation in a queue.
- DEQUEUE is the DELETE operation in a queue.
- Because the FIFO property of queues dequeue has no element argument.
- A queue has a **head** and a **tail**.

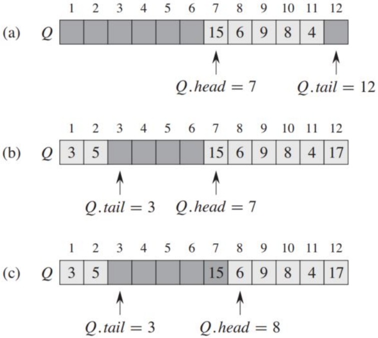


Picture from Wikipedia:

`https://en.wikipedia.org/wiki/Queue_(abstract_data_type)`

- With enqueue an element take always the place at the tail of the queue
- With dequeue we eliminate the element at the head of the queue

Queues: standard operations



ENQUEUE(Q, x)

$Q[Q.tail] = x$
if $Q.tail == Q.length$
    $Q.tail = 1$
else $Q.tail = Q.tail + 1$

DEQUEUE(Q)

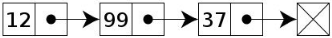
$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
else $Q.head = Q.head + 1$
return x

Example of a queue implemented using an array $Q[1 \ldots 12]$. Each operation takes $O(1)$ time.

Linked list

The material in the next slides is consodered optional (it will be not examinated).

- A **linked list** is a data structure consisting of objects that are arranged in a linear order.
- In an array the order is given by the same order of the array indices.
- In a linked list the order is determined by a pointer in each object.
- All operations listed above for dynamic sets are supported by linked lists.



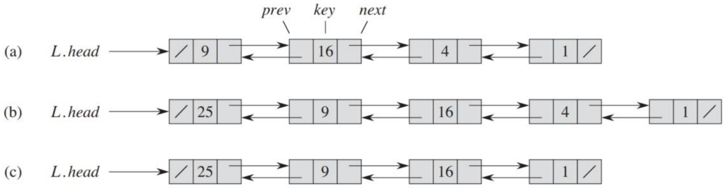
Picture from Wikipedia: `https://en.wikipedia.org/wiki/Linked_list`

- A list can be *singly linked*, as above, or *doubly linked*, sorted or not, and it may be circular or not.
- in a doubly linked list, other than an attribute key, there are two pointer attributes, *next* and *prev*. In a singly linked list we omit the *prev* pointer.

Doubly-Linked-List

- Let L be a doubly linked list with an attribute *key*. The objects can also contain other satellite data.
- For an element x of the list, $x.next$ points to its successor in the linked list. $x.prev$ to its predecessor.
- $x.prev =$NIL ($x.next =$NIL) means that the element x has no predecessor (successor). This means that x is the first (last) element of the list, that is the **head** (**tail**).
- $L.head$ ($L.tail$) point to the first (last) element of the list.

Linked-List



(a) L represents a doubly linked list on the set $\{1, 4, 9, 16\}$. Each element is an object with attributes for the key and pointers (the arrows)

(b) List after operation LIST–INSERT(L, x), where $x.key = 25$.

(c) List after operation LIST–DELETE(L, x) where x points to the object with key 4.

Linked-List: standard operations

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

LIST–SEARCH(L, k) returns a pointer to the first element of the list with key k. No element with k exists, so the answer is NIL. The time cost (in the worst case) is $\Theta(n)$ (where n is the number of objects in the list).

Linked-List: standard operations

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

Let x be an element whose attributes have been already set. LIST–INSERT sticks the element x in fronf of the list (see previous figure). It can be that our List is empty. $L.head.prev$ denotes the attribute of the object that $L.head$ points to. The running time does not depend on the number of objects, so it is $O(1)$.

Linked-List: standard operations

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

LIST-DELETE removes an element x from a linked list L. We have a pointer to x, that deletes x and updates the pointers. If we want to delete an element with a given key, we must first call LIST–SEARCH to get a pointer to that element.
The running time is $O(1)$ if the element x is given. It is $\Theta(n)$ (with n the numbers of objects) if we have to call LIST–SEARCH.

Circular doubly linked-list:

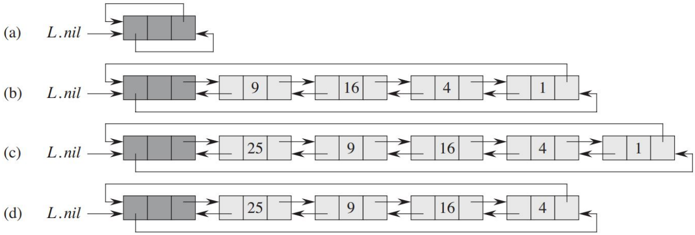
If we were allowed to do not consider boundary conditions, the previous pseudo–codes would be much simpler, for instance:

LIST-DELETE$'(L, x)$

1 $x.prev.next = x.next$
2 $x.next.prev = x.prev$

We simplify boundary conditions by considering a **sentinel**, which is a dummy element that will denote by $L.nil$ and added to a doubly linked list L. This new element will appear between the head an tail of the list L obtaining a **circular doubly linked list**.

Circular doubly linked-list:



- (a) An empty list.
- (b) The linked list from the previous slides, with key 9 at the head and key 1 at the tail.
- (c) The list after executing LIST-INSERT(L, x), where $x.key = 25$.
- (d) The list after deleting the object with key 1. The new tail is the object with key 4.

Circular doubly linked-list:

In a circular doubly linked list the sentinel *L.nil* is such that the attribute *L.nil.prev* points to *L.tail*. Whereas *L.nil.next* points to *L.head*

LIST-SEARCH$'(L, k)$

$x = L.nil.next$
while $x \neq L.nil$ and $x.key \neq k$
    $x = x.next$
return x

LIST-INSERT$'(L, x)$

$x.next = L.nil.next$
$L.nil.next.prev = x$
$L.nil.next = x$
$x.prev = L.nil$

How you can see the use of sentinel simplifies and clarifies the codes.

Circular doubly linked-list:

- The insertion of a sentinel in the above two cases simplifies and clarifies the code but, it doesn't affect the running time. Thus it remain $O(1)$ for LIST–INSERT and $O(n)$ for LIST–SEARCH.
- In some other case the use of sentinels can decrease significantly the running time.
- Sometime one has to use sentinels judiciously. For example, when there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.

Remarks:

- Queues and stacked are **abstract data structures**.
- They can be implemented using an array or linked list.
- The latter is a good choice given that only the head / tail needs to be accessed, added, or deleted.
- Tips: think about the pile of plates and the order in which they arrive and you wash them.