Algorithms

Stacks, queues, and linked lists

Emanuele Rodolà
rodola@di.uniroma1.it

Sapienza Università di Roma

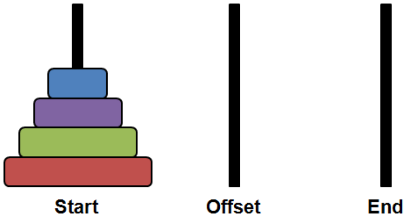
2nd semester a.y. 2020/2021 · March 25, 2021

Stack

In a stack, the last element we insert is also the first we take off. This is called **LIFO** (last-in, first-out) policy:

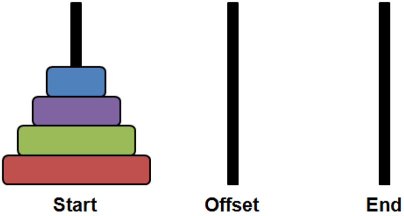
Stack

In a stack, the last element we insert is also the first we take off. This is called **LIFO** (last-in, first-out) policy:



Stack

In a stack, the last element we insert is also the first we take off. This is called **LIFO** (last-in, first-out) policy:



Insert (push) and remove (pop) operations must be efficient at the top.

Stack

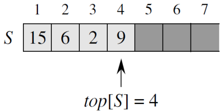
A stack of n elements can be implemented using an array $S[1 \ldots n]$.

Further, we use a pointer $top[S]$ to the most recently inserted element.

Stack

A stack of n elements can be implemented using an array $S[1 \ldots n]$.

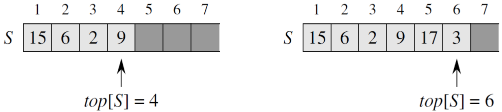
Further, we use a pointer $top[S]$ to the most recently inserted element.



Stack

A stack of n elements can be implemented using an array $S[1 \ldots n]$.

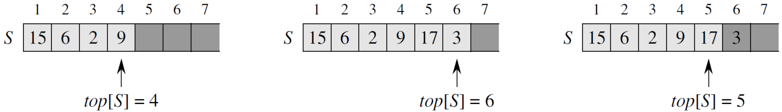
Further, we use a pointer $top[S]$ to the most recently inserted element.



Stack

A stack of n elements can be implemented using an array $S[1 \ldots n]$.

Further, we use a pointer $top[S]$ to the most recently inserted element.

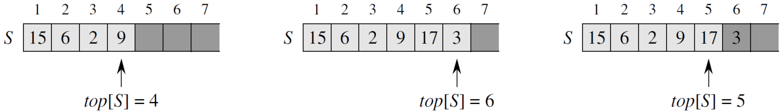


The pop operation does not necessarily remove the value from the array.

Stack

A stack of n elements can be implemented using an array $S[1 \ldots n]$.

Further, we use a pointer $top[S]$ to the most recently inserted element.



The pop operation does not necessarily remove the value from the array.

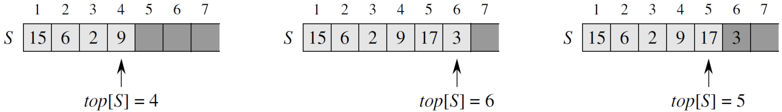
In practice, each stack has a maximum size.

If a push is made beyond the max size, we get a stack overflow.

Stack

A stack of n elements can be implemented using an array $S[1 \ldots n]$.

Further, we use a pointer $top[S]$ to the most recently inserted element.



The pop operation does not necessarily remove the value from the array.

In practice, each stack has a maximum size.

If a push is made beyond the max size, we get a stack overflow.

Typical situation of a stack overflow: execution stack in recursive calls.

Stack operations

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

Stack operations

STACK-EMPTY(S)

if $top[S] = 0$
  then return TRUE
  else return FALSE

PUSH(S, x)

$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow x$

Stack operations

```
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]
```

Stack operations

```
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]
```

Each operation takes $O(1)$ time.

Queue

A queue implements the **FIFO** (first-in, first-out) policy. The first element to be inserted is also the first one that is removed.

Queue

A queue implements the **FIFO** (first-in, first-out) policy. The first element to be inserted is also the first one that is removed.

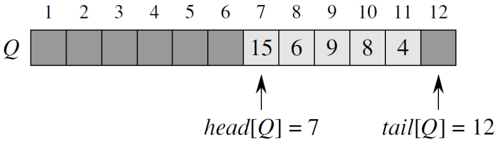
Queue

A queue implements the **FIFO** (first-in, first-out) policy. The first element to be inserted is also the first one that is removed.

↑ head ↑ tail

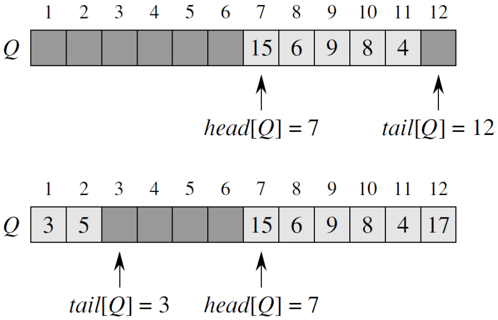
Queue

Assume we want to implement a queue using a fixed size array.
(In principle, we could avoid imposing a "length budget".)



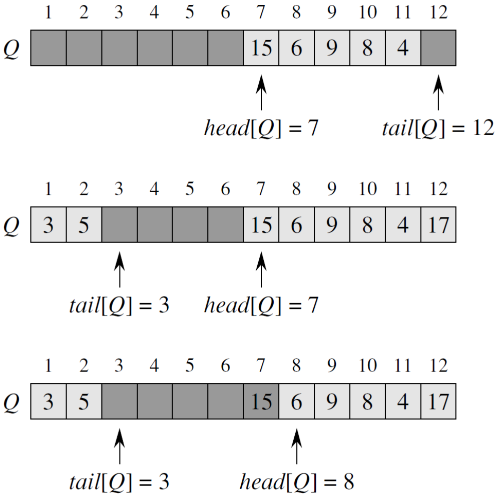
Queue

Assume we want to implement a queue using a fixed size array.
(In principle, we could avoid imposing a “length budget”.)

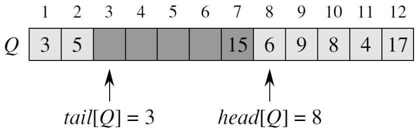


Queue

Assume we want to implement a queue using a fixed size array.
(In principle, we could avoid imposing a “length budget”.)

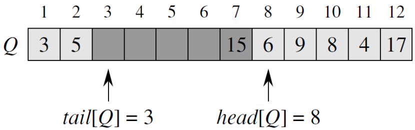


Queue



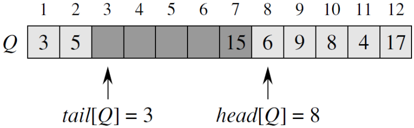
- The tail points to the next location to be filled in.

Queue



- The tail points to the next location to be filled in.
- For an array of capacity n, the queue has capacity $n-1$.

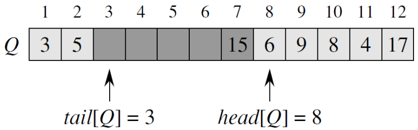
Queue



- The tail points to the next location to be filled in.
- For an array of capacity n, the queue has capacity $n-1$.
- In order, the elements are in the locations:

$$head[Q],\ head[Q]+1,\ \ldots,\ tail[Q]-1$$

Queue

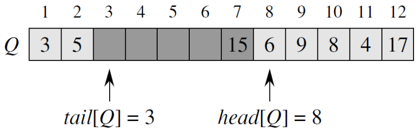


- The tail points to the next location to be filled in.
- For an array of capacity n, the queue has capacity $n-1$.
- In order, the elements are in the locations:

$$head[Q],\ head[Q]+1,\ \ldots,\ tail[Q]-1$$

- There is a circular shift at the boundaries of the container array. This way, we must not ensure that $head[Q]=1$ always.

Queue

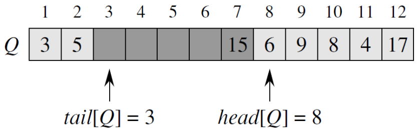


- The tail points to the next location to be filled in.
- For an array of capacity n, the queue has capacity $n - 1$.
- In order, the elements are in the locations:

$$head[Q],\ head[Q]+1,\ \ldots,\ tail[Q]-1$$

- There is a circular shift at the boundaries of the container array. This way, we must not ensure that $head[Q] = 1$ always.
- $head[Q] = tail[Q] \ \Rightarrow$ empty queue.

Queue



- The tail points to the next location to be filled in.
- For an array of capacity n, the queue has capacity $n-1$.
- In order, the elements are in the locations:

$$head[Q],\ head[Q]+1,\ \ldots,\ tail[Q]-1$$

- There is a circular shift at the boundaries of the container array. This way, we must not ensure that $head[Q]=1$ always.
- $head[Q]=tail[Q]\ \Rightarrow$ empty queue.
- $head[Q]=tail[Q]+1\ \Rightarrow$ full queue, risk of overflow.

Queue operations

ENQUEUE(Q, x)

$Q[tail[Q]] \leftarrow x$
if $tail[Q] = length[Q]$
  then $tail[Q] \leftarrow 1$
  else $tail[Q] \leftarrow tail[Q] + 1$

Queue operations

```
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
```

Queue operations

```
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
```

Each operation takes $O(1)$ time.

Linked list

An array is a sequence of objects arranged in linear order:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

From a computational perspective, an array is a contiguous structure.

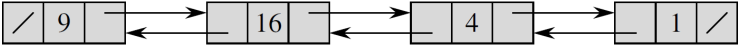
Linked list

An array is a sequence of objects arranged in linear order:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

From a computational perspective, an array is a contiguous structure.

In a linked list the ordering is determined by a pointer in each object:



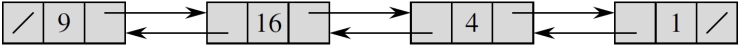
Linked list

An array is a sequence of objects arranged in linear order:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

From a computational perspective, an array is a contiguous structure.

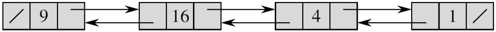
In a linked list the ordering is determined by a pointer in each object:



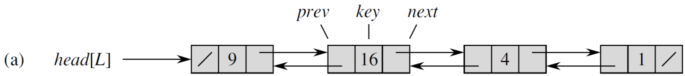
A linked list is not contiguous, as each element has its own context.

Linked list

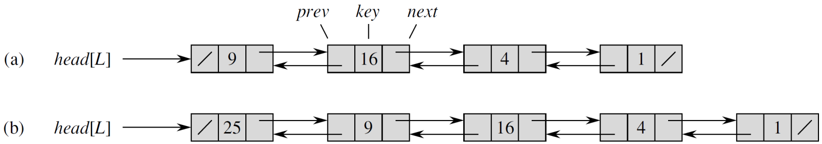
(a)



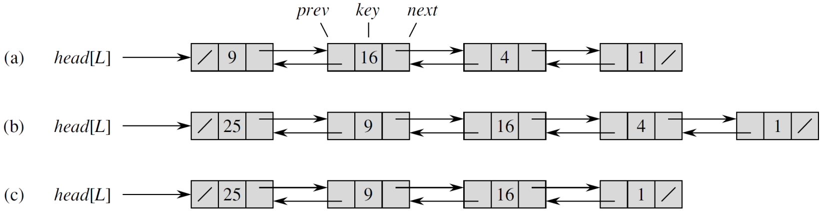
Linked list



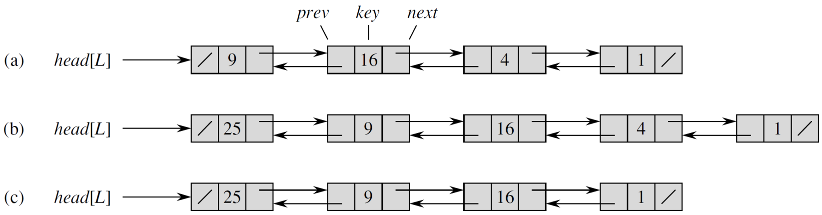
Linked list



Linked list

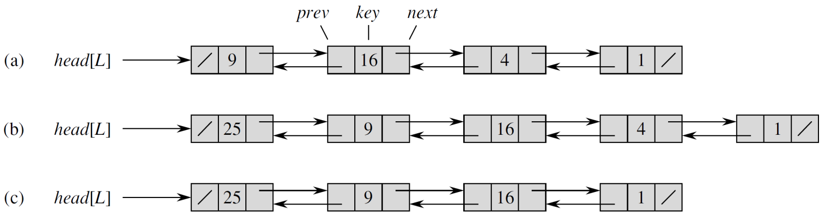


Linked list



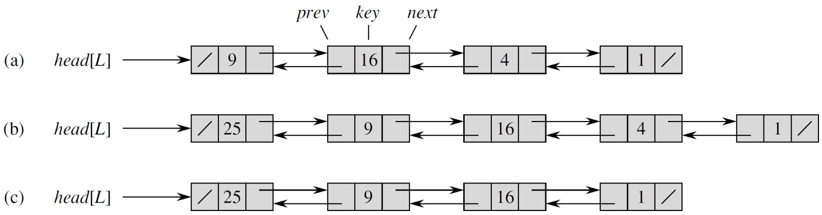
- A linked list usually has only $prev$ or only $next$ pointers.

Linked list



- A linked list usually has only $prev$ or only $next$ pointers.
- A doubly linked list has both.

Linked list



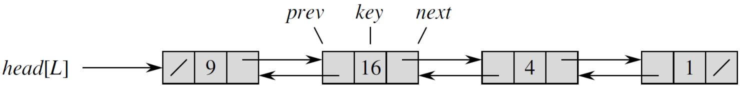
- A linked list usually has only $prev$ or only $next$ pointers.
- A doubly linked list has both.
- A circular list has the tail pointing to the head.

Linked list: Search

Look for element with key k, return a pointer to it.

LIST-SEARCH(L, k)

$x \leftarrow head[L]$
while $x \neq$ NIL and $key[x] \neq k$
    do $x \leftarrow next[x]$
return x

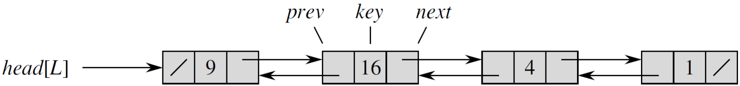


Linked list: Search

Look for element with key k, return a pointer to it.

LIST-SEARCH(L, k)

$x \leftarrow head[L]$
while $x \neq$ NIL and $key[x] \neq k$
  do $x \leftarrow next[x]$
return x



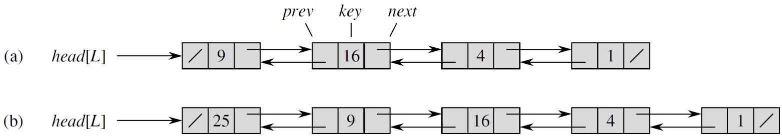
If the list has n objects, complexity is upper bounded as $O(n)$.

Linked list: Insert

Insert an element x at the front of the list.

LIST-INSERT(L, x)

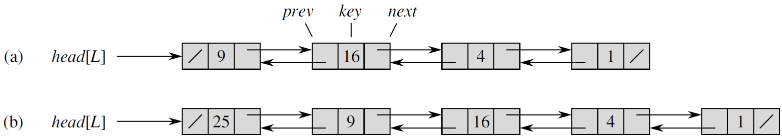
```
next[x] ← head[L]
if head[L] ≠ NIL
    then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```



Linked list: Insert

Insert an element x at the front of the list.

```
LIST-INSERT(L, x)
  next[x] ← head[L]
  if head[L] ≠ NIL
     then prev[head[L]] ← x
  head[L] ← x
  prev[x] ← NIL
```



Complexity is $O(1)$.

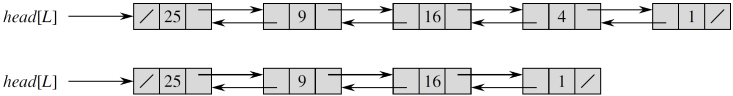
Linked list: Delete

Remove an element x from any location of the list.

```
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]
```

(b)

(c)

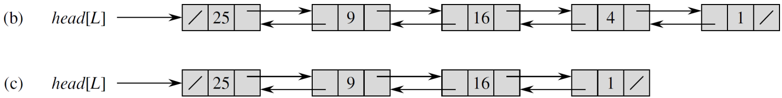


Linked list: Delete

Remove an element x from any location of the list.

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]
```

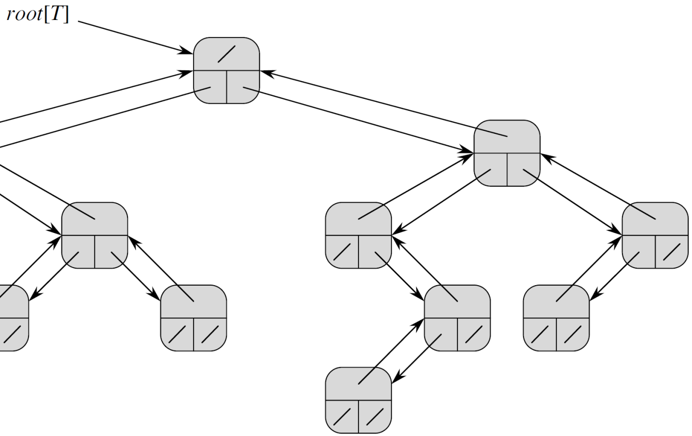


Complexity is $O(1)$.

Trees

Linked lists can be used to represent general trees.

For example, consider a binary tree:



Trees

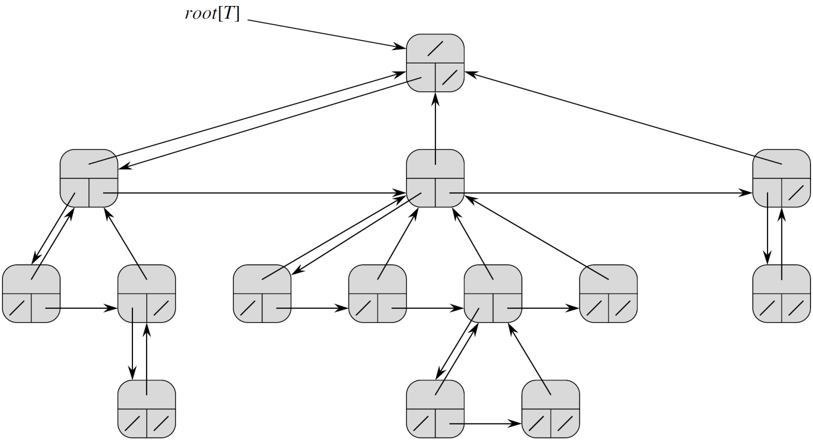
If the tree has > 2 children per node, use the following trick at each node:

- Store a pointer to the leftmost child.
- Store a pointer to the sibling to the right.

Trees

If the tree has > 2 children per node, use the following trick at each node:

- Store a pointer to the leftmost child.
- Store a pointer to the sibling to the right.



Exercises

- Write Python code implementing a tree with > 2 children, together with iterative or recursive code for traversing the tree.
- Using an array as the container, implement a deque (**d**ouble-**e**nded queue), which is similar to the queue, but allows insertion and deletion at both ends, so that each of these operations has $\Theta(1)$ complexity.

Send your solutions to rodola@di.uniroma1.it

Suggested reading

Chapters 10 Introduction, 10.1, 10.2 (skip the "Sentinels" paragraph), and 10.4 of:

"Introduction to Algorithms – 2nd Ed.", Cormen et al.