COM1009
Introduction to Algorithms and Data Structures

Topic 03: Elementary Data Structures

Essential Reading:

Chapter 10 up to page 238 (exclude Sentinels).

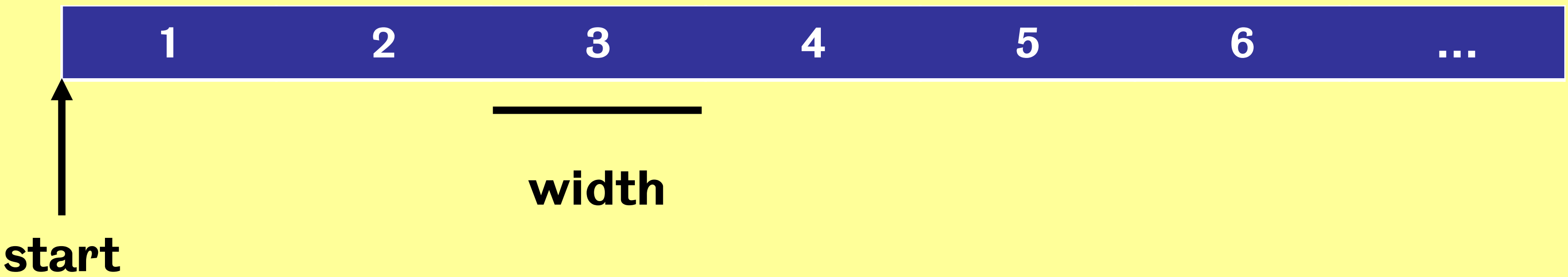
Aims of this (quick) section

- To introduce **data structures** and their typical operations.
- To revisit **stacks**, **queues**, and **linked lists**.
- To work out the **running time** for operations on these data structures.
- To identify pros and cons for data structures in terms of efficiency.

Data Structures

- **Dynamic sets** that can store and retrieve elements.
- Elements can contain **satellite data** and a **key** is used to identify the element
  - Often keys stem from a totally ordered set (e. g. numbers)
- Operations on dynamic sets S:
  - **Search(S, k):** returns element x with key k, or NIL
  - **Insert(S, x):** adds element x to S
  - **Delete(S, x):** removes element x from S
  - **Minimum(S), Maximum(S):** only for totally ordered sets
  - **Successor(S, x), Predecessor(S, x):** next or previous element
- **Time** often measured using n as the number of elements in S.

Arrays



Easy to access the n'th entry in an array

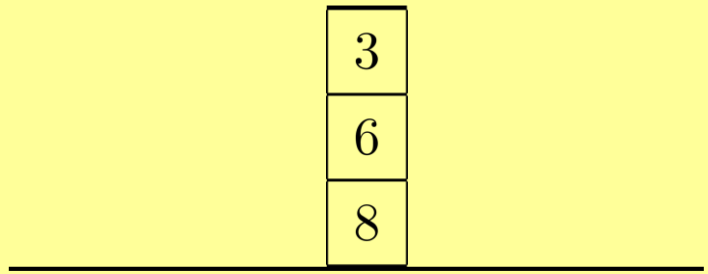
- Basically a single block of memory
- All values have the same type & size (*width*) in memory
- The 1st entry starts at memory location *start*
- The 2nd entry starts at memory location *start+width*
- The n'th entry starts at *start+(n-1)width*

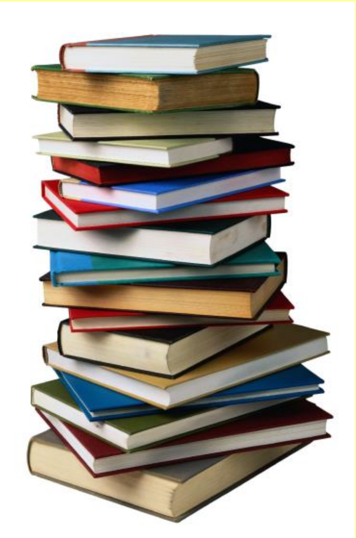
But the total capacity is fixed at creation

- To add extra values you need to create a new, bigger array and copy the existing values across

What is deletion?

Stacks





- Only the **top element** is accessible in a stack.
  - Last-in, first-out policy (LIFO)
- Insert is usually called **Push**, and Delete is called **Pop**.

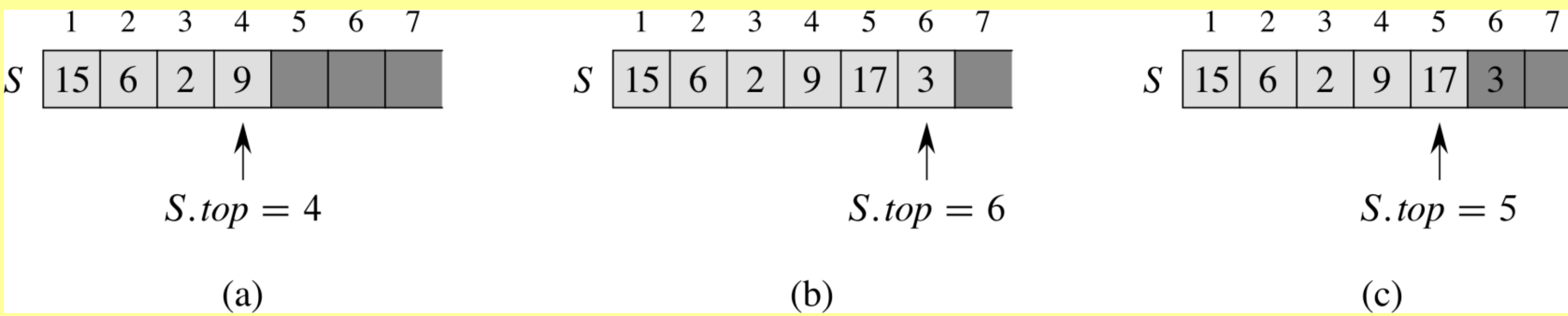
Stacks implemented using arrays

- Stacks can be implemented as an array S with attribute S.top.

```
STACK-EMPTY(S)
    return S.top == 0
```

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

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



All stack operations take time $O(1)$

▶ Queues

head	3	6	8	tail

- The first element in a queue is accessible.
  - First-in, first-out policy (FIFO)
- Insert is called Enqueue, Delete is called Dequeue.
- Queues have a **head** and a **tail**, like in a supermarket
  - Elements are added to the tail
  - Elements are extracted from the head

Queues implemented using arrays

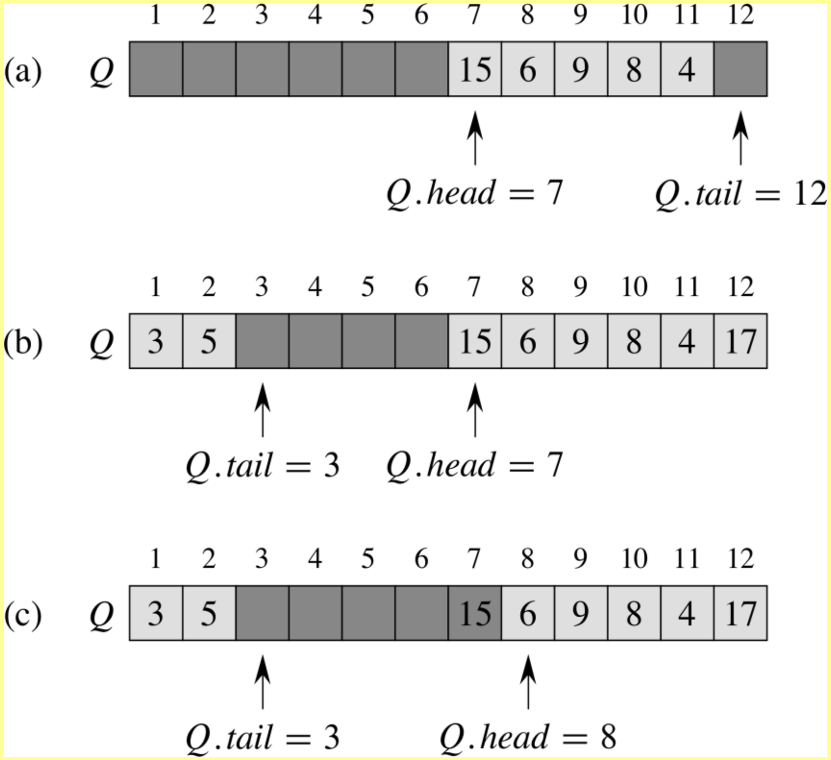
- Queues can be stored in an array "**wrapped around**".

ENQUEUE(Q, x)

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

DEQUEUE(Q)

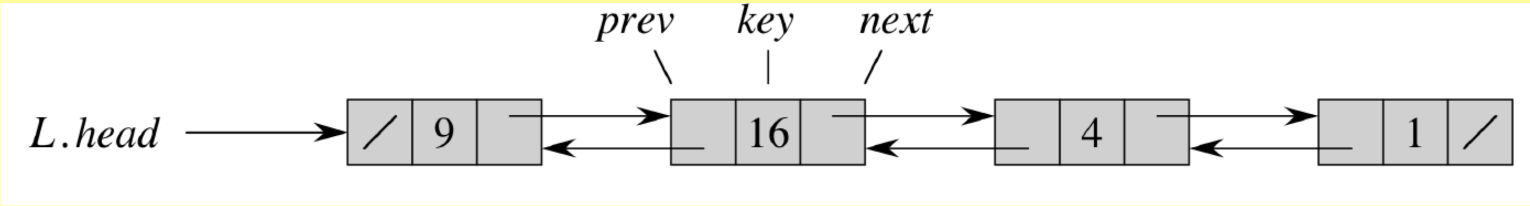
```
x = Q[Q.head]
if Q.head == Q.length then
    Q.head = 1
else
    Q.head = Q.head + 1
return x
```



All queue operations take time $O(1)$

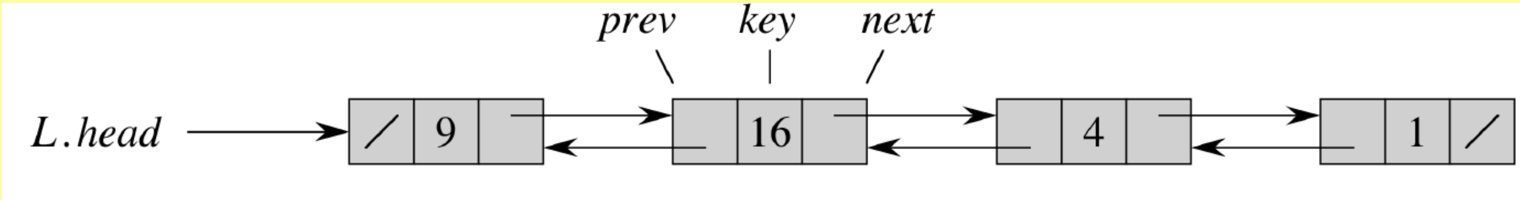
Linked Lists

- Objects are linked using **pointers to the next element**.
- Linked lists can be **singly linked** or **doubly linked**: pointers to next and previous elements.



- Each element x has attributes
  - x.key – the key used to identify the element
  - x.next – a pointer to the next element
  - x.prev – a pointer to the previous element
  - Optional: further satellite data

Linked Lists: Searching



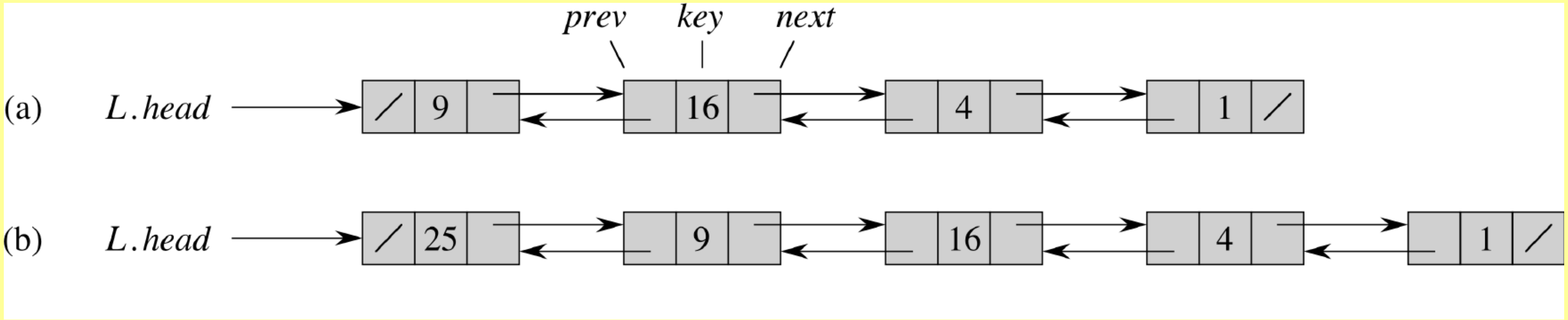
- Search inspects all elements in sequence and stops when the key has been found or the end of the list is reached.

List-Search(L, k)

```
x = L.head
while x ≠ NIL and x.key ≠ k do
    x = x.next
return x
```

- The worst-case time is $\Theta(n)$, since it may have to search the entire list.

Linked Lists: Inserting



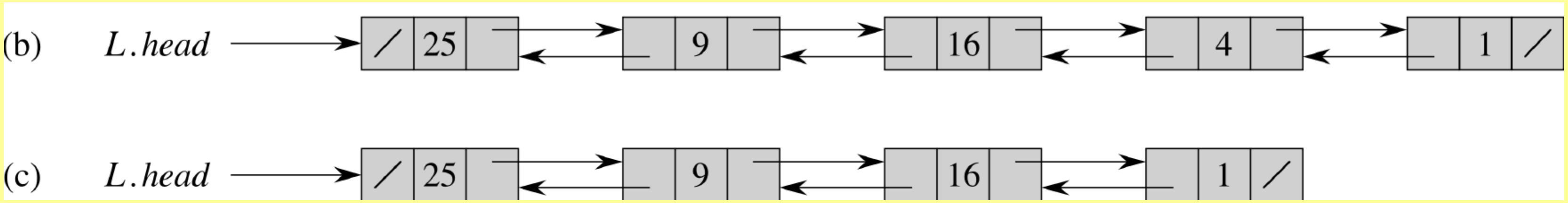
- New elements are added to the front of the list.

LIST-INSERT(L, x)

x.next $= L$.head
if L.head $\neq$ NIL **then**
    L.head.prev $= x$
L.head $= x$
x.prev $=$ NIL

- The time for an insertion is $O(1)$.

Linked Lists: Deleting



- If element x is known, update pointers to take it out.

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

- The time for a deletion is $O(1)$ if we know x (and hence its components). But if we only have the key … we need to search for element x, so it's $\Theta(n)$ in the worst case.

Summary

- Stacks and Queues are simple data structures that can
  - be implemented efficiently in arrays (ignoring space issues)
  - Have a restricted set of operations, but these run in time $O(1)$.
- Linked lists form an **unordered list** of elements
  - **Insertion** is fast: time $O(1)$.
  - **Searching** takes worst-case time $\Theta(n)$.
  - **Deletion** runs in time $O(1)$ if the element is known, otherwise we need to run a search beforehand and incur time $\Theta(n)$.
  - Linked lists can be **doubly linked**: finding **successors** and **predecessors** in time $O(1)$.