# Data structures

## Abstract Data Types (ADT's)

- An ADT is a specification of
  - a set of data;
  - a set of operations that ca be performed on the data.
    - This operation set is the focus
- ADT is abstract in the sense that it is independent of various concrete implementations.
  - Encapsulates data structures and relevant algorithms.
  - Provides access interface.

## Data abstractions

- ADT: An entity + opeations that can be performed on it
  - E.g., integer.
  - addition, subtraction, negation, multiplication, division, comparison.
- Advantages
  - Users only need to know what the allowed operations are, not how they are implemented.
  - You can choose the best possible implementation for an ADT when you need to use it.
    - Allows to separate the concerns of correctness and the performance analysis of your algorithms.
- We will learn ADTs of fundamental data structures
  - Arrays, stracks, queues, etc.
  - Focus on operations.

## Array

Typically, occupies sequential storage locations. Length is determined when created. Each element has a fixed unique index. Any element of the array can be visited or updated by using its index. $O(1)$ = constant time for accessing array. Can be used for implementing other structures like stacks and queues.

## Stack

- Intuitive view: a pile of things on top of each other.
- An object added to the stack goes on the "top" of the stack (**push**)
- An object removed from the stack is taken from the "top" (**pop**)
- LIFO: last in first out.

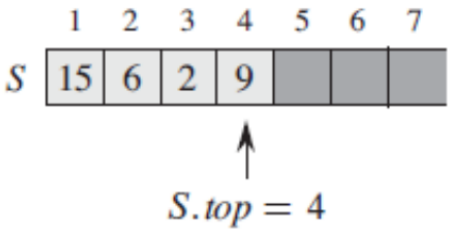
## Stack ADT

- A stack is a container of object. They are removed and added according to the LIFO.
  - Sequence of elements $< a_1, a_2, .., a_i >$, but only $a_i$ is accessible as the “top” of the stack.
- Operations:

- Push(S, x) inserts an element x into the stack S.
- Pop(S) deletes the element on top of the stack S. (Does not take an element argument.)
- Stack-Empty(S) return whether the stack is empty.

### Stack: an array implementation

- Using an array of n elements: S[1..n]
- The array has an attribute *S.top* that indexes the most recently inserted element.
- The stack consists of elements S[1..S.top].
  - S[1] is the element at the bottom
  - S[S.top] is the element at the top.
- Consider a stack with at most 7 elements and after inserting 15, 6, 2 and 9, we have the following:




- Stack-Empty(S)

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

- Push(S, x)
  - Overflow issue

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

- Pop(S)

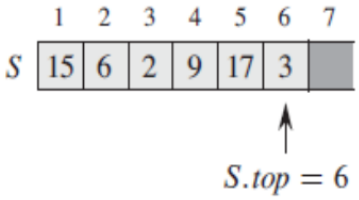
```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

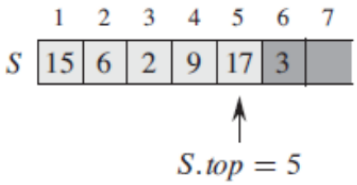
***Complexity:***

All operations take constant time.
Does not relate to the size of the stack n.

With the stack array: $S[] = 15, 6, 2, 9, 17$

- What does the array looks like after calling Push(S, 3).
- What does the array looks like after calling Pop(S).




The stack consists of elements S[1..S.top]. Although 3 is in the array, 3 is not part of the stack.



## Stack: Application of stack

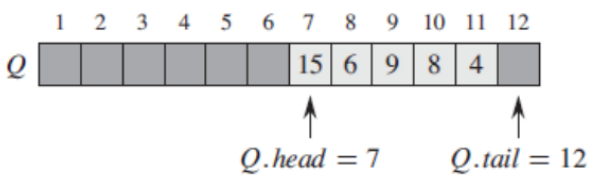
- Parenthesis balance checking: $(2 + 3) - 6) * 2$
- Push when you see "(" and pop when you see ")"
- This example will stop when you hit the second ")" because you wanna pop and there is nothing to pop. The stack should also be empty when done for the check to return true.

## Queue + ADT

- Intuitive view: a real-life queue.
- FIFO: first in first out.
- A queue is a container of elements. Elements are inserted an removed according to the FIFO principle.
  - Sequence of elements $< a_1, a_2, .., a_i >$, but only $a_i$ is accessible as the *head* of the queue.
- Operations
  - Enqueue(Q, x) inserts an element x into the queue Q.
  - Dequeue(Q) deletes the head element in the queue Q. (Does not take an element argument)

## Queue: An array implementation

- Implement a queue of at most n-1 elements using an array Q[1..n]
- The array has attribute **Q.head** that indexes the head element, i.e., the one has been in the queue the longest.
- The array has attribute **Q.tail** that indexes the next location at which a newly arriving element will be inserted in the queue.
- The elements in the queue reside in locations Q.head, Q.head+1, ..., Q.tail-1, if we assume location 1 immediately follows location n in a circular order.
- Initially, we have **Q.head = Q.tail**.
- Example:



- Enqueue:
  - Issues for overflow.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3      Q.tail = 1
4  else Q.tail = Q.tail + 1
```

*We assume location 1 immediately follows location n in a circular order.*

- Dequeue:
  - Issues for underflow.

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3      Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

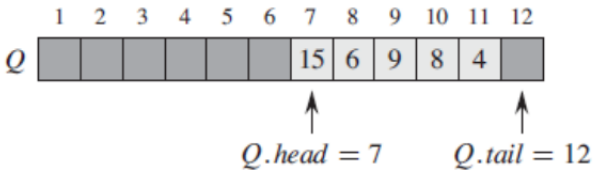
*We assume location 1 immediately follows location n in a circular order.*

***Complexity:***

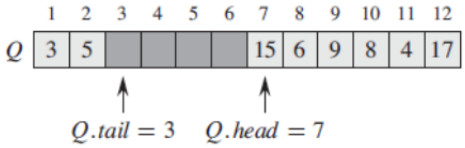
All operations take constant time.
Does not relate to the size of the queue n.

**Queue: Example**

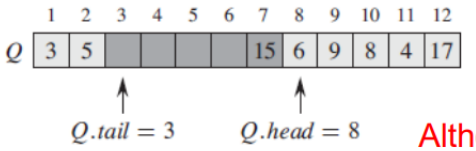
- Do the following on




- Enqueue(Q, 17), Enqueue(Q, 3), and Enqueue(Q,5)




- Dequeue(Q)



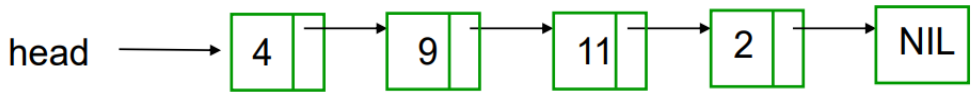

Although 15 is still in the array, it does not belong to the queue anymore.

**Linked List**

- Intuitive view: a sequence of elements.
- Each element in the lined list is with:
  - On key
  - One or more pointers
- There are different types of linked lists depending on how the elements in the list are linked.

**Linked list: Singly linked list**

- Element
  - A key
  - One pointer: "**next**", which points to the element's successor.
  - The last element's pointer points to a "NIL".
- On additional pointer for the whole list
  - Pointer "**head**": pointing to the first element of the list.

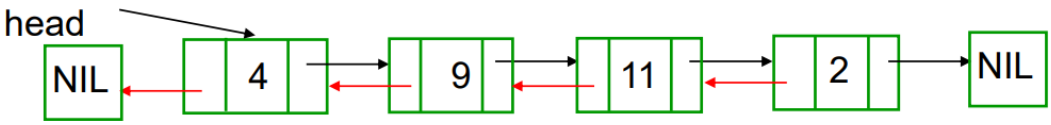



Note that; "next" pointers are for elements. E.g., x.next. "head" pointer is for the whole list. E.g., L.head.

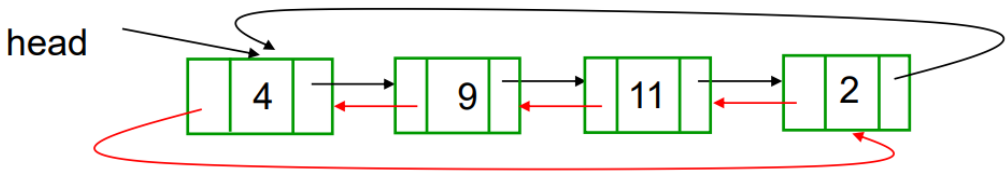
- An alternative: circular list
  - The last element's next pointer points to the first element. This gives a circular structure.

## Linked list: Doubly linked list

- Element
  - A key
  - Two pointers: "**next**", which points to the element's successor.
  - "**prev**", which points to the elements predecessor.
- Also has the "head" pointer for the whole list.




- An alternative: circular list
  - The last elements **next** pointer points to the first element.
  - The first elements **prev** pointer points to the last element.
  - This gives again a circular structure.




28

## Operations on doubly linked lists

- Search
  - Given a key *k*, finds the first element with a key *k* in list *L*.
- Insertion
  - Insert an element *x* in list *L*.
- Deletion
  - Delete an element x from list L.
- Note that for insertion and deletion, it is insert/delete an element x, not an element with key x.

## Implementation: Search

- Searching: linear search the list.

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

Runtime = $\Theta(n)$, we have to search the entire list.

## Implementations: Insert

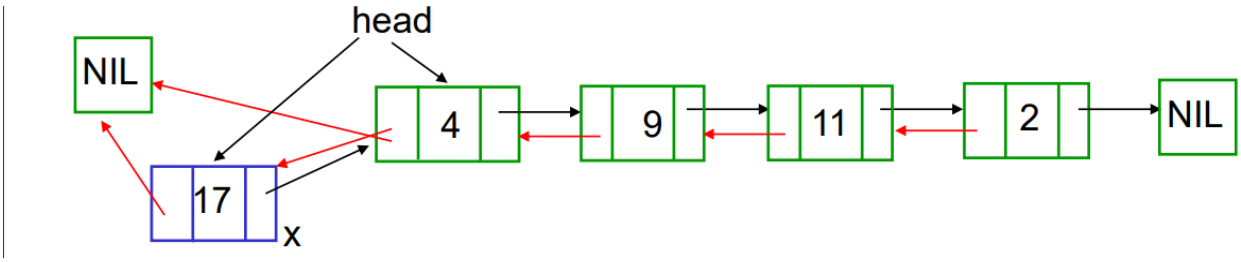
- Insertion: insert element x onto the front of the linked list L.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

Can you switch line 1 and lines 2 and 3?

Can you move line 4 to the top?

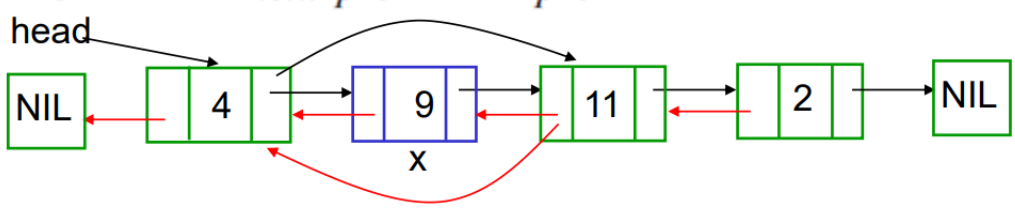
line 2-4 skal være i den rækkefølge og line 1 skal være før line 4.




Running time = $\Theta(1)$ Contant time.

## Implementations: delete

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```




Running time = $\Theta(1)$ Contant time.

## Mini quiz on Moodle

From exam 2016

> **7** (*5 points*) Consider a **doubly linked list** where each element has two pointers `next` and `prev`. Given an element $p$ that is already in the doubly linked list, which of the following operations can insert a new element $s$ after $p$ into the doubly linked list?
>
> ☐ a) p.next=s; s.prev=p; p.next.prev=s; s.next=p.next;
> ☐ b) p.next=s; p.next.prev=s; s.prev=p; s.next=p.next;
> ☐ c) s.prev=p; s.next=p.next; p.next=s; p.next.prev=s;
> ☒ d) s.prev=p; s.next=p.next; p.next.prev=s; p.next=s;

## Mini quiz

- Can we use a linked list to implement a stack?
  - Push: add an element onto the front of the linked list.
  - Pop: delete the element pointed by the "head" pointer.
- Are the push and pop operations still constant time?
  - Yes, because add and delete operations of a linked list are also constant time.

| | **Push** | **Pop** |
|---|---|---|
| Array | $\Theta(1)$ | $\Theta(1)$ |
| Linked List | $\Theta(1)$ | $\Theta(1)$ |

- This is why Stack is an ADT, which can be implemented by different means, e.g., using arrays or linked lists.

## Priority Queues

- A priority queue (PQ) is a container for maintaining a set A of elements, each with an associated value called key.
- A PQ supports the following operations
  - Insert(A, x) insert element x in set A (A=A∪{x})
  - Maximum(A) returns the element of A with the largest key (i.e., highest priority)
  - extract-Max(A) returns and removes the element of A with the largest key from A

## A Max Heap implementation of Priority Queues

- **Maximum**(A)

```
Heap-Maximum(A)
01 return A[1]
```

- Takes constant time.
- **Extract-Max**(A)

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

- Analysis
  - Max-Heapify takes Θ(lgn), the remaining lines take constant time.
  - Thus, in total, Θ(lgn).
- **Insert**(A,key)
- Enlarge the heap and propagate the new element from last place "up" the heap.

```
Heap-Insert(A, key)
01 A.heapsize = A.heapsize + 1
02 i = A.heapsize
03 A[i] = key
04 while i > 1 and A[Parent(i)] < A[i] do
05   Exchange A[i] with A[Parent(i)]
06   i= Parent(i)
```

- Run time of Heap-Insert = $\Theta(lgn)$

## Priority Queue

- Applications:
  - Job scheduling shared computing resources
    - Call Extract-Max to select the highest-priority job from those pending jobs.
    - Call Insert to insert a new job to the queue.
  - As a building block for other algorithms
    - Dijkstra's algorithm for computing shortest routes in a graph.
- We used a heap to implement priority queues. Other implementations are possible.
  - Can we implement a priority queue using a (singly) linked list?
  - What are the complexities of the 3 operations?

## Different implementations

| | Maximum | Extract-Max | Insert |
|---|---|---|---|
| Heap | $\Theta(1)$ | $\Theta(lgn)$ | $\Theta(lgn)$ |
| Singly Linked List | $\Theta(n)$ | $\Theta(n)$ | $\Theta(1)$ |
| Ordered Singly Linked List | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |

Consider a specific workload: you first build a priority queue with n elements by keep inserting the n elements and then Extract-Max half of the elements. Which implementation will be the most efficient?

| | Maximum | Extract-Max | Insert |
|---|---|---|---|
| Heap | $\Theta(lgn) * n$ | $\Theta(lgn) * n/2$ | $\Theta(nlgn)$ |
| Singly Linked List | $\Theta(1) * n$ | $\Theta(n) * n/2$ | $\Theta(n^2)$ |
| Ordered Singly Linked List | $\Theta(n) * n$ | $\Theta(1) * n/2$ | $\Theta(n^2)$ |

## ADT vs. Data Structure

- Abstract vs. concrete
- Specification vs. implementation
    - Stack
        - Specification of operations: push, pop.
        - Implementation: array, linked list.
    - Priority queue
        - Specification of operations: insert, maximum, extract-Max
        - Implementation: heap, linked list.