Algorithms and Data Structures for Mathematicians

Lecture 3: Basic Data Structures

Peter Kostolányi
kostolanyi at fmph and so on
Room M-258

12 October 2017

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain “objects”
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Elements of dynamic sets:

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Elements of dynamic sets:

- We shall think of them as of objects in a programming language

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Elements of dynamic sets:

- We shall think of them as of objects in a programming language
- Elements can have attributes; we shall usually assume that each element x has an attribute $x.key$

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Elements of dynamic sets:

- We shall think of them as of objects in a programming language
- Elements can have attributes; we shall usually assume that each element x has an attribute $x.\mathit{key}$
- This allows us to realise multisets

Dynamic Sets and Data Structures

- Many algorithms need to keep record of sets of certain "objects"
- These sets usually have to be modified in time by operations such as insertion, deletion, etc.
- Requirements for supported operations may vary
- Dynamic sets: sets that can be modified in time by one of several operations specified
- Data structures: "realisations" (or "almost-implementations") of dynamic sets usually aiming at efficiency

Elements of dynamic sets:

- We shall think of them as of objects in a programming language
- Elements can have attributes; we shall usually assume that each element x has an attribute $x.\mathit{key}$
- This allows us to realise multisets
- Elements might not be implemented as objects, but we shall always think of them as pointers (not values)

Typical Operations on Dynamic Sets

Modifying operations:

Typical Operations on Dynamic Sets

Modifying operations:

- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Queries:

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Queries:

- $\text{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Queries:

- $\text{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\text{EMPTY}(X)$: returns `true` if X is empty

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Queries:

- $\text{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\text{EMPTY}(X)$: returns `true` if X is empty
- $\text{MIN}(X)$: returns some element x of X with minimal $x.key$ (only if keys are taken from a totally ordered set)

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- $\ldots$

Queries:

- $\text{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\text{EMPTY}(X)$: returns `true` if X is empty
- $\text{MIN}(X)$: returns some element x of X with minimal $x.key$ (only if keys are taken from a totally ordered set)
- $\text{SUCC}(X, x)$: returns a successor y of x in some total ordering of keys

Typical Operations on Dynamic Sets

Modifying operations:

- $\text{INSERT}(X, x)$: inserts x into a dynamic set X
- $\text{DELETE}(X, x)$: deletes x from X
- ...

Queries:

- $\text{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\text{EMPTY}(X)$: returns `true` if X is empty
- $\text{MIN}(X)$: returns some element x of X with minimal $x.key$ (only if keys are taken from a totally ordered set)
- $\text{SUCC}(X, x)$: returns a successor y of x in some total ordering of keys
- ...

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, ...)

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, ...)

Data structures are "realisations" of dynamic sets:

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, ...)

Data structures are "realisations" of dynamic sets:

- Aim is to realise the supported operations efficiently

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, ...)

Data structures are “realisations” of dynamic sets:

- Aim is to realise the supported operations efficiently
- There is no optimal data structure for all operations

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, . . .)

Data structures are "realisations" of dynamic sets:

- Aim is to realise the supported operations efficiently
- There is no optimal data structure for all operations
- The choice of a data structure thus depends on the type of the dynamic set realised

Dynamic Sets and Data Structures

By a type of a dynamic set, we shall understand:

- The collection of supported operations
- Programming terminology: abstract data types
- Most frequent ones have their own names (e.g., dictionaries, ...)

Data structures are "realisations" of dynamic sets:

- Aim is to realise the supported operations efficiently
- There is no optimal data structure for all operations
- The choice of a data structure thus depends on the type of the dynamic set realised
- Examples: arrays, linked lists, heaps, trees, ...

Dictionaries

Dynamic sets supporting the following operations:

Dictionaries

Dynamic sets supporting the following operations:

- SEARCH(D, k): returns some element x of D such that $x.key = k$ (if there is at least one such element)

Dictionaries

Dynamic sets supporting the following operations:

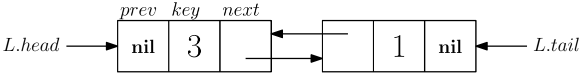
- $\text{SEARCH}(D, k)$: returns some element x of D such that $x.key = k$ (if there is at least one such element)
- $\text{INSERT}(D, x)$: inserts x into D

Dictionaries

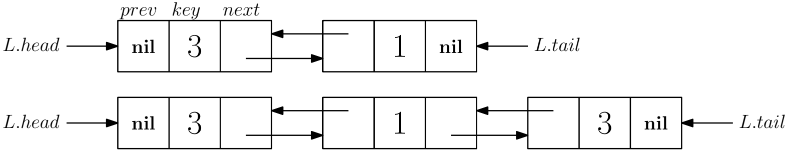
Dynamic sets supporting the following operations:

- $\text{SEARCH}(D, k)$: returns some element x of D such that $x.key = k$ (if there is at least one such element)
- $\text{INSERT}(D, x)$: inserts x into D
- $\text{DELETE}(D, x)$: deletes x from D

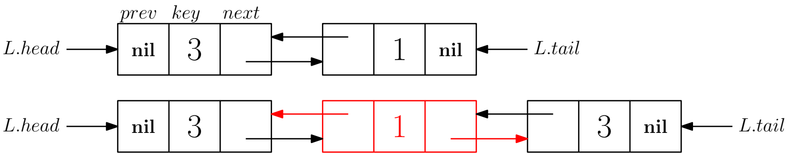
Dictionaries via (Doubly) Linked Lists



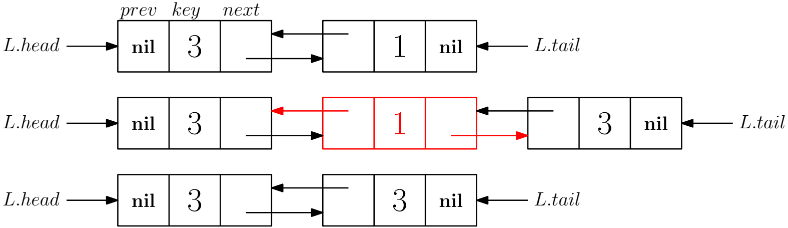
Dictionaries via (Doubly) Linked Lists



Dictionaries via (Doubly) Linked Lists



Dictionaries via (Doubly) Linked Lists



Dictionaries via (Doubly) Linked Lists

$\text{LISTINSERT}(L, x)$:

Dictionaries via (Doubly) Linked Lists

LISTINSERT(L, x):

Input : A linked list L; x not in L

Behaviour: Inserts x to L

```
x.prev ← L.tail;
x.next ← nil;
if L.tail ≠ nil then
 |  L.tail.next ← x
end
else
 |  L.head ← x
end
L.tail ← x;
```

Dictionaries via (Doubly) Linked Lists

LISTINSERT(L, x):

Input : A linked list L; x not in L

Behaviour: Inserts x to L

$x.prev \leftarrow L.tail$;
$x.next \leftarrow$ **nil**;
if $L.tail \neq$ **nil then**
  $L.tail.next \leftarrow x$
end
else
  $L.head \leftarrow x$
end
$L.tail \leftarrow x$;

- Time complexity: $\Theta(1)$

Dictionaries via (Doubly) Linked Lists

LISTDELETE(L, x):

Dictionaries via (Doubly) Linked Lists

LISTDELETE(L, x):

Input : A linked list L; x in L
Behaviour: Deletes x from L

if $x.prev \neq$ **nil then**
  $x.prev.next \leftarrow x.next$
end
else
  $L.head \leftarrow x.next$
end
if $x.next \neq$ **nil then**
  $x.next.prev \leftarrow x.prev$
end
else
  $L.tail \leftarrow x.prev$
end

Dictionaries via (Doubly) Linked Lists

LISTDELETE(L, x):

Input : A linked list L; x in L
Behaviour: Deletes x from L

if $x.prev \neq$ **nil then**
| $x.prev.next \leftarrow x.next$
end
else
| $L.head \leftarrow x.next$
end
if $x.next \neq$ **nil then**
| $x.next.prev \leftarrow x.prev$
end
else
| $L.tail \leftarrow x.prev$
end

- Time complexity: $\Theta(1)$

Dictionaries via (Doubly) Linked Lists

LISTSEARCH(L, k):

Dictionaries via (Doubly) Linked Lists

LISTSEARCH(L, k):

Input : A linked list L; a key k
Output: First x in L such that $x.key = k$ or **nil** if there is no such x

$x \leftarrow L.head$;
while $x \neq$ **nil** *and* $x.key \neq k$ **do**
  $x \leftarrow x.next$
end
return x;

Dictionaries via (Doubly) Linked Lists

LISTSEARCH(L, k):

Input : A linked list L; a key k

Output: First x in L such that $x.key = k$ or **nil** if there is no such x

$x \leftarrow L.head$;
while $x \neq$ **nil** *and* $x.key \neq k$ **do**
    $x \leftarrow x.next$
end
return x;

- Worst-case time complexity: $\Theta(n)$, where n is the number of elements in L

Stacks

Dynamic sets supporting the following operations:

Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S

Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

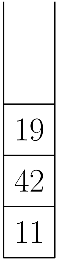
LIFO = Last In First Out

Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out

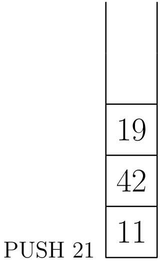


Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out

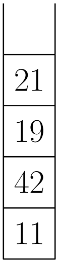


Stacks

Dynamic sets supporting the following operations:

- $\mathrm{PUSH}(S, x)$: inserts x into S
- $\mathrm{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out

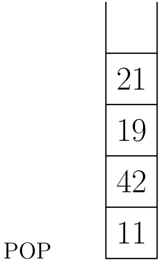


Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out

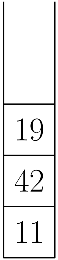


Stacks

Dynamic sets supporting the following operations:

- PUSH(S, x): inserts x into S
- POP(S): removes and returns the last inserted x

LIFO = Last In First Out

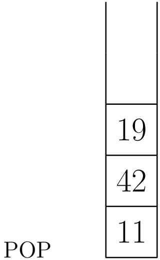


Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out

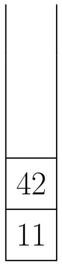


Stacks

Dynamic sets supporting the following operations:

- $\text{PUSH}(S, x)$: inserts x into S
- $\text{POP}(S)$: removes and returns the last inserted x

LIFO = Last In First Out



Stacks via Arrays

We shall represent a stack as an "object" S containing:

Stacks via Arrays

We shall represent a stack as an “object” S containing:

- A (dynamical) array $S.a$ containing elements of the stack

Stacks via Arrays

We shall represent a stack as an "object" S containing:

- A (dynamical) array $S.a$ containing elements of the stack
- An integer $S.num$ representing the number of elements on the stack

Stacks via Arrays

We shall represent a stack as an "object" S containing:

- A (dynamical) array $S.a$ containing elements of the stack
- An integer $S.num$ representing the number of elements on the stack

PUSH(S, x):

Stacks via Arrays

We shall represent a stack as an "object" S containing:

- A (dynamical) array $S.a$ containing elements of the stack
- An integer $S.num$ representing the number of elements on the stack

PUSH(S, x):

Input : A stack S; an element x
Behaviour: Places x on the top of the stack

$S.num \leftarrow S.num + 1;$
$S.a[S.num] \leftarrow x;$

Stacks via Arrays

We shall represent a stack as an "object" S containing:

- A (dynamical) array $S.a$ containing elements of the stack
- An integer $S.num$ representing the number of elements on the stack

PUSH(S, x):

Input : A stack S; an element x
Behaviour: Places x on the top of the stack

$S.num \leftarrow S.num + 1$;
$S.a[S.num] \leftarrow x$;

POP(S):

Stacks via Arrays

We shall represent a stack as an "object" S containing:

- A (dynamical) array $S.a$ containing elements of the stack
- An integer $S.num$ representing the number of elements on the stack

PUSH(S, x):

Input : A stack S; an element x
Behaviour: Places x on the top of the stack

$S.num \leftarrow S.num + 1$;
$S.a[S.num] \leftarrow x$;

POP(S):

Input : A stack S
Output: An element x on the top of the stack, which is removed from S

```
if S.num = 0 then
 |  error(underflow)
end
else
 |  S.num ← S.num − 1;
 |  return S.a[S.num + 1];
end
```

Queues

Dynamic sets supporting the following operations:

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

$$19 \quad 16 \quad 37 \quad 42 \quad 10$$

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

$19 \quad 16 \quad 37 \quad 42 \quad 10$ ENQUEUE 33

Queues

Dynamic sets supporting the following operations:

- ENQUEUE(Q, x): inserts x into Q
- DEQUEUE(Q): removes and returns x that is in Q for the longest time

FIFO = First In First Out

19	16	37	42	10		ENQUEUE 33
19	16	37	42	10	33	

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

19	16	37	42	10		ENQUEUE 33
19	16	37	42	10	33	DEQUEUE

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

19	16	37	42	10		ENQUEUE 33
19	16	37	42	10	33	DEQUEUE
16	37	42	10	33		

Queues

Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

FIFO = First In First Out

19	16	37	42	10		ENQUEUE 33
19	16	37	42	10	33	DEQUEUE
16	37	42	10	33		DEQUEUE

Queues

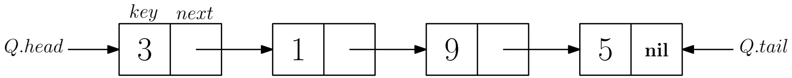
Dynamic sets supporting the following operations:

- $\text{ENQUEUE}(Q, x)$: inserts x into Q
- $\text{DEQUEUE}(Q)$: removes and returns x that is in Q for the longest time

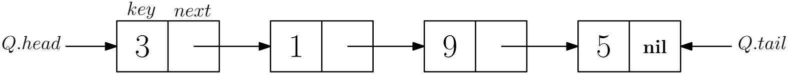
FIFO = First In First Out

19	16	37	42	10		ENQUEUE 33
19	16	37	42	10	33	DEQUEUE
16	37	42	10	33		DEQUEUE
37	42	10	33			

Queues via Singly Linked Lists

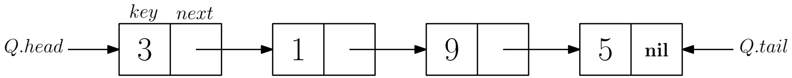


Queues via Singly Linked Lists



ENQUEUE(Q, x):

Queues via Singly Linked Lists

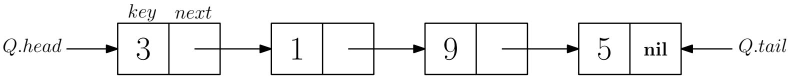


ENQUEUE(Q, x):

Input : A queue Q; x not in Q
Behaviour: Inserts x to Q

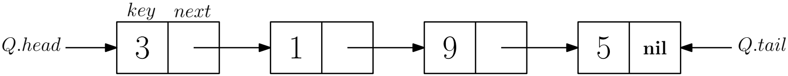
$x.next \leftarrow$ **nil**;
if $Q.tail \neq$ **nil then**
| $Q.tail.next \leftarrow x$
end
else
| $Q.head \leftarrow x$
end
$Q.tail \leftarrow x$;

Queues via Singly Linked Lists



DEQUEUE(Q):

Queues via Singly Linked Lists



DEQUEUE(Q):

Input : A queue Q

Output: The element x inserted to Q the longest time ago, which is removed from Q

$x \leftarrow Q.head$;
if $x \neq$ **nil then**
| $Q.head \leftarrow x.next$
end
if $Q.head =$ **nil then**
| $Q.tail \leftarrow$ **nil**
end
return x;

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.\mathit{key}$ from a totally ordered set $(S, \preceq)$):

- $\mathrm{INSERT}(Q, x)$: inserts x into Q

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

- $\mathrm{INSERT}(Q, x)$: inserts x into Q
- $\mathrm{MAX}(Q)$: returns some x in Q with maximal $x.key$

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

- $\mathrm{INSERT}(Q, x)$: inserts x into Q
- $\mathrm{MAX}(Q)$: returns some x in Q with maximal $x.key$
- $\mathrm{EXTRACTMAX}(Q)$: removes and returns some x in Q with maximal $x.key$

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

- $\mathrm{INSERT}(Q, x)$: inserts x into Q
- $\mathrm{MAX}(Q)$: returns some x in Q with maximal $x.key$
- $\mathrm{EXTRACTMAX}(Q)$: removes and returns some x in Q with maximal $x.key$
- $\mathrm{INCREASEKEY}(Q, x, k)$: assuming that k is greater than $x.key$, changes $x.key$ to k

Priority Queues

Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

- $\mathrm{INSERT}(Q, x)$: inserts x into Q
- $\mathrm{MAX}(Q)$: returns some x in Q with maximal $x.key$
- $\mathrm{EXTRACTMAX}(Q)$: removes and returns some x in Q with maximal $x.key$
- $\mathrm{INCREASEKEY}(Q, x, k)$: assuming that k is greater than $x.key$, changes $x.key$ to k

We shall realise priority queues via data structures called max-heaps

Max-Heaps

- "Nearly complete" binary trees

Max-Heaps

- "Nearly complete" binary trees
- Each level except the last one is complete

Max-Heaps

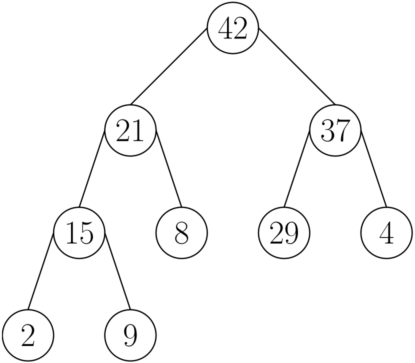
- "Nearly complete" binary trees
- Each level except the last one is complete
- The last level is complete from left up to some point

Max-Heaps

- "Nearly complete" binary trees
- Each level except the last one is complete
- The last level is complete from left up to some point
- Each parent has a larger (or equal) key than any of its children

Max-Heaps

- "Nearly complete" binary trees
- Each level except the last one is complete
- The last level is complete from left up to some point
- Each parent has a larger (or equal) key than any of its children

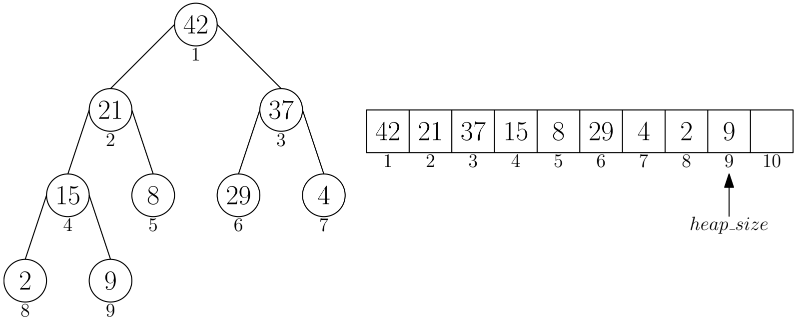


Max-Heaps

- We shall represent max-heaps using arrays

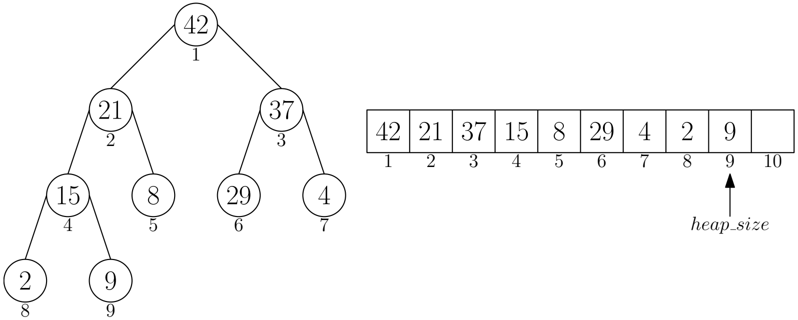
Max-Heaps

- We shall represent max-heaps using arrays



Max-Heaps

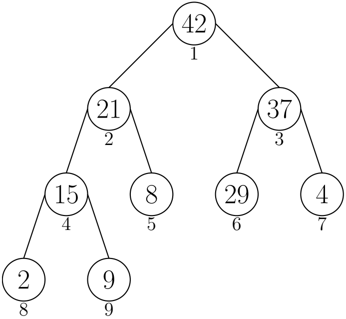
- We shall represent max-heaps using arrays

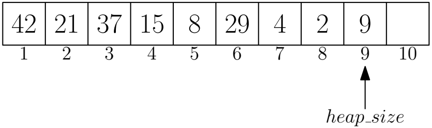


PARENT(i):

Max-Heaps

- We shall represent max-heaps using arrays



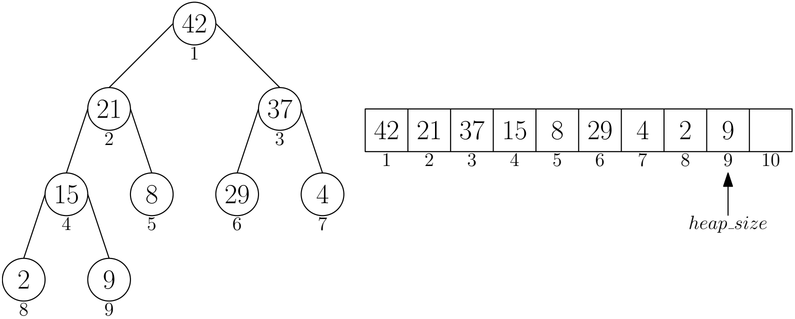


PARENT(i):

return $\lfloor i/2 \rfloor$;

Max-Heaps

- We shall represent max-heaps using arrays



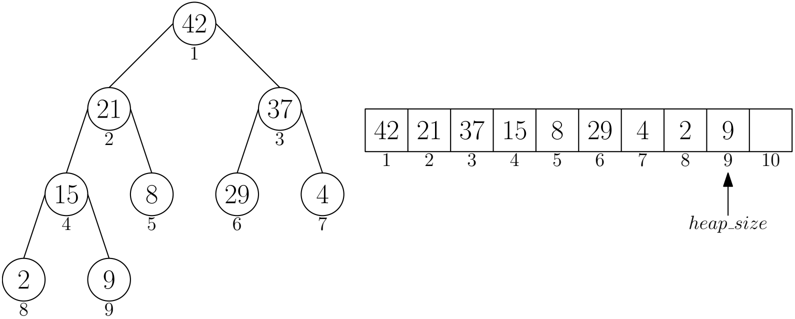
PARENT(i):

return $\lfloor i/2 \rfloor$;

LEFT(i):

Max-Heaps

- We shall represent max-heaps using arrays



PARENT(i):

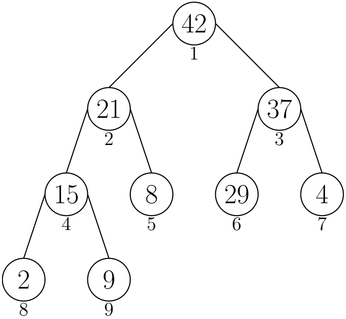
return $\lfloor i/2 \rfloor$;

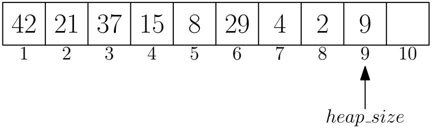
LEFT(i):

return $2i$;

Max-Heaps

- We shall represent max-heaps using arrays





PARENT(i):

return $\lfloor i/2 \rfloor$;

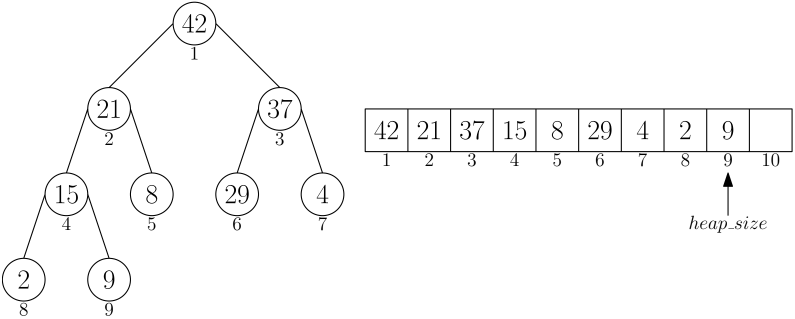
LEFT(i):

return $2i$;

RIGHT(i):

Max-Heaps

- We shall represent max-heaps using arrays



PARENT(i):

return $\lfloor i/2 \rfloor$;

LEFT(i):

return $2i$;

RIGHT(i):

return $2i + 1$;

Max-Heaps

- We shall describe an operation $\text{HEAPIFY}(H, i)$ that maintains the heap property at index i

Max-Heaps

- We shall describe an operation $\text{HEAPIFY}(H, i)$ that maintains the heap property at index i
- Assumption: both children of the i-th node are roots of valid heaps

Max-Heaps

- We shall describe an operation $\mathrm{HEAPIFY}(H, i)$ that maintains the heap property at index i
- Assumption: both children of the i-th node are roots of valid heaps
- The i-th node can have a smaller key than some of its children

Max-Heaps

- We shall describe an operation $\text{HEAPIFY}(H, i)$ that maintains the heap property at index i
- Assumption: both children of the i-th node are roots of valid heaps
- The i-th node can have a smaller key than some of its children
- The operation $\text{HEAPIFY}(H, i)$ transforms the subtree rooted at i into a valid heap

Max-Heaps

HEAPIFY(H, i):

Max-Heaps

$\text{HEAPIFY}(H, i)$:

$max \leftarrow i$;
if $\text{LEFT}(i) \leq H.heap\_size$ *and* $H[\text{LEFT}(i)].key \succ H[max].key$ **then**
  $max \leftarrow \text{LEFT}(i)$
end
if $\text{RIGHT}(i) \leq H.heap\_size$ *and* $H[\text{RIGHT}(i)].key \succ H[max].key$ **then**
  $max \leftarrow \text{RIGHT}(i)$
end
if $max \neq i$ **then**
  $H[i] \leftrightarrow H[max]$;
  $\text{HEAPIFY}(H, max)$;
end

Max-Heaps

$\text{HEAPIFY}(H, i)$:

```
max ← i;
if LEFT(i) ≤ H.heap_size and H[LEFT(i)].key ≻ H[max].key then
  | max ← LEFT(i)
end
if RIGHT(i) ≤ H.heap_size and H[RIGHT(i)].key ≻ H[max].key then
  | max ← RIGHT(i)
end
if max ≠ i then
  | H[i] ↔ H[max];
  | HEAPIFY(H, max);
end
```

- Time complexity: $\Theta(h)$, where h is the height of the i-th node

Max-Heaps

$\text{HEAPIFY}(H, i)$:

```
max ← i;
if LEFT(i) ≤ H.heap_size and H[LEFT(i)].key ≻ H[max].key then
 |  max ← LEFT(i)
end
if RIGHT(i) ≤ H.heap_size and H[RIGHT(i)].key ≻ H[max].key then
 |  max ← RIGHT(i)
end
if max ≠ i then
 |  H[i] ↔ H[max];
 |  HEAPIFY(H, max);
end
```

- Time complexity: $\Theta(h)$, where h is the height of the i-th node
- Worst-case over all nodes: $\Theta(\log n)$

Priority Queues via Max-Heaps

MAX(H):

Priority Queues via Max-Heaps

MAX(H):

return $H[1]$;

Priority Queues via Max-Heaps

$\text{MAX}(H)$:

return $H[1]$;

- Time complexity: $\Theta(1)$

Priority Queues via Max-Heaps

MAX(H):

return $H[1]$;

- Time complexity: $\Theta(1)$

EXTRACTMAX(H):

Priority Queues via Max-Heaps

MAX(H):

return $H[1]$;

- Time complexity: $\Theta(1)$

EXTRACTMAX(H):

```
if H.heap_size = 0 then
    error(underflow)
end
else
    max ← H[1];
    H[1] ← H[H.heap_size];
    H.heap_size ← H.heap_size − 1;
    HEAPIFY(H, 1);
    return max;
end
```

Priority Queues via Max-Heaps

$\text{MAX}(H)$:

```
return H[1];
```

- Time complexity: $\Theta(1)$

$\text{EXTRACTMAX}(H)$:

```
if H.heap_size = 0 then
|  error(underflow)
end
else
|  max ← H[1];
|  H[1] ← H[H.heap_size];
|  H.heap_size ← H.heap_size − 1;
|  HEAPIFY(H, 1);
|  return max;
end
```

- Time complexity: $\Theta(\log n)$

Priority Queues via Max-Heaps

$\text{INCREASEKEY}(H, i, k)$:

Priority Queues via Max-Heaps

$\text{INCREASEKEY}(H, i, k)$:

```
if k ≺ H[i].key then
 |  error(too small key)
end
else
 |  H[i].key ← k;
 |  while i > 1 and H[PARENT(i)].key ≺ H[i].key do
 |   |  H[i] ↔ H[PARENT(i)];
 |   |  i ← PARENT(i);
 |  end
end
```

Priority Queues via Max-Heaps

$\text{INCREASEKEY}(H, i, k)$:

```
if k ≺ H[i].key then
|   error(too small key)
end
else
|   H[i].key ← k;
|   while i > 1 and H[PARENT(i)].key ≺ H[i].key do
|   |   H[i] ↔ H[PARENT(i)];
|   |   i ← PARENT(i);
|   end
end
```

- Time complexity: $\Theta(\log n)$

Priority Queues via Max-Heaps

$\text{INSERT}(H, x)$:

Priority Queues via Max-Heaps

$\text{INSERT}(H, x)$:

$k \leftarrow x.key$;
$x.key \leftarrow \perp$;
$H.heap\_size \leftarrow H.heap\_size + 1$;
$H[H.heap\_size] \leftarrow x$;
$\text{INCREASEKEY}(H, H.heap\_size, k)$;

Priority Queues via Max-Heaps

$\text{INSERT}(H, x)$:

$k \leftarrow x.key$;
$x.key \leftarrow \bot$;
$H.heap\_size \leftarrow H.heap\_size + 1$;
$H[H.heap\_size] \leftarrow x$;
$\text{INCREASEKEY}(H, H.heap\_size, k)$;

- Time complexity: $\Theta(\log n)$

An Application of Heaps: Heap Sort

Idea:

An Application of Heaps: Heap Sort

Idea:

- First "convert" an array a into a heap: $\text{BUILDHEAP}(a)$

An Application of Heaps: Heap Sort

Idea:

- First “convert” an array a into a heap: $\mathrm{BUILDHEAP}(a)$
- The root of the heap is the maximum element of a

An Application of Heaps: Heap Sort

Idea:

- First "convert" an array a into a heap: $\mathrm{BUILDHEAP}(a)$
- The root of the heap is the maximum element of a
- Extract the maximum and place it at the single "free" position in a

An Application of Heaps: Heap Sort

Idea:

- First "convert" an array a into a heap: $\text{BUILDHEAP}(a)$
- The root of the heap is the maximum element of a
- Extract the maximum and place it at the single "free" position in a
- Repeat until all elements are processed

Heap Sort

BUILDHEAP(a):

Heap Sort

BUILDHEAP(a):

```
a.heap_size ← a.length;
for i ← ⌊a.length/2⌋ downto 1 do
  | HEAPIFY(a, i)
end
```

Heap Sort

BUILDHEAP(a):

$a.heap\_size \leftarrow a.length$;
for $i \leftarrow \lfloor a.length/2 \rfloor$ **downto** *1* **do**
| HEAPIFY(a, i)
end

Worst-Case Time Complexity $T(n)$:

Heap Sort

BUILDHEAP(a):

$a.heap\_size \leftarrow a.length$;
for $i \leftarrow \lfloor a.length/2 \rfloor$ **downto** *1* **do**
| HEAPIFY(a, i)
end

Worst-Case Time Complexity $T(n)$:

- Obviously $T(n) = \Omega(n)$

Heap Sort

BUILDHEAP(a):

$a.heap\_size \leftarrow a.length$;
for $i \leftarrow \lfloor a.length/2 \rfloor$ **downto** *1* **do**
| HEAPIFY(a, i)
end

Worst-Case Time Complexity $T(n)$:

- Obviously $T(n) = \Omega(n)$
- On the other hand, we have

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right),$$

Heap Sort

BUILDHEAP(a):

$a.heap\_size \leftarrow a.length$;
for $i \leftarrow \lfloor a.length/2 \rfloor$ **downto** *1* **do**
| HEAPIFY(a, i)
end

Worst-Case Time Complexity $T(n)$:

- Obviously $T(n) = \Omega(n)$
- On the other hand, we have

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right),$$

where

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = O(1)$$

Heap Sort

BUILDHEAP(a):

```
a.heap_size ← a.length;
for i ← ⌊a.length/2⌋ downto 1 do
 |  HEAPIFY(a, i)
end
```

Worst-Case Time Complexity $T(n)$:

- Obviously $T(n) = \Omega(n)$
- On the other hand, we have

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right),$$

  where

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = O(1)$$

- Hence, $T(n) = O(n)$, implying that $T(n) = \Theta(n)$

Heap Sort

HEAPSORT(a):

Heap Sort

HEAPSORT(a):

```
BUILDHEAP(a);
while a.heap_size ≥ 2 do
    x ← EXTRACTMAX(a);
    a[heap_size + 1] ← x;
end
```

Heap Sort

HEAPSORT(a):

```
BUILDHEAP(a);
while a.heap_size ≥ 2 do
    x ← EXTRACTMAX(a);
    a[heap_size + 1] ← x;
end
```

- Worst-case time complexity $T(n)$: surely $T(n) = O(n \log n)$

Heap Sort

HEAPSORT(a):

```
BUILDHEAP(a);
while a.heap_size ≥ 2 do
    x ← EXTRACTMAX(a);
    a[heap_size + 1] ← x;
end
```

- Worst-case time complexity $T(n)$: surely $T(n) = O(n \log n)$
- It can also be proved that $T(n) = \Omega(n \log n)$ (exercise)

Heap Sort

HEAPSORT(a):

```
BUILDHEAP(a);
while a.heap_size ≥ 2 do
    x ← EXTRACTMAX(a);
    a[heap_size + 1] ← x;
end
```

- Worst-case time complexity $T(n)$: surely $T(n) = O(n \log n)$
- It can also be proved that $T(n) = \Omega(n \log n)$ (exercise)
- As a result: $T(n) = \Theta(n \log n)$