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"
- 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 $x.key$
- 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:

- $\text{INSERT}(X, x)$: 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:

- 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:

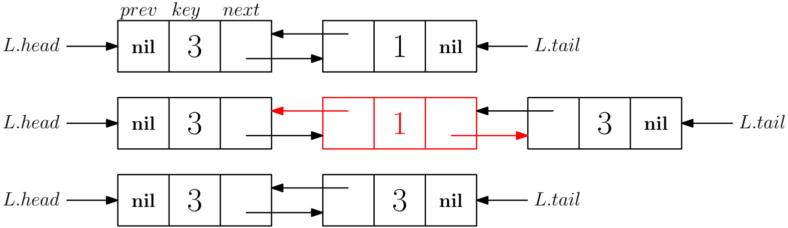
- Aim is to realise the supported operations efficiently
- There is no optimal data structure for all operations
- The choice of a data structure thus depends on the type of the dynamic set realised
- Examples: arrays, linked lists, heaps, trees, ...

Dictionaries

Dynamic sets supporting the following operations:

- $\text{SEARCH}(D, k)$: 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

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):

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

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

- Time complexity: $\Theta(1)$

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;

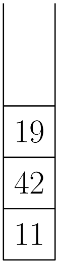
- Worst-case time complexity: $\Theta(n)$, where n is the number of elements in L

Stacks

Dynamic sets supporting the following operations:

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

LIFO = Last In First Out

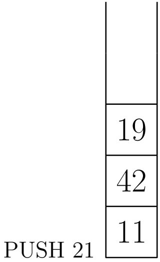


Stacks

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

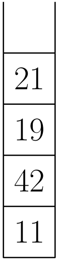


Stacks

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

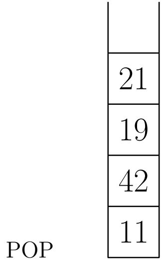


Stacks

Dynamic sets supporting the following operations:

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

LIFO = Last In First Out

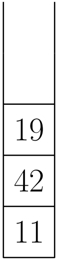


Stacks

Dynamic sets supporting the following operations:

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

LIFO = Last In First Out

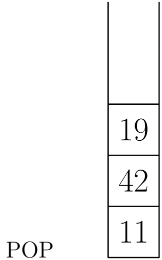


Stacks

Dynamic sets supporting the following operations:

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

LIFO = Last In First Out

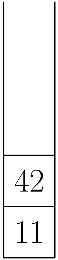


Stacks

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 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
  $S.num \leftarrow S.num - 1$;
  return $S.a[S.num + 1]$;
end

Queues

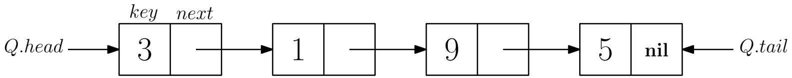
Dynamic sets supporting the following operations:

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

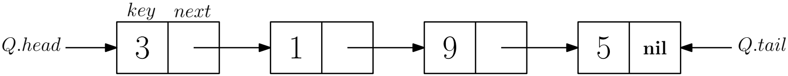


ENQUEUE(Q, x):

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):

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

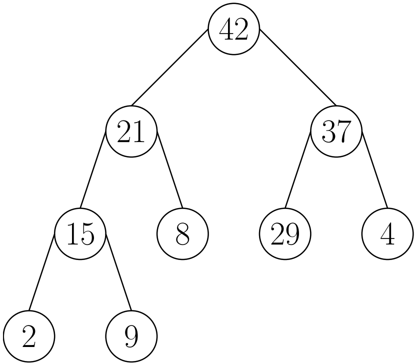
Dynamic sets supporting the following operations (assuming that each x has an attribute $x.key$ from a totally ordered set $(S, \preceq)$):

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

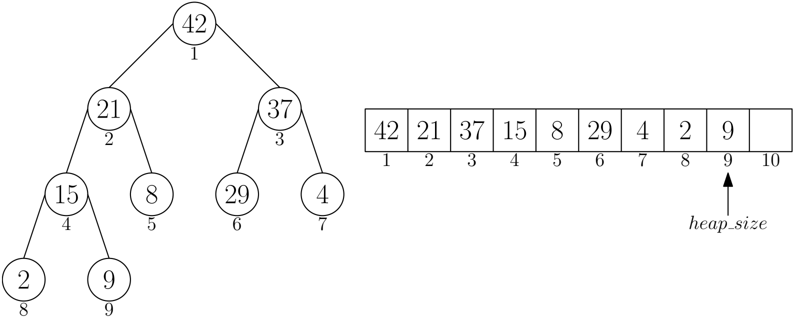
Max-Heaps

- "Nearly complete" binary trees
- 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



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
- 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

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

$max \leftarrow i$;
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

- 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):

return $H[1]$;

- Time complexity: $\Theta(1)$

EXTRACTMAX(H):

if $H.heap\_size = 0$ **then**
  error(underflow)
end
else
  $max \leftarrow H[1]$;
  $H[1] \leftarrow H[H.heap\_size]$;
  $H.heap\_size \leftarrow H.heap\_size - 1$;
  HEAPIFY($H, 1$);
  return max;
end

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

Priority Queues via Max-Heaps

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

```
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

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

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

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

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):

$a.heap\_size \leftarrow a.length$;
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

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

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

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)
- As a result: $T(n) = \Theta(n \log n)$