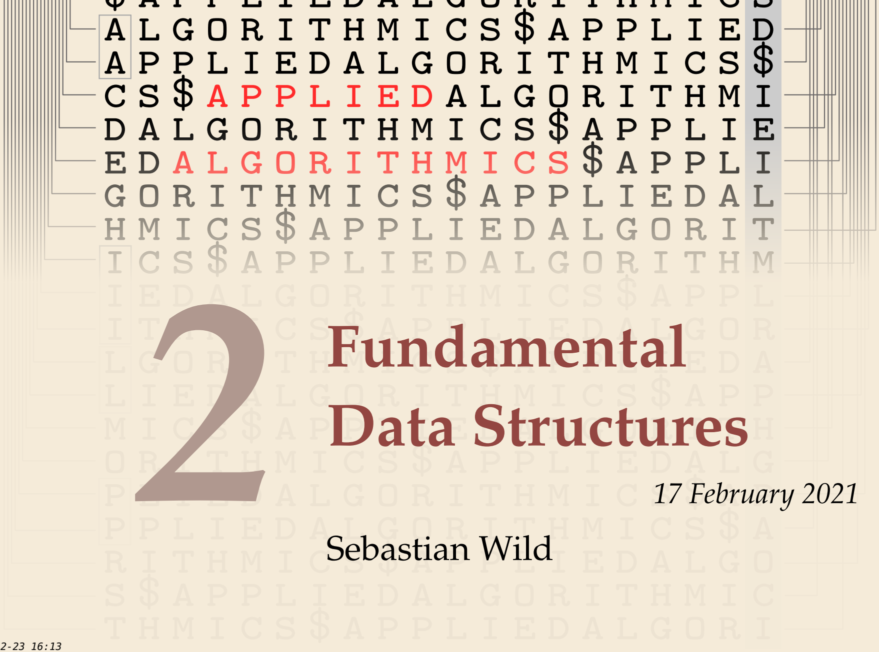


2

Fundamental Data Structures

17 February 2021

Sebastian Wild

version `2021-02-23 16:13`

Outline

2 Fundamental Data Structures

2.1 Stacks & Queues

Abstract Data Types

abstract data type (ADT)

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

≈ Java `interface`
(with Javadoc comments)

vs.

data structures

- specify exactly
  how data is represented
- algorithms for operations
- has concrete costs
  (space and running time)

≈ Java `class` (implementing interfaces)
(non abstract)

Abstract Data Types

abstract data type (ADT)

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

≈ Java `interface`
(with Javadoc comments)

VS.

data structures

- specify exactly
  how data is represented
- algorithms for operations
- has concrete costs
  (space and running time)

≈ Java `class`
(non abstract)

Why separate?

- Can swap out implementations ⇝ "drop-in replacements")

⇝ reusable code!

- (Often) better abstractions
- Prove generic lower bounds (⇝ Unit 3)

Abstract Data Types

abstract data type (ADT)

- list of supported operatio
- **what** should happen
- **not:** how to do it
- **not:** how to store data

≈ Java `interface`
(with Javadoc comments)

Why separate?

- Can swap out implement

⇝ reusable code!

- (Often) better abstractions
- Prove generic lower bounds (⇝ Unit 3)

Clicker Question

Which of the following are examples of abstract data types?

A ADT
B Stack
C Deque
D Linked list
E binary search tree
F Queue
G resizable array
H heap
I priority queue
J dictionary/symbol table
K hash table

`sli.do/comp526`

Click on “Polls” tab

Clicker Question

Which of the following are examples of abstract data types?

- **A** ~~ADT~~
- **B** Stack ✓
- **C** Deque ✓
- **D** ~~Linked list~~
- **E** ~~binary search tree~~
- **F** Queue ✓
- **G** ~~resizable array~~
- **H** ~~heap~~
- **I** priority queue ✓
- **J** dictionary/symbol table ✓
- **K** ~~hash table~~

`sli.do/comp526`

Click on “Polls” tab

Stacks

Stack ADT

- `top()`
  Return the topmost item on the stack
  Does not modify the stack.
- `push(`x`)`
  Add x onto the top of the stack.
- `pop()`
  Remove the topmost item from the stack
  (and return it).
- `isEmpty()`
  Returns `true` iff stack is empty.
- `create()`
  Create and return an new empty stack.

Clicker Question

Suppose a stack initially contains the numbers $1, 2, 3, 4, 5$ with 1 at the top.
What is the content of the stack after the following operations:

```
pop(); pop(); push(1);
```

- **A** 1,2,3,1
- **B** 3,4,5,1
- **C** 1,3,4,5
- **D** empty
- **E** 1,2,3,4,5

~~1~~
~~2~~ 1
3
4
5

`sli.do/comp526`

Click on “Polls” tab

Clicker Question

Suppose a stack initially contains the numbers 1, 2, 3, 4, 5 with 1 at the top.
What is the content of the stack after the following operations:

`pop(); pop(); push(1);`

- **A** ~~1,2,3,1~~
- **B** ~~3,4,5,1~~
- **C** 1,3,4,5 ✓
- **D** ~~empty~~
- **E** ~~1,2,3,4,5~~

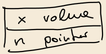
`sli.do/comp526`

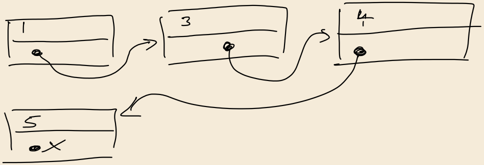
Click on “Polls” tab

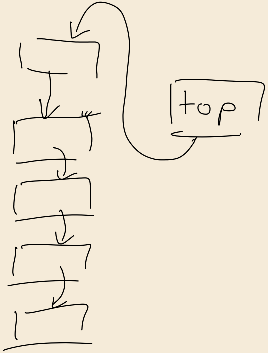
Linked-list implementation for Stack

Invariants:

- maintain `top` pointer to topmost element
- each element points to the element below it (or `null` if bottommost)







Linked-list implementation for Stack

Invariants:

- maintain `top` pointer to topmost element
- each element points to the element below it (or `null` if bottommost)

Linked stacks:

$n \cdot$

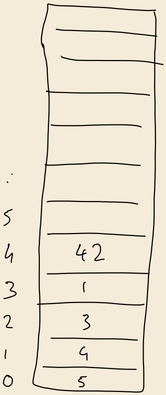
- require $\underline{\Theta(n)}$ space when n elements on stack
- All operations take $\underline{O(1)}$ time

Array-based implementation for Stack

Can we avoid extra space for pointers?
⇝ array-based implementation

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`.



top: ~~3~~ 4

Array-based implementation for Stack

Can we avoid extra space for pointers?
⇝ array-based implementation

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`.

What to do if stack is full upon `pop`?

Array stacks:

- require *fixed capacity C* (known at creation time)!
- require $\Theta(C)$ space for a capacity of C elements
- all operations take $O(1)$ time

2.2 Resizable Arrays

Digression – Arrays as ADT

Arrays can also be seen as an ADT!

Array operations:

- `create(`n`)` *Java*: `A = new int[`n`];`
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$
- `get(`i`)` *Java*: `A[`i`]`
  Return the content of cell i
- `set(`i`,`x`)` *Java*: `A[`i`] = `x`;`
  Set the content of cell i to x.

⇝ Arrays have fixed size (supplied at creation).

Digression – Arrays as ADT

Arrays can also be seen as an ADT! . . . but are commonly seen as specific data structure

Array operations:

- `create(`n`)` *Java*: `A = new int[`n`];`
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$
- `get(`i`)` *Java*: `A[`i`]`
  Return the content of cell i
- `set(`i`,`x`)` *Java*: `A[`i`] = `x`;`
  Set the content of cell i to x.

⇝ Arrays have fixed size (supplied at creation).

Usually directly implemented by compiler + operating system / virtual machine.

⚠ **Difference to others ADTs:** *Implementation usually fixed*
to "a contiguous chunk of memory".

Doubling trick

Can we have unbounded stacks based on arrays? Yes!

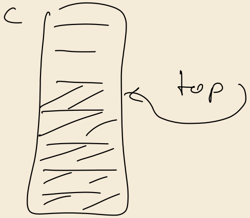
Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`
- maintain capacity $C = \texttt{S.length}$ so that $\boxed{\frac{1}{4}C \le n \le C}$

$\leadsto$ can always push more elements!



Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array `S` of elements, from bottommost to topmost
- maintain index `top` of position of topmost element in `S`
- maintain capacity $C =$ `S.length` so that $\frac{1}{4}C \le n \le C$

⇝ can always push more elements!

How to maintain the last invariant?

- before `push`
  If $n = C$, allocate new array of size $2n$, copy all elements.
- after `pop`
  If $n < \frac{1}{4}C$, allocate new array of size $2n$, copy all elements.

⇝ ***"Resizing Arrays"***

← an implementation technique, not an ADT!

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

A The elements are stored in an array of size $2n$.

B Adding or deleting an element at the end takes constant time.

C A sequence of m insertions or deletions at the end of the array takes time $O(n+m)$.

D Inserting and deleting any element takes $O(1)$ amortized time.

`sli.do/comp526`

Click on “Polls” tab

Amortized Analysis

- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!

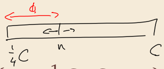
Amortized Analysis

blue parts are corrections after lecture
(looks different in video recordings)

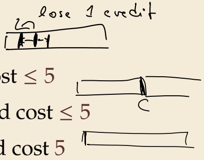
- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!

#array accesses

distance to boundary



Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$



- amortized cost of an operation = actual cost (array accesses) $-\,4 \cdot$ change in Φ
  - cheap `push`/`pop`: actual cost 1 array access, consumes ≤ 1 credits ⇝ amortized cost ≤ 5
  - copying `push`: actual cost $2n+1$ array accesses, creates $\frac{1}{2}n+1$ credits ⇝ amortized cost ≤ 5
  - copying `pop`: actual cost $2n+1$ array accesses, creates $\frac{1}{2}n-1$ credits ⇝ amortized cost 5

⇝ **sequence** of m operations: total actual cost $\le$ total amortized cost $+$ final credits

here: $\le \quad 5m \quad + \quad 4 \cdot 0.6n \quad = \Theta(m+n)$

$$a_i = c_i - 4(\phi_i - \phi_{i-1}) \le 5$$

$$\sum_{i=1}^{m} a_i \le 5m \ge \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i - 4\sum_{i=1}^{m}(\phi_i - \phi_{i-1}) = \sum_{i=1}^{m} c_i - 4(\phi_m - \phi_0)$$

$$\sum_{i=1}^{m} c_i \le 5m + 4\phi_m - 4\phi_0 \le 5m + 4\phi_m$$

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

- **A** The elements are stored in an array of size $2n$.
- **B** Adding or deleting an element at the end takes constant time.
- **C** A sequence of m insertions or deletions at the end of the array takes time $O(n + m)$.
- **D** Inserting and deleting any element takes $O(1)$ amortized time.

`sli.do/comp526`

Click on "Polls" tab

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

- **A** ~~The elements are stored in an array of size $2n$.~~
- **B** ~~Adding or deleting an element at the end takes constant time.~~
- **C** A sequence of m insertions or deletions at the end of the array takes time $O(n + m)$. ✓
- **D** ~~Inserting and deleting any element takes $O(1)$ amortized time.~~

`sli.do/comp526`

Click on “Polls” tab

Queues

Operations:

- `enqueue(`x`)`
  Add x at the end of the queue.
- `dequeue()`
  Remove item at the front of the queue and return it.

→

Implementations similar to stacks.

Bags

What do Stack and Queue have in common?

Bags

What do Stack and Queue have in common?

They are special cases of a ***Bag***!

Operations:

- `insert(`x`)`
  Add x to the items in the bag.
- `delAny()`
  Remove any one item from the bag and return it.
  (Not specified which; any choice is fine.)
- roughly similar to Java's `Collection`

Sometimes it is useful to state that order is irrelevant ⇝ Bag
Implementation of Bag usually just a Stack or a Queue

2.3 Priority Queues

Clicker Question

What is a heap-ordered tree?

- **A** A tree in which every node has exactly 2 children.
- **B** A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.
- **C** A tree where all keys in the left subtree and right subtree are bigger than the key at the root.
- **D** An tree that is stored in the heap-area of the memory.

`sli.do/comp526`

Click on “Polls” tab

Priority Queue ADT

Now: elements in the bag have different *priorities*.

(Max-oriented) Priority Queue (MaxPQ):

- `construct(`A`)`
  Construct from from elements in array A.
- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- `max()`
  Return item with largest priority. (Does not modify the PQ.)
- `delMax()`
  Remove the item with largest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to *increasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

Priority Queue ADT – min-oriented version

Now: elements in the bag have different *priorities*.

(~~Max-~~Min-oriented) Priority Queue (~~Max~~MinPQ):

- `construct(`A`)`
  Construct from from elements in array A.
- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- ~~`max`~~`min()`
  Return item with ~~largest~~ smallest priority. (Does not modify the PQ.)
- `del`~~`Max`~~`Min()`
  Remove the item with ~~largest~~ smallest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to *~~in~~decreasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

Clicker Question

Suppose we start with an empty priority queue and insert the numbers $7, 2, 4, 1, 9$ in that order. What is the result of `delMin()`?

- **A** $-\infty$
- **B** 1
- **C** 2
- **D** 4
- **E** 7
- **F** 9
- **G** not allowed

sli.do/comp526

Click on “Polls” tab

Clicker Question

Suppose we start with an empty priority queue and insert the numbers $7, 2, 4, 1, 9$ in that order. What is the result of `delMin()`?

- **A** ~~$-\infty$~~
- **B** 1 ✓
- **C** ~~2~~
- **D** ~~4~~
- **E** ~~7~~
- **F** ~~9~~
- **G** ~~not allowed~~

`sli.do/comp526`

Click on “Polls” tab

PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

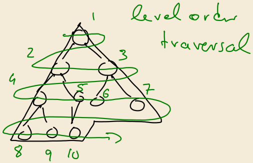
Can we get something between these extremes? Like a "slightly sorted" list?

PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

Can we get something between these extremes? *Like a "slightly sorted" list?*



Yes! *Binary heaps.*

Array view

Heap = array A with
$\forall i \in [n] : A[\lfloor i/2 \rfloor] \geq A[i]$

$\equiv$

store nodes
in level order
in $A[1..n]$

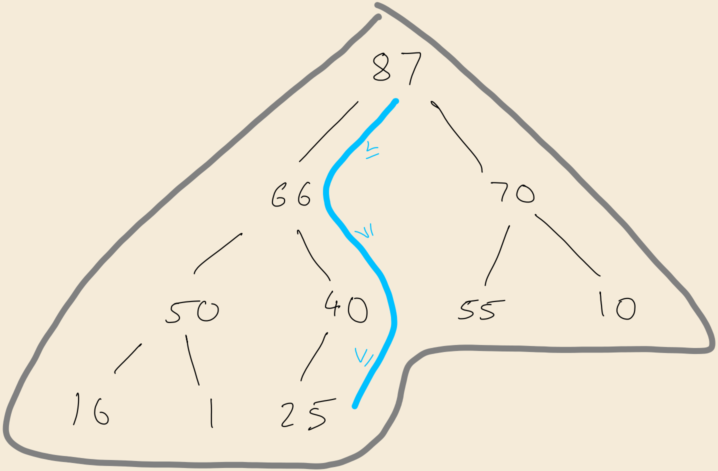
Tree view

Heap = tree that is
(i) a complete binary tree
(ii) heap ordered

all but last level full
last level flush left

parent
~~father~~ $\geq$ children

Binary heap example



Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape ⇝ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:

  For a node at index k in A
  - parent at $\lfloor k/2 \rfloor$
  - left child at $2k$
  - right child at $2k+1$

Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape ⇝ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:

  For a node at index k in A
  - parent at $\lfloor k/2 \rfloor$
  - left child at $2k$
  - right child at $2k+1$

Why heap ordered?

- Maximum must be at root! ⇝ `max()` is trivial!
- But: Sorted only along paths of the tree; leaves lots of leeway for fast inserts

how? . . . stay tuned

Clicker Question

What is a heap-ordered tree?

- **A** ~~A tree in which every node has exactly 2 children.~~
- **B** ~~A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.~~
- **C** A tree where all keys in the left subtree and right subtree are bigger than the key at the root. ✓
- **D** ~~An tree that is stored in the heap area of the memory.~~

`sli.do/comp526`

Click on “Polls” tab

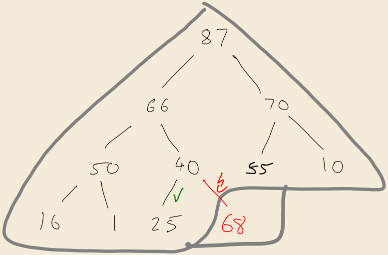
Insert

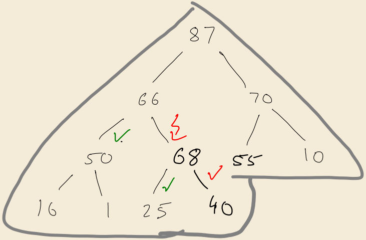
68

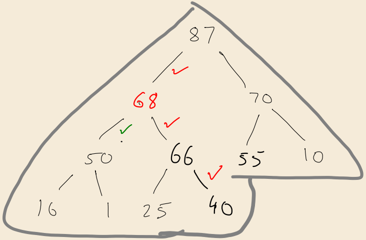
shape: only 1 possible position

BUT heap-order ↯

⇒ swim up the heap

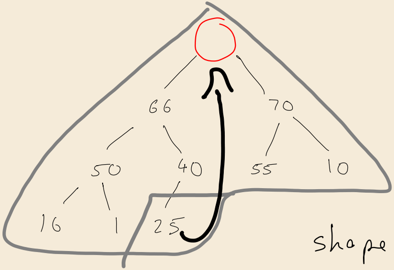






✓

Delete Max



87 find max is easy

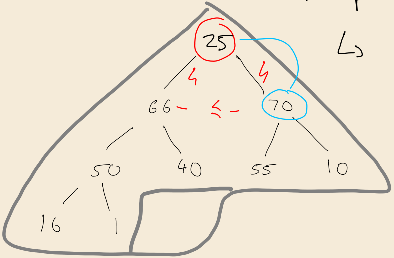
after removing it

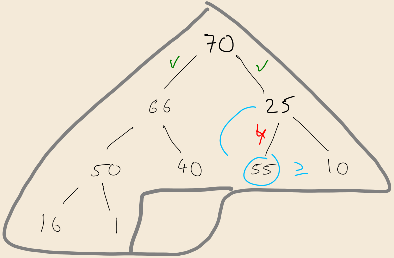
↯ complete binary tree

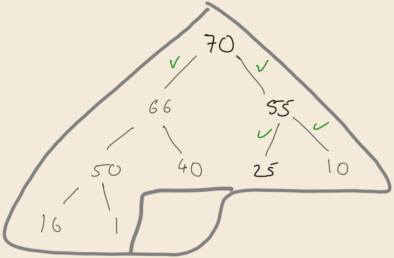
shape ✓

heap-order ↯

↳ let 25 sink in heap



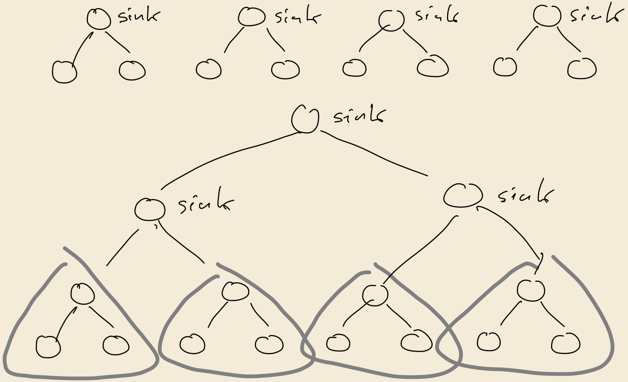

70
✓
✓
66
25
50
40
55
≥
10
16
1


70
✓
✓
66
55
✓
✓
50
40
25
10
16
1

Heap construction

n · insert $\Rightarrow$ $\Theta(n \log n)$

can do better:



$\frac{n}{4}$ · sink in heap of size ≤ 3

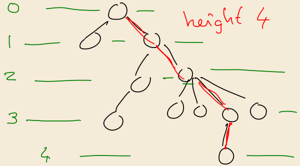
$\frac{n}{8}$ · 〃 ≤ 7

$\frac{n}{2^k}$ $\leq 2^k - 1$

$k = \lg n$

$\Theta(n)$ total time for heap constr.

Analysis



Height of binary heaps:

- *height* of a tree: #edges on longest root-to-leaf path
- *depth/level* of a node: #edges from root ⇝ root has depth 0
- How many nodes on first k *full* levels? $\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

$\log_2$

⇝ Height of binary heap: $h \;=\; \min k \text{ s.t. } 2^{k+1} - 1 \geq n \;=\; \lfloor \lg(n) \rfloor$

Analysis

Height of binary heaps:

- *height* of a tree: # edges on longest root-to-leaf path
- *depth/level* of a node: # edges from root $\rightsquigarrow$ root has depth 0
- How many nodes on first k *full* levels? $\displaystyle\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

$\rightsquigarrow$ Height of binary heap: $h \;=\; \min k \text{ s.t. } 2^{k+1} - 1 \ge n \;=\; \lfloor \lg(n) \rfloor$

Analysis:

- `insert`: new element "swims" up $\rightsquigarrow$ $\le h$ steps (h cmps)
- `delMax`: last element "sinks" down $\rightsquigarrow$ $\le h$ steps ($2h$ cmps)
- `construct` from n elements:

  cost = cost of letting *each node* in heap sink!

$$\begin{aligned}
\text{cost} &= \text{cost of letting each node in heap sink!}\\
&\le 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^{\ell} \cdot (h-\ell) + \cdots + 2^{h-1} \cdot 1 + 2^{h} \cdot 0\\
&= \sum_{\ell=0}^{h} 2^{\ell}(h-\ell) \;=\; \sum_{i=0}^{h} \frac{2^{h}}{2^{i}} i \;=\; 2^{h} \sum_{i=0}^{h} \frac{i}{2^{i}} \;\le\; 2 \cdot 2^{h} \;\le\; 4n
\end{aligned}$$

Binary heap summary

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(n)$
`max()`	$O(1)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

2.4 Binary Search Trees

Clicker Question

Have you ever used a printed dictionary (physical book)?

- **A** Yes
- **B** No

`sli.do/comp526`

Click on "Polls" tab

Clicker Question

What is a binary search tree (tree in symmetric order)?

A A tree in which every node has exactly 2 children.

B A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.

C A tree where all keys in the left subtree and right subtree are bigger than the key at the root.

D A tree that is stored in the heap-area of the memory.

`sli.do/comp526`

Click on "Polls" tab

Clicker Question

What is a binary search tree (tree in symmetric order)?

A ~~A tree in which every node has exactly 2 children.~~

B A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root. ✓

C ~~A tree where all keys in the left subtree and right subtree are bigger than the key at the root.~~

D ~~A tree that is stored in the heap area of the memory.~~

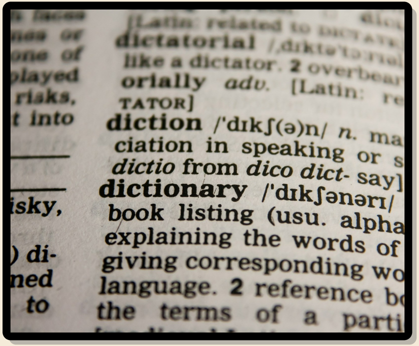
`sli.do/comp526`

Click on “Polls” tab

Symbol table ADT

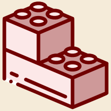
Java: `java.util.Map<K,V>`

Symbol table / Dictionary / Map / Associative array / key-value store:



[Latin: related to DICTA
dictatorial /dɪktəˈtɔːrɪal
like a dictator. 2 overbear
orially adv. [Latin: re
TATOR]
diction /ˈdɪkʃ(ə)n/ n. ma
ciation in speaking or s
dictio from dico dict- say]
dictionary /ˈdɪkʃənərɪ/
book listing (usu. alpha
explaining the words of
giving corresponding wo
language. 2 reference b
the terms of a parti

- `put(`k`,`v`)` *Python* dict: `d[`k`] = `v
  Put key-value pair (k, v) into table
- `get(`k`)` *Python* dict: `d[`k`]`
  Return value associated with key k
- `delete(`k`)`
  Remove key k (any associated value) form table
- `contains(`k`)`
  Returns whether the table has a value for key k
- `isEmpty(), size()`
- `create()`



Most fundamental building block in computer science.

(Every programming library has a symbol table implementation.)

Symbol tables vs mathematical functions

- similar interface
- but: mathematical functions are *static* (never change their mapping)
  (Different mapping is a *different* function)
- symbol table $=$ *dynamic* mapping
  Function may change over time

Elementary implementations

Unordered (linked) list:

👍 Fast `put`

👎 $\Theta(n)$ time for `get`

$\leadsto$ Too slow to be useful

Elementary implementations

Unordered (linked) list:

👍 Fast `put`

👎 $\Theta(n)$ time for `get`

⇝ Too slow to be useful

Sorted *linked* list:

👎 $\Theta(n)$ time for `put`

👎 $\Theta(n)$ time for `get`

⇝ Too slow to be useful

⇝ *Sorted order does not help us at all?!*

Binary search

It does help . . . if we have a sorted ***array***!

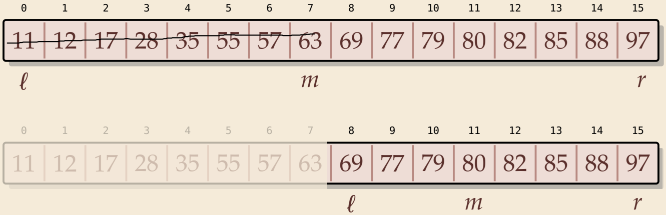
Example: search for 69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

Binary search

It does help . . . if we have a sorted ***array****!*

Example: search for 69



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
~~11~~	~~12~~	~~17~~	~~28~~	~~35~~	~~55~~	~~57~~	~~63~~	69	77	79	80	82	85	88	97
ℓ							m								r

$$63 \overset{<}{?} 69$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

Binary search

It does help . . . if we have a sorted ***array****!*

Example: search for 69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

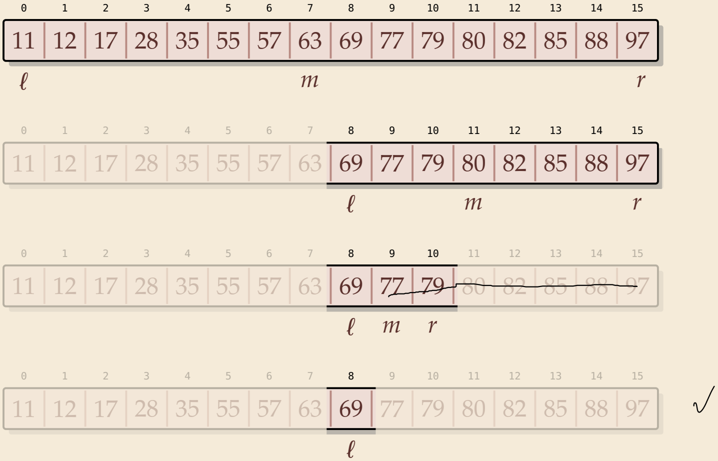
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ	m	r					

Binary search

*It does help . . . if we have a sorted **array**!*

Example: search for 69



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ	m	r					

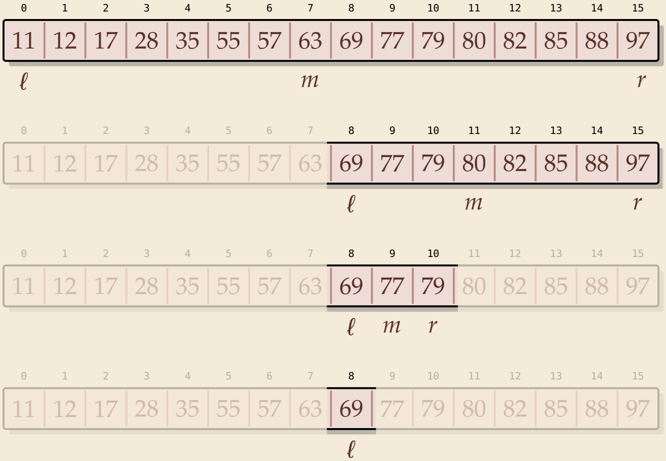
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ							

✓

Binary search

It does help . . . if we have a sorted ***array****!*

Example: search for 69



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ	m	r					

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ							

Binary search:

±1

- halve remaining list in each step

⇝ $\le \lfloor \lg n \rfloor + 1$ cmps in the worst case

⚠ needs random access

Clicker Question

Suppose we have a sorted array containing the numbers $10, 20, 30, 40, 50, 60, 70$ and we use binary search to check whether this array contains key 25.
What is the sequence of comparisons executed by the binary search algorithm?

- **A** $10 < 25, 20 < 25, 30 > 25$
- **B** $40 > 25, 20 < 25, 30 > 25$
- **C** $20 < 25 < 30$
- **D** $40 > 25, 20 < 25$
- **E** don’t know

`sli.do/comp526`

Click on “Polls” tab

Clicker Question

Suppose we have a sorted array containing the numbers $10, 20, 30, 40, 50, 60, 70$ and we use binary search to check whether this array contains key 25.
What is the sequence of comparisons executed by the binary search algorithm?

- **A** ~~$10 < 25, 20 < 25, 30 > 25$~~
- **B** $40 > 25, 20 < 25, 30 > 25$ ✓
- **C** ~~$20 < 25 < 30$~~
- **D** ~~$40 > 25, 20 < 25$~~
- **E** ~~don’t know~~

`sli.do/comp526`

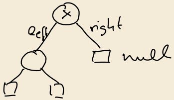
Click on “Polls” tab

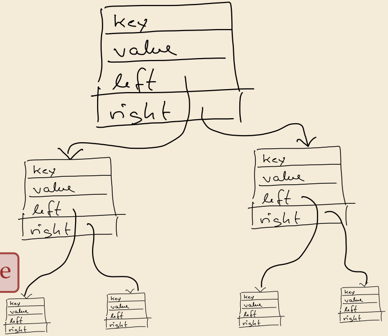
Binary search trees

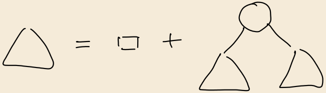
Binary search trees (BSTs) ≈ dynamic sorted array

- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

all keys in left subtree ≤ root key ≤ all keys in right subtree

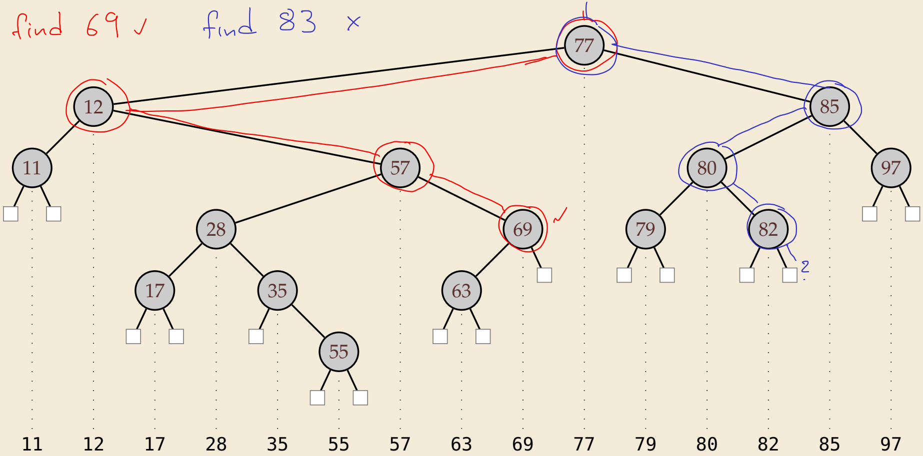






BST example & find

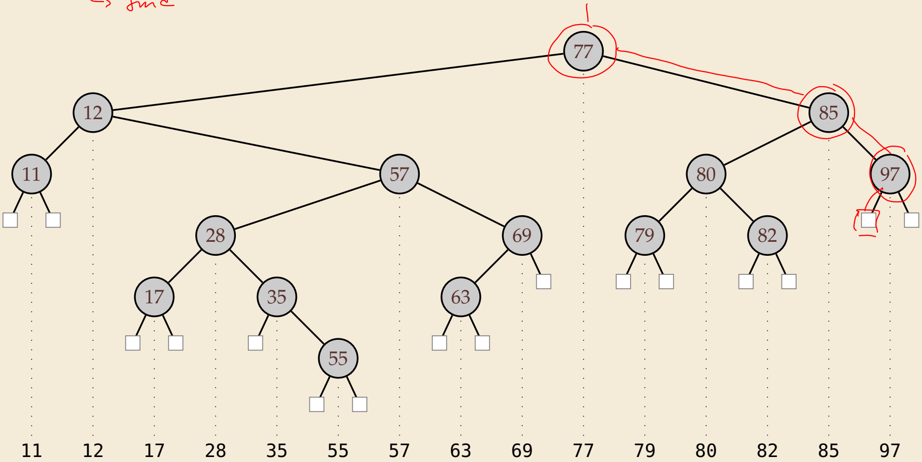
find 69 ✓ find 83 ×



BST insert

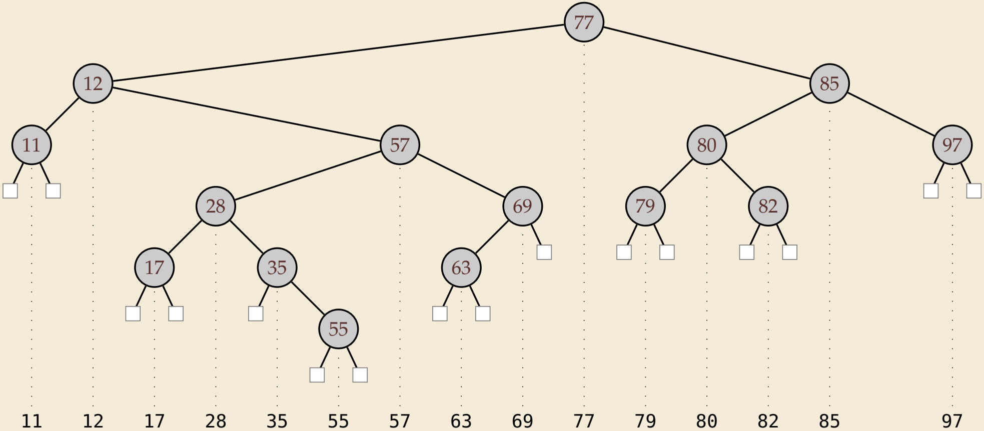
Example: Insert 88

↳ find



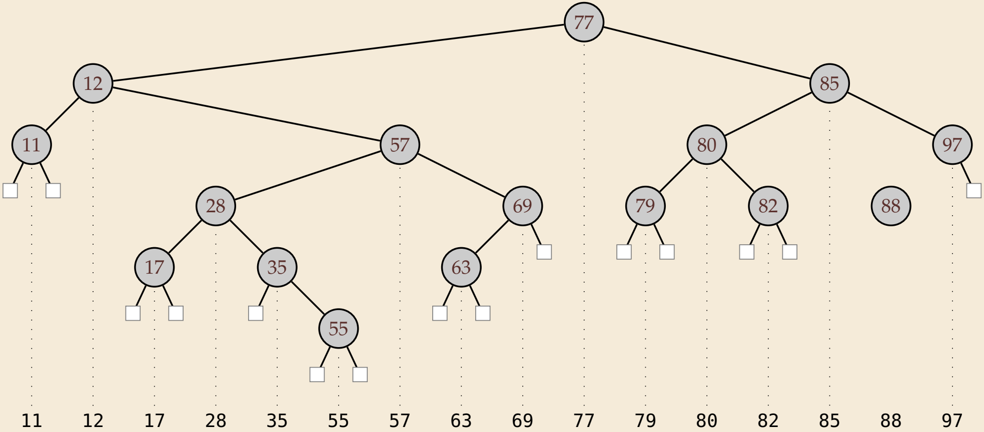
BST insert

Example: Insert 88



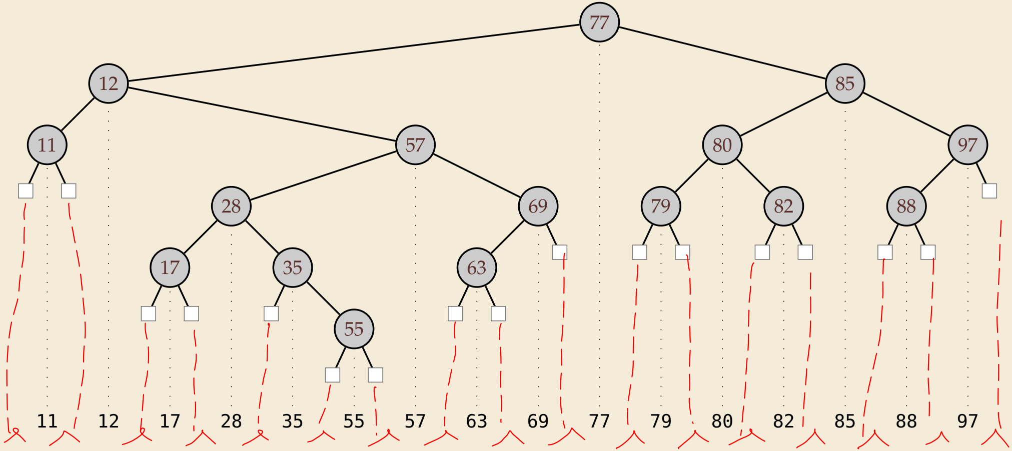
BST insert

Example: Insert 88



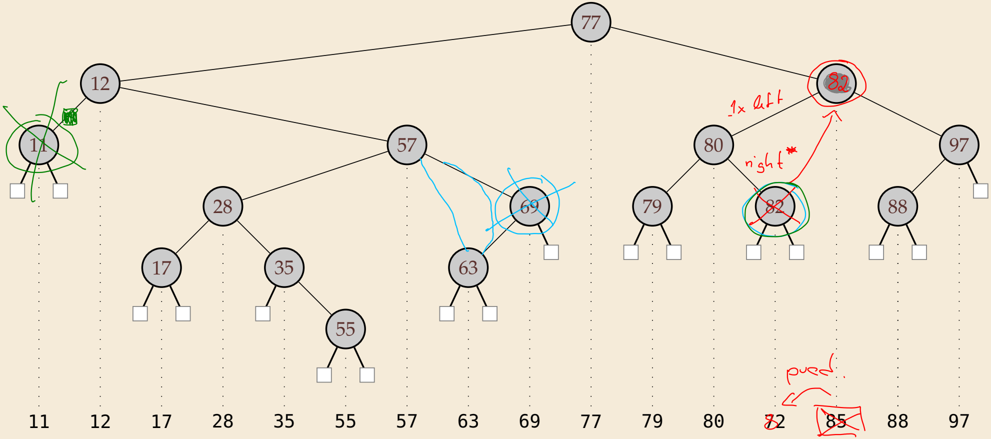
BST insert

Example: Insert 88



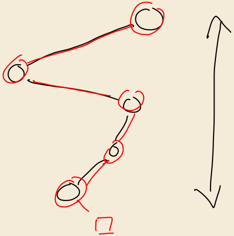
BST delete

- Easy case: remove leaf, e. g., 11 ⇝ replace by `null`
- Medium case: remove unary, e. g., 69 ⇝ replace by unique child
- Hard case: remove binary, e. g., 85 ⇝ swap with predecessor, recurse



Analysis

search :



$\leq$ height of BST

(h+1 cmps)
(<, >, =)

insert : same as search $O(h)$

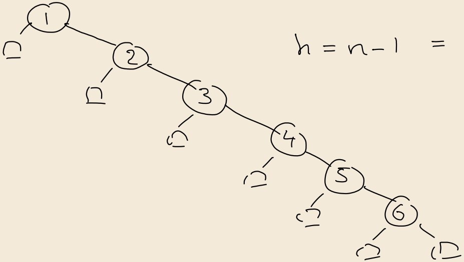
delete : search for k plus its predecessor $O(h)$

BST summary

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(nh)$
`put(`k`,`v`)`	$O(h)$
`get(`k`)`	$O(h)$
`delete(`k`)`	$O(h)$
`contains(`k`)`	$O(h)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

What is the height of a BST ?

a) worst case



$h = n-1 = \Theta(n)$

b) average case

(insertions in random order & no delete)

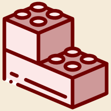
$h = \Theta(\log n)$ (even with high probability)

2.5 Ordered Symbol Tables

Ordered symbol tables

ADT

- `min()`, `max()`
  Return the smallest resp. largest key in the ST
- `floor(`x`)`, $\lfloor x \rfloor = \mathbb{Z}$`.floor(`$x$`)`
  Return largest key k in ST with $k \le x$.
- `ceiling(`x`)` $\lceil x \rceil$
  Return smallest key k in ST with $k \ge x$.
- `rank(`x`)`
  Return the number of keys k in ST $k < x$.
- `select(`$\overset{i}{i}$`)` A[i]
  Return the $\overset{i}{\cancel{i}}$th smallest key in ST (zero-based, i. e., $\overset{i}{\cancel{i}} \in [0..n)$)



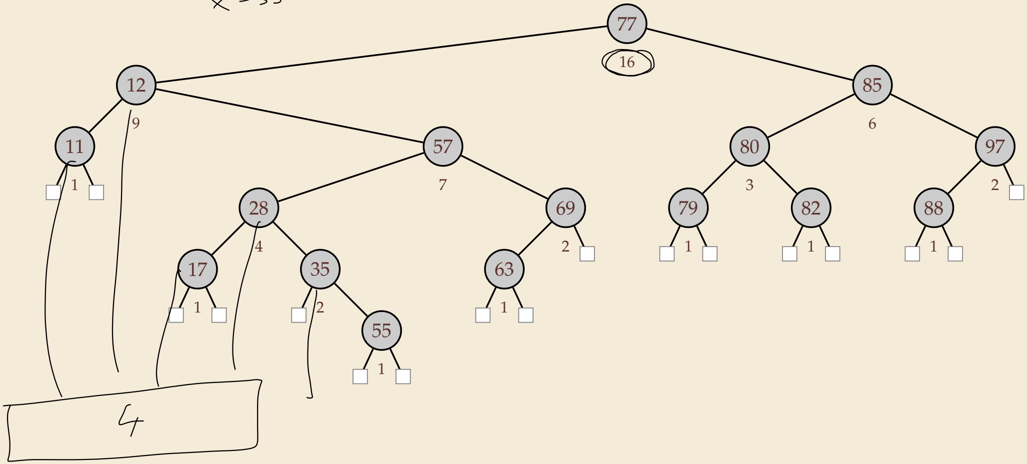
With select, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

Augmented BSTs

slido: rank(35)

x = 35

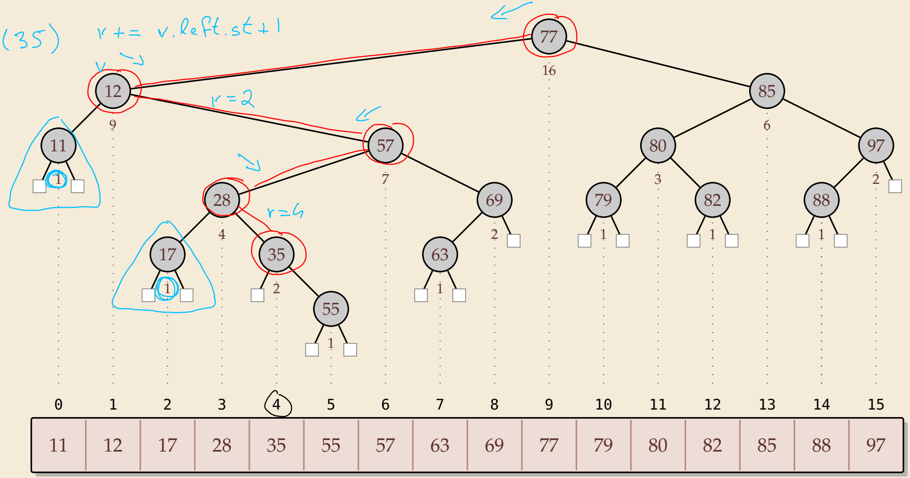


Rank

rank(35)

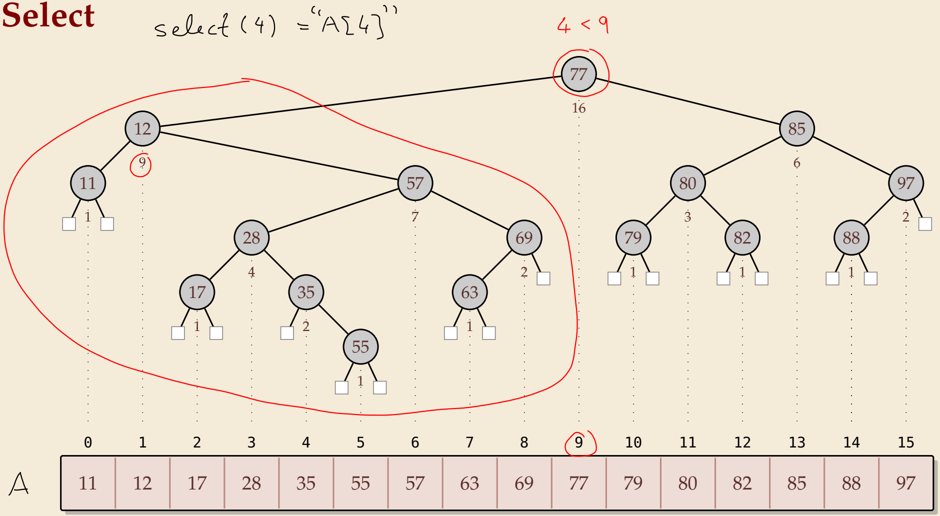
r = 0

r += v.left.st + 1

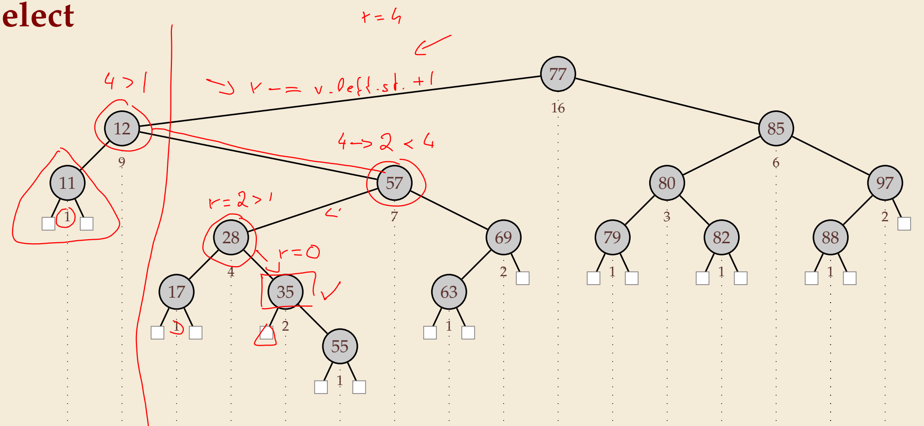


Select

select(4) = "A[4]"



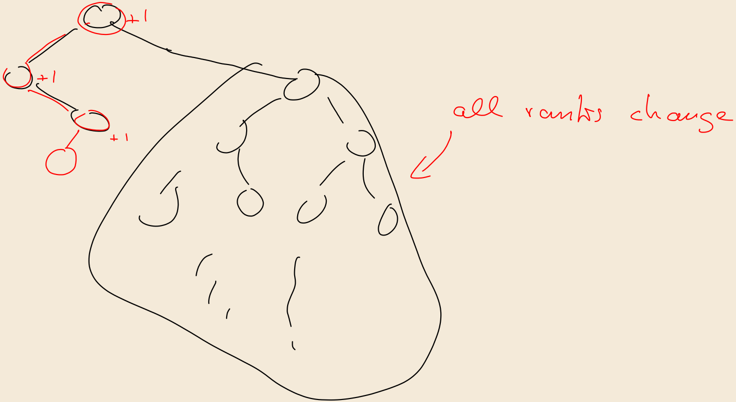
Select



0	1	2	3 -2	4	5	6 -2	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97

each node stores its subtree size (not ranks)

can be maintained upon updates



2.6 Balanced BSTs

Clicker Question

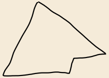
What ways of maintaining a **balanced** binary search tree do you know?
Write “none” if you have not seen balanced BSTs before.

`sli.do/comp526`

Click on “Polls” tab

Balanced BSTs

heaps

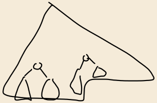


too strict for BST

Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates

Balanced BSTs

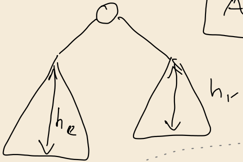


Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates
- many examples known
  - AVL trees (height-balanced trees)
  - red-black trees
  - weight-balanced trees ($\mathrm{BB}[\alpha]$ trees)
  - …

shape invariant:

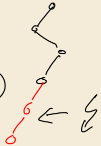
AVL: $|h_\ell - h_r| \le 1$



invariant: red edges
black edges

Sedgewick & Wayne

①



no two red edges in a row

② all leaves have same black height

Balanced BSTs

Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates

- many examples known
  - AVL trees (height-balanced trees)
  - red-black trees
  - weight-balanced trees ($\mathrm{BB}[\alpha]$ trees)
  - ...

- other (simpler) options:
  - amortization: splay trees, scapegoat trees
  - randomization: randomized BSTs, treaps, skip lists

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps
- MaxPQ abstraction still helpful

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(A[1..n])`	$O(n \log n)$
`put(k,v)`	$O(\log n)$
`get(k)`	$O(\log n)$
`delete(k)`	$O(\log n)$
`contains(k)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n)$ $\rightsquigarrow$ $O(1)$
`floor(x)`	$O(\log n)$
`ceiling(x)`	$O(\log n)$
`rank(x)`	$O(\log n)$
`select(i)`	$O(\log n)$

~~Binary heaps~~ *Strict Fibonacci heaps*

Operation	Running Time
`construct(A[1..n])`	$O(n)$
`insert(x,p)`	~~$O(\log n)$~~ $O(1)$
`delMax()`	$O(\log n)$
`changeKey(x,p')`	~~$O(\log n)$~~ $O(1)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`,
  BSTs always as good as binary heaps
- MaxPQ abstraction still helpful
- and faster heaps exist!