

3 Fundamental Data Structures

28 October 2024

Prof. Dr. Sebastian Wild

Learning Outcomes

Unit 3: Fundamental Data Structures

1. Understand and demonstrate the difference between *abstract data type (ADT)* and its *implementation*
2. Be able to define the ADTs *stack*, *queue*, *priority queue* and *dictionary / symbol table*
3. Understand *array*-based implementations of stack and queue
4. Understand *linked lists* and the corresponding implementations of stack and queue
5. Know *binary heaps* and their performance characteristics
6. Understand *binary search trees* and their performance characteristics
7. Know high-level idea of basic *hashing strategies* and their performance characteristics

Outline

3 Fundamental Data Structures

- 3.1 Stacks & Queues
- 3.2 Resizable Arrays
- 3.3 Priority Queues & Binary Heaps
- 3.4 Operations on Binary Heaps
- 3.5 Symbol Tables
- 3.6 Binary Search Trees
- 3.7 Ordered Symbol Tables
- 3.8 Balanced BSTs
- 3.9 Hashing

Clicker Question

What's the running time (on our word-RAM model with word size w) of this Java instruction?

`Object[] A = new Object[n];`



- | | | |
|----------------------|-------------------|------------------------|
| (A) 1 | (D) $\Theta(w)$ | (G) $\Theta(n \log n)$ |
| (B) $\Theta(1)$ | (E) $\Theta(n/w)$ | (H) $\Theta(nw)$ |
| (C) $\Theta(\log n)$ | (F) $\Theta(n)$ | (I) $\Theta(n^2)$ |



→ *sli.do/cs566*

Clicker Question

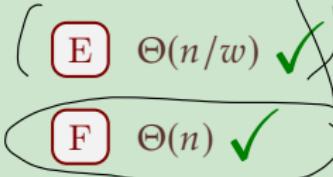
What's the running time (on our word-RAM model with word size w) of this Java instruction?

`Object[] A = new Object[n];`



- (A) $\Theta(1)$
- (B) $\Theta(n)$
- (C) $\Theta(\log n)$

(D) $\Theta(w)$



(F) $\Theta(n)$

(G) $\Theta(n \log n)$

(H) $\Theta(nw)$

(I) $\Theta(n^2)$



→ sli.do/cs566

Recap: The Random Access Machine

- ▶ Data structures make heavy use of pointers and dynamically allocated memory.
- ▶ Recall: Our RAM model supports
 - ▶ basic pseudocode (\approx simple Python/Java code)
 - ▶ creating arrays of a fixed/known size.
 - ▶ creating instances (objects) of a known class.

Recap: The Random Access Machine

- ▶ Data structures make heavy use of pointers and dynamically allocated memory.
- ▶ Recall: Our RAM model supports
 - ▶ basic pseudocode (\approx simple Python/Java code)
 - ▶ creating arrays of a fixed/known size.
 - ▶ creating instances (objects) of a known class.



Python abstracts this away!

no predefined capacity!

There are **no arrays in Python, only its built-in lists.**

But: Python *implementations* create lists based on fixed-size arrays (stay tuned!)



Python \neq RAM:

Not every built-in Python instruction runs in $O(1)$ time!

3.1 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, Python ABCs
(with comments)

VS.

data structures

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

≈ Java/Python class
(non abstract)

abstract base classes



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, Python ABCs
(with comments)

abstract base classes



VS.

data structures

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

≈ Java/Python class
(non abstract)

Why separate?

- ▶ Can swap out implementations ↵ “drop-in replacements”
- ↪ **reusable code!**
- ▶ (Often) better abstractions
- ▶ Prove generic lower bounds (↵ Unit 4 Unit 8)

Abstract Data Types

abstract data type (ADT)

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

abstract
data type

- ≈ Java interface, Python ABCs
(with comments)

Why separate?

- ▶ Can swap out implementations
- ≈ **reusable code!**
- ▶ (Often) better abstractions
- ▶ Prove generic lower bounds (≈ Unit 3)



Clicker Question

Which of the following are examples of abstract data types?



- | | | | |
|----------------------------|--------------------|----------------------------|-------------------------|
| <input type="checkbox"/> A | ADT | <input type="checkbox"/> G | resizable array |
| <input type="checkbox"/> B | Stack | <input type="checkbox"/> H | heap |
| <input type="checkbox"/> C | Deque | <input type="checkbox"/> I | priority queue |
| <input type="checkbox"/> D | Linked list | <input type="checkbox"/> J | dictionary/symbol table |
| <input type="checkbox"/> E | binary search tree | <input type="checkbox"/> K | hash table |
| <input type="checkbox"/> F | Queue | | |



→ *sli.do/cs566*

Clicker Question

Which of the following are examples of abstract data types?

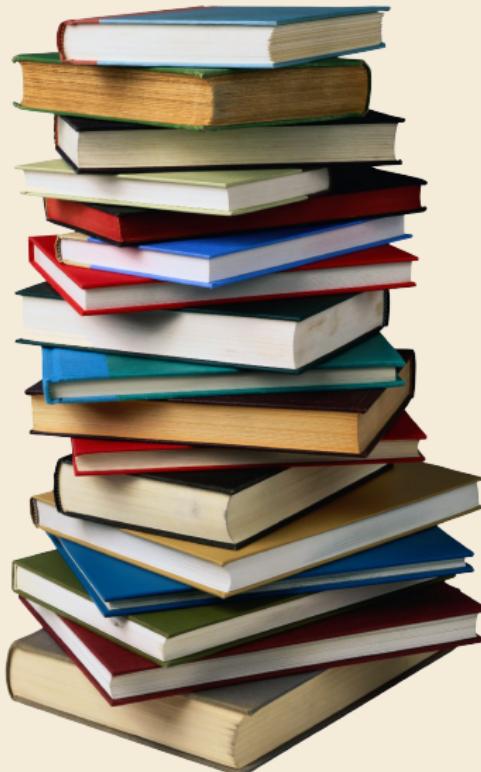


- A ~~ADT~~
- B Stack ✓
- C Deque ✓
- D ~~Linked list~~
- E ~~binary search tree~~
- F Queue ✓
- G ~~resizable array~~
- H ~~heap~~
- I priority queue ✓
- J dictionary/symbol table ✓
- K ~~hash table~~



→ *sli.do/cs566*

Stacks



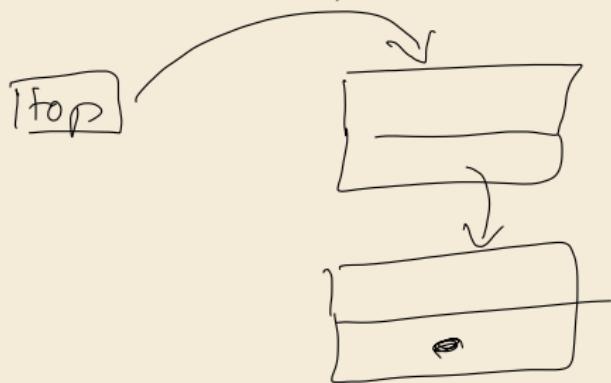
Stack ADT

- ▶ `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.

Linked-list implementation for Stack

Invariants:

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



```
1 class Node
2     value
3     next
4
5 class Stack
6     top := null
7     procedure top()
8         return top.value
9     procedure push(x)
10        top := new Node(x, top)
11    procedure pop()
12        t := top()
13        top := top.next
14        return t
```

Linked-list implementation for Stack – Discussion

Linked stacks:

 require $\Theta(n)$ space when n elements on stack

 All operations take $O(1)$ time

 require $\Theta(n)$ space when n elements on stack

Can we avoid extra space for pointers?

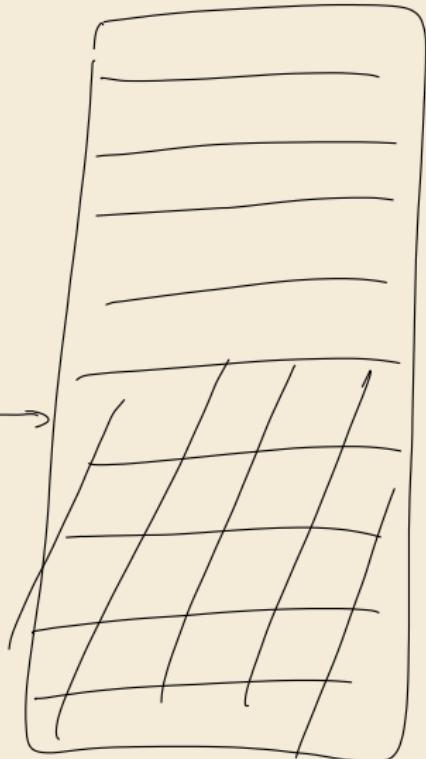
Array-based implementation for Stack

If we want no pointers \rightsquigarrow array-based implementation

Invariants:

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

top →



Array-based implementation for Stack

If we want no pointers \rightsquigarrow 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 push?

Array stacks:

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

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 `java.util.Collection`

Python's `collections.abc.Collection`



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

3.2 Resizable Arrays

Digression – Arrays as ADT

Arrays can also be seen as an ADT!

Array operations:

- ▶ `create(n)` *Java: A = new int[*n*]; Python: A = [0] * *n**

Create a new array with *n* cells, with positions $0, 1, \dots, n - 1$;
we write $\underbrace{A[0..n)} = A[0..n - 1]$

- ▶ `get(i)` *Java/Python: A[*i*]*

Return the content of cell *i*

- ▶ `set(i, x)` *Java/Python: A[*i*] = *x*;*

Set the content of cell *i* to *x*.

↝ Arrays have *fixed* size (supplied at creation). (\neq lists in Python)

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*]; Python: A = [0] * *n**

Create a new array with *n* cells, with positions $0, 1, \dots, n - 1$;
we write $A[0..n) = A[0..n - 1]$

- ▶ `get(i)` *Java/Python: A[*i*]*

Return the content of cell *i*

- ▶ `set(i, x)` *Java/Python: A[*i*] = *x*;*

Set the content of cell *i* to *x*.

⇝ Arrays have *fixed* size (supplied at creation). (\neq lists in Python)

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



Difference to “real” ADTs: *Implementation usually fixed*
to “a contiguous chunk of memory”.

Doubling trick

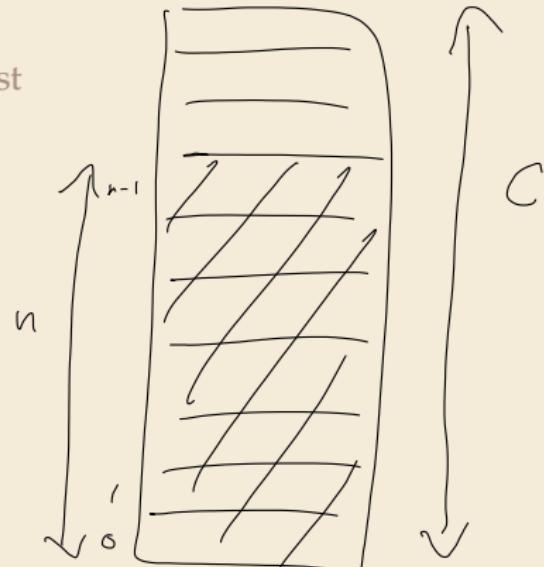
Can we have unbounded stacks based on arrays? Yes!

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 $C = S.length$ so that $\frac{1}{4}C \leq n \leq C$
- ~~ 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 $\frac{1}{4}C \leq n \leq C$
- ~~ can always push more elements!

for n elements we use $\Theta(n)$ space

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 ^{always} 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/cs566*

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!

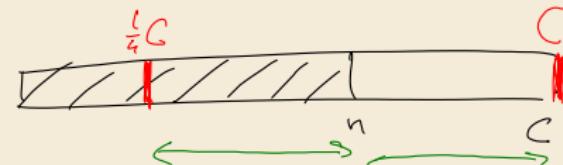
Amortized analysis w/ potential function

c_i = actual cost of operation i

Φ_i = potential after operation i

a_i = amortized cost of operation i

$$:= c_i - 4(\Phi_i - \Phi_{i-1})$$



Φ_i = distance to dangerous boundary = $\min\{C-n, n-\frac{1}{4}C\}$

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!

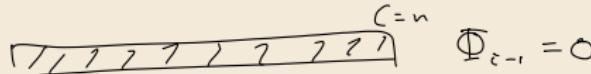
Formally: consider “credits/potential” $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

distance to boundary since $n \leq C \leq 4n$

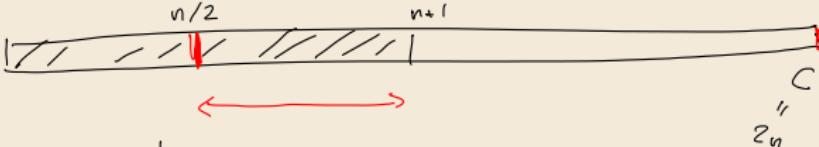
- ▶ amortized cost of an operation = actual cost (array accesses) $- 4 \cdot$ change in Φ
 - ▶ cheap push/pop: actual cost 1 array access, consumes ≤ 1 credits \rightsquigarrow amortized cost $\underline{\leq 5}$
 - ▶ copying push: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n + 1$ credits \rightsquigarrow amortized cost $\underline{\leq 5}$
 - ▶ copying pop: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n - 1$ credits \rightsquigarrow amortized cost $\underline{5}$
- \rightsquigarrow **sequence of m operations**: total actual cost \leq total amortized cost + final credits
here: $\leq 5m + 4 \cdot 0.6n = \Theta(m + n)$

copying push :

before



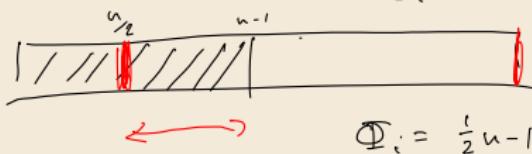
after



$$\Phi_i = \frac{1}{2}n + 1$$

$$a_i = 2n+1 - 4\left(\frac{1}{2}n + 1\right) = 1 - 4 = -3 \leq 5$$

copying pop



$$a_i = 2n+1 - 4\left(\frac{1}{2}n - 1\right) = 5$$

$$\begin{aligned}
 5m &\geq \sum_{i=1}^m a_i = \sum_{i=1}^m c_i - 4(\Phi_i - \Phi_{i-1}) \\
 &= \sum_{i=1}^m c_i - 4 \underbrace{\sum_{i=1}^m (\Phi_i - \Phi_{i-1})}_{\text{telescoping sum}} \\
 &= " - 4(\Phi_m - \Phi_0) \quad \sum_{i=1}^m \cancel{\Phi_i} - \cancel{\Phi_{i-1}}
 \end{aligned}$$

$$\begin{aligned}
 \Leftrightarrow \sum_{i=1}^m c_i &\leq 5m + 4(\Phi_m - \Phi_0) \\
 &\leq 5m + 4 \cdot 0.6n = 0
 \end{aligned}$$

~~$= (\Phi_4 - \Phi_3)$~~
 ~~$+ (\Phi_3 - \Phi_2)$~~
 ~~$+ (\Phi_2 - \Phi_1)$~~
 ~~$+ (\Phi_1 - \Phi_0)$~~

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/cs566*

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 $O(n + m)$. ✓ \Rightarrow "O(1) amortized per operation"
- D ~~Inserting and deleting any element takes $O(1)$ amortized time.~~



→ *sli.do/cs566*

Deamortized Resizable Arrays

What if we need $O(1)$ worst case time?

Deamortized Resizable Arrays

What if we need $O(1)$ worst case time?

- ▶ It's possible to *de-amortize* the resizing arrays solution!
- ▶ maintain 3 arrays: S (as before) and S_2 and $S_{1/2}$
of twice and half the size of S

Deamortized Resizable Arrays

What if we need $O(1)$ worst case time?

- ▶ It's possible to *de-amortize* the resizing arrays solution!
- ▶ maintain 3 arrays: S (as before) and S_2 and $S_{1/2}$
of twice and half the size of S
- ▶ write operations go to all 3 arrays
- ▶ upon resize, "shift" arrays up/down $\rightsquigarrow S_2$ resp. $S_{1/2}$ become new S
 - ▶ allocate new array, but **delay filling it with elements** ← general strategy!
 - ▶ every insert or delete copies 2 slots from last resize
- \rightsquigarrow by time for next resize, we have caught up and S_2 resp. $S_{1/2}$ ready to use

Deamortized Resizable Arrays

What if we need $O(1)$ worst case time?

- ▶ It's possible to *de-amortize* the resizing arrays solution!
 - ▶ maintain 3 arrays: S (as before) and S_2 and $S_{1/2}$
of twice and half the size of S
 - ▶ write operations go to all 3 arrays
 - ▶ upon resize, "shift" arrays up/down $\rightsquigarrow S_2$ resp. $S_{1/2}$ become new S
 - ▶ allocate new array, but **delay filling it with elements** general strategy!
 - ▶ every insert or delete copies 2 slots from last resize
- \rightsquigarrow by time for next resize, we have caught up and S_2 resp. $S_{1/2}$ ready to use

Analysis:

- ▶ $O(1)$ worst case time for read/write by index, push, and pop!
- ▶ up to 7 array accesses per operation
- ▶ up to $7n$ space other time-space trade-offs possible

Rabbit Hole: Can we do this more space-efficiently?

- ▶ It might appear as if every efficient implementation of a stack needs $\Omega(n)$ extra space on top of space for storing the n elements in the stack.

Rabbit Hole: Can we do this more space-efficiently?

- ▶ It might appear as if every efficient implementation of a stack needs $\Omega(n)$ extra space on top of space for storing the n elements in the stack.
- ▶ But this is not true!

Rabbit Hole: Can we do this more space-efficiently?

- ▶ It might appear as if every efficient implementation of a stack needs $\Omega(n)$ extra space on top of space for storing the n elements in the stack.
- ▶ But this is not true!
- ▶ Can get operations in $O(1)$ worst-case time with $O(\sqrt{n})$ extra space at any time (!)
 - ▶ Maintain a collection of small arrays (plus header with pointers to them)
 - ▶ Clever choice of block sizes guarantees
 $\overbrace{O(\sqrt{n}) \text{ blocks of } O(\sqrt{n}) \text{ elements throughout}}$
and fast calculation of address for an index.
imaginary “superblocks” of sizes 2^k , $k = 0, 1, \dots, \lg n$
 k th superblock consists of $2^{k/2}$ actual blocks of $2^{k/2}$ elements each.
 - ▶ $O(\sqrt{n})$ extra space is best possible

↙ exam

Resizable Arrays in Optimal Time and Space

Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro & Robert Sedgewick

WADS 1999

3.3 Priority Queues & Binary Heaps

Clicker Question



What is a heap-ordered tree?

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



→ *sli.do/cs566*

Priority Queue ADT

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

(Max-oriented) Priority Queue (MaxPQ):

▶ `construct(A)`

Construct 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*.

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

► `construct(A)`

Construct from elements in array *A*.

► `insert(x, p)`

Insert item *x* with priority *p* into PQ.

► `min()`

Return item with ~~largest~~^{smallest} priority. (Does not modify the PQ.)

► `delMin()`

Remove the item with ~~largest~~^{smallest} 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.



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 (both linked & array)

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

PQ implementations

Elementary implementations

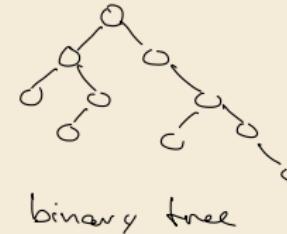
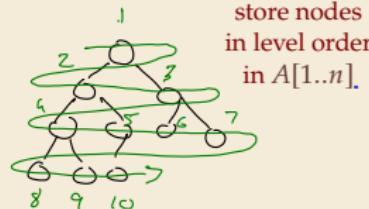
- unordered list $\rightsquigarrow \Theta(1)$ insert, but $\Theta(n)$ delMax
- sorted list $\rightsquigarrow \Theta(1)$ delMax, but $\Theta(n)$ insert

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

Yes! *Binary heaps*.

Array view

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



complete binary tree

full
left-aligned



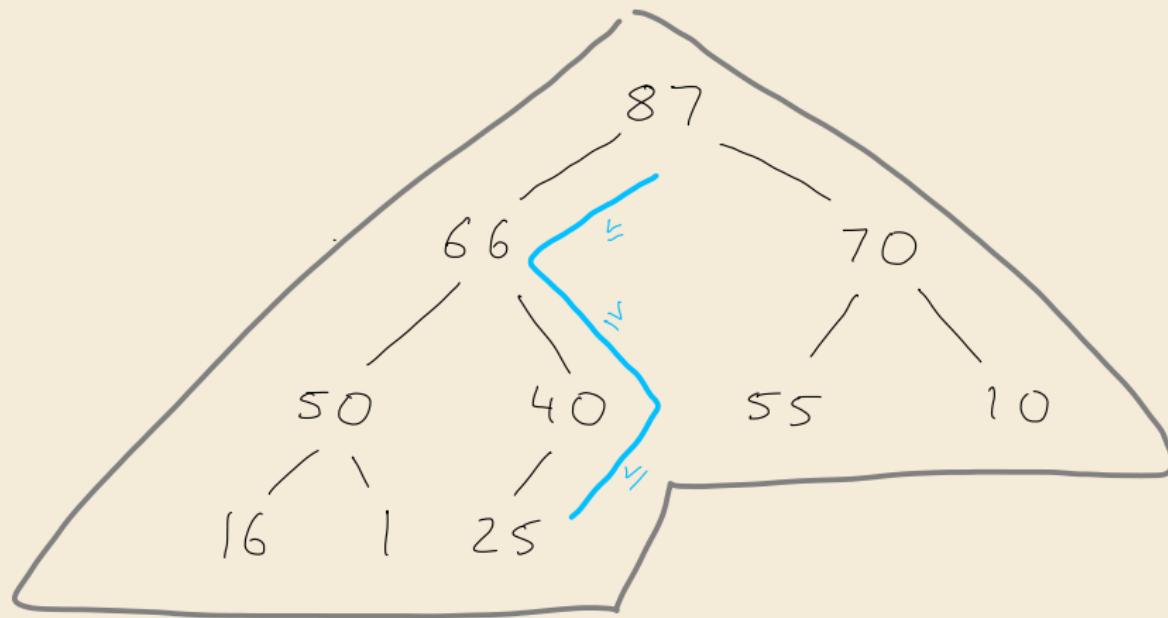
Tree view

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

all but last level full
last level flush left

parent \geq children

Binary heap example



Why heap-shaped trees?

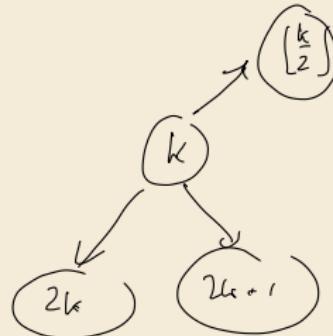
Why complete binary tree shape?

- ▶ only one possible tree shape \rightsquigarrow 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

▲ Recall: nodes at indices [1..n]

- ▶ parent at $\lfloor k/2 \rfloor$ (for $k \geq 2$)
- ▶ left child at $2k$
- ▶ right child at $2k + 1$



Why heap-shaped trees?

Why complete binary tree shape?

- ▶ only one possible tree shape \rightsquigarrow 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

▲ Recall: nodes at indices [1..n]

- ▶ parent at $\lfloor k/2 \rfloor$ (for $k \geq 2$)
- ▶ left child at $2k$
- ▶ right child at $2k + 1$

Why heap ordered?

- ▶ Maximum must be at root! \rightsquigarrow `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 smaller than the key at the root. ✓
- D ~~An tree that is stored in the heap area of the memory.~~

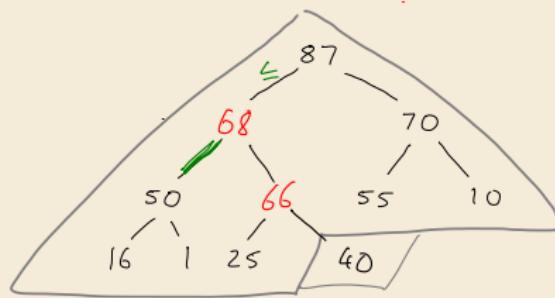
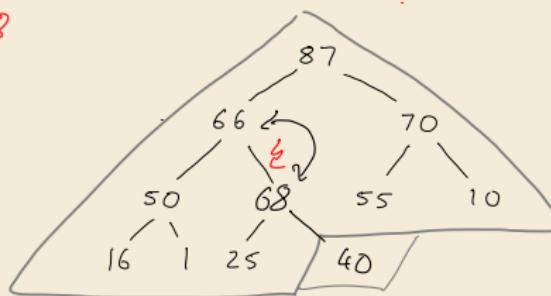
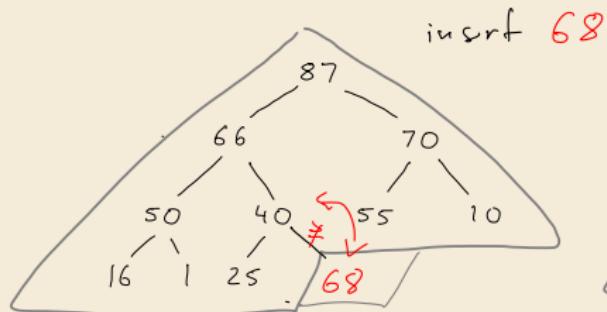


→ *sli.do/cs566*

3.4 Operations on Binary Heaps

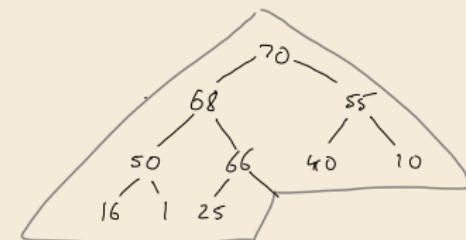
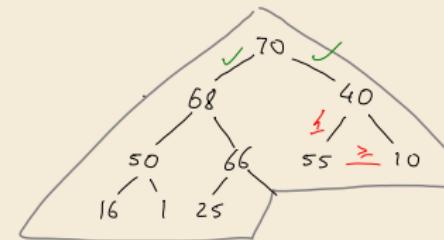
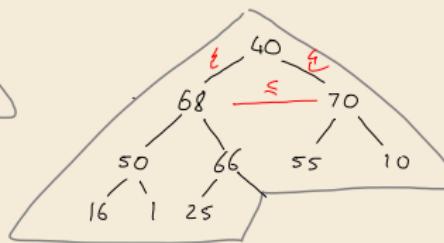
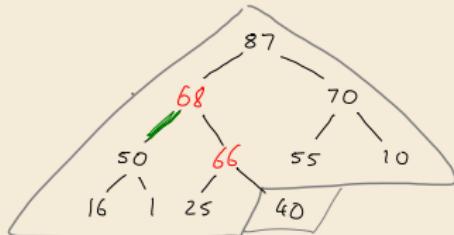
Insert

1. Add new element at only possible place: bottom-most level, next free spot.
2. Let element *swim* up to repair heap order.



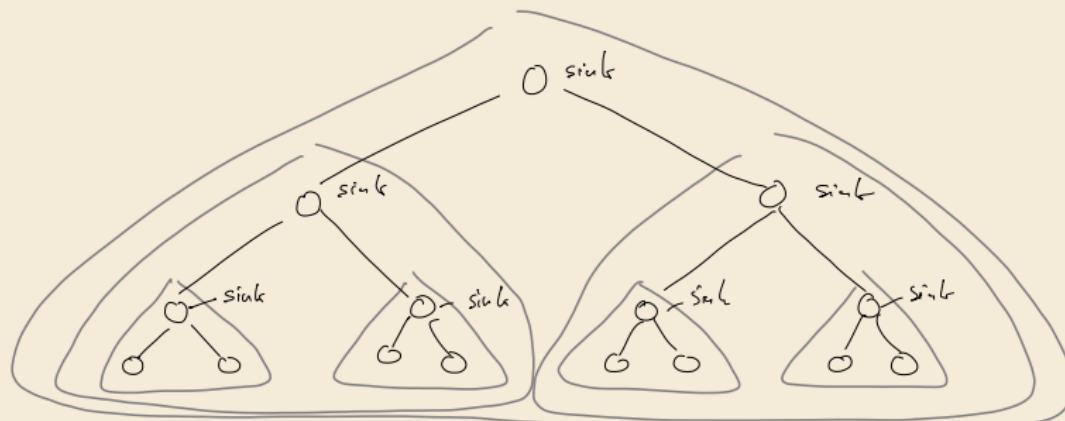
Delete Max

1. Remove max (must be in root).
2. Move last element (bottom-most, rightmost) into root.
3. Let root key *sink* in heap to repair heap order.



Heap construction

- ▶ n times insert $\rightsquigarrow \Theta(n \log n)$ 
- ▶ instead:
 1. Start with singleton heaps (one element)
 2. Repeatedly merge two heaps of height k with new element into heap of height $k + 1$



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?
$$\sum_{\ell=0}^k 2^\ell = 2^{k+1} - 1$$

 \rightsquigarrow Height of binary heap: $h = \min k$ 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?
$$\sum_{\ell=0}^k 2^\ell = 2^{k+1} - 1$$

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

Analysis:

- ▶ **insert**: new element “swims” up $\rightsquigarrow \leq h$ steps (h cmps)
- ▶ **delMax**: last element “sinks” down $\rightsquigarrow \leq h$ steps ($2h$ cmps)
- ▶ **construct** from n elements:

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

$$\begin{aligned} &\leq 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^\ell \cdot (h-\ell) + \cdots + 2^{h-1} \cdot 1 + 2^h \cdot 0 \\ &= \sum_{\ell=0}^h 2^\ell (h-\ell) = \sum_{i=0}^h \frac{2^h}{2^i} i = 2^h \sum_{i=0}^h \frac{i}{2^i} \leq 2 \cdot 2^h \leq \underbrace{4n}_{\text{4n}} \end{aligned}$$

Binary heap summary

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

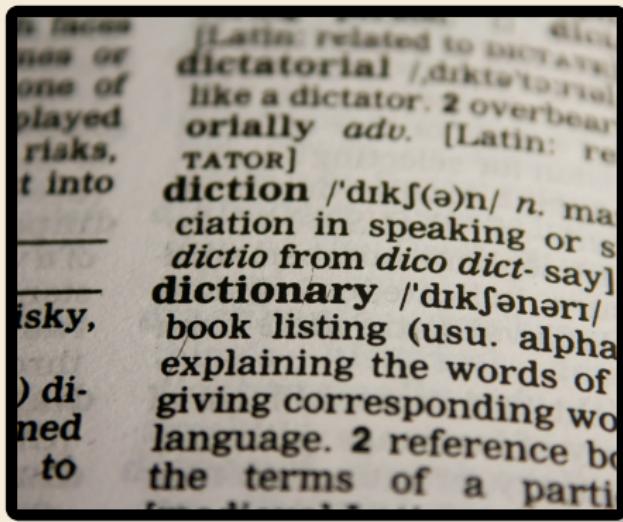
3.5 Symbol Tables

Symbol table ADT

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

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

Python dict {k:v}



- ▶ `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)` Python dict: `del d[k]`
Remove key k (any associated value) from table
- ▶ `contains(k)` Python dict: `k in d`
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.)

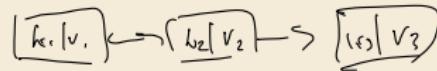
Symbol tables vs. mathematical functions

- ▶ similar interface
- ▶ but: mathematical functions are *static/immutable* (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

⇝ 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:

assumes keys are ordered

👎 $\Theta(n)$ time for put

👎 $\Theta(n)$ time for get

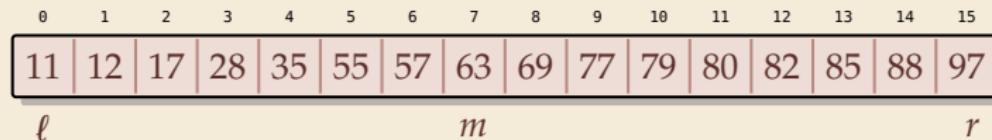
~~> Too slow to be useful

~~> *Sorted order does not help us at all?!*

Binary search

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

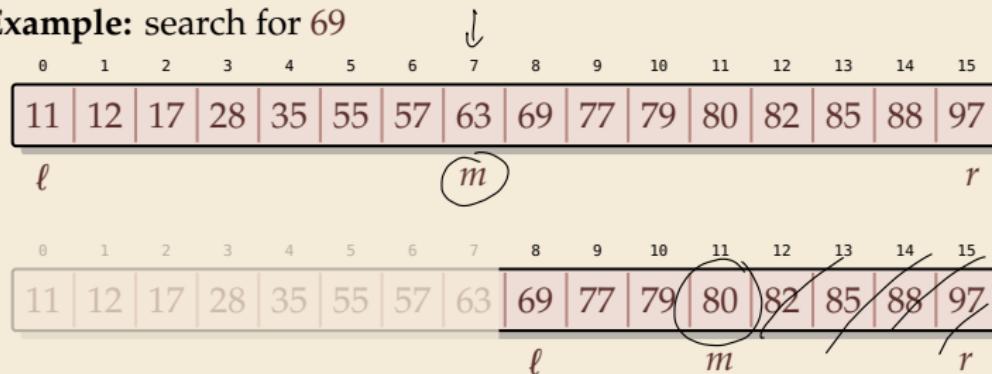
Example: search for 69



Binary search

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

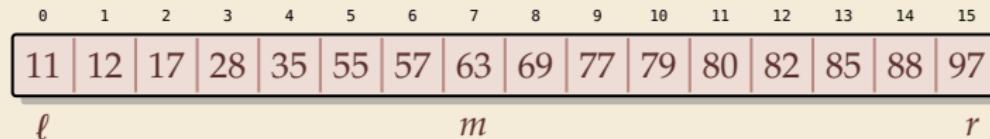
Example: search for 69



Binary search

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

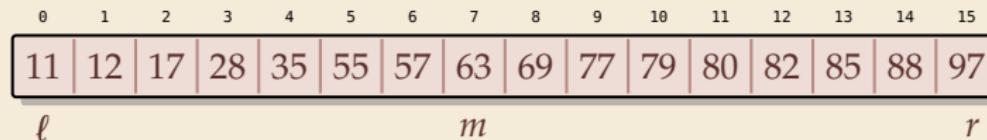
Example: search for 69



Binary search

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

Example: search for 69



Binary search

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

unsuccessful search (search miss)
successful search (search hit)
↳ best case 1

Example: search for 69.1



Binary search:

- ▶ halve remaining list in each step
 $\approx \log_2 <, =, >$
- $\rightsquigarrow \leq \lfloor \lg n \rfloor + 1 \text{ cmps}$
in the worst case



needs random access!

3.6 Binary Search Trees

Clicker Question



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

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



→ *sli.do/cs566*

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/cs566*

Binary search trees

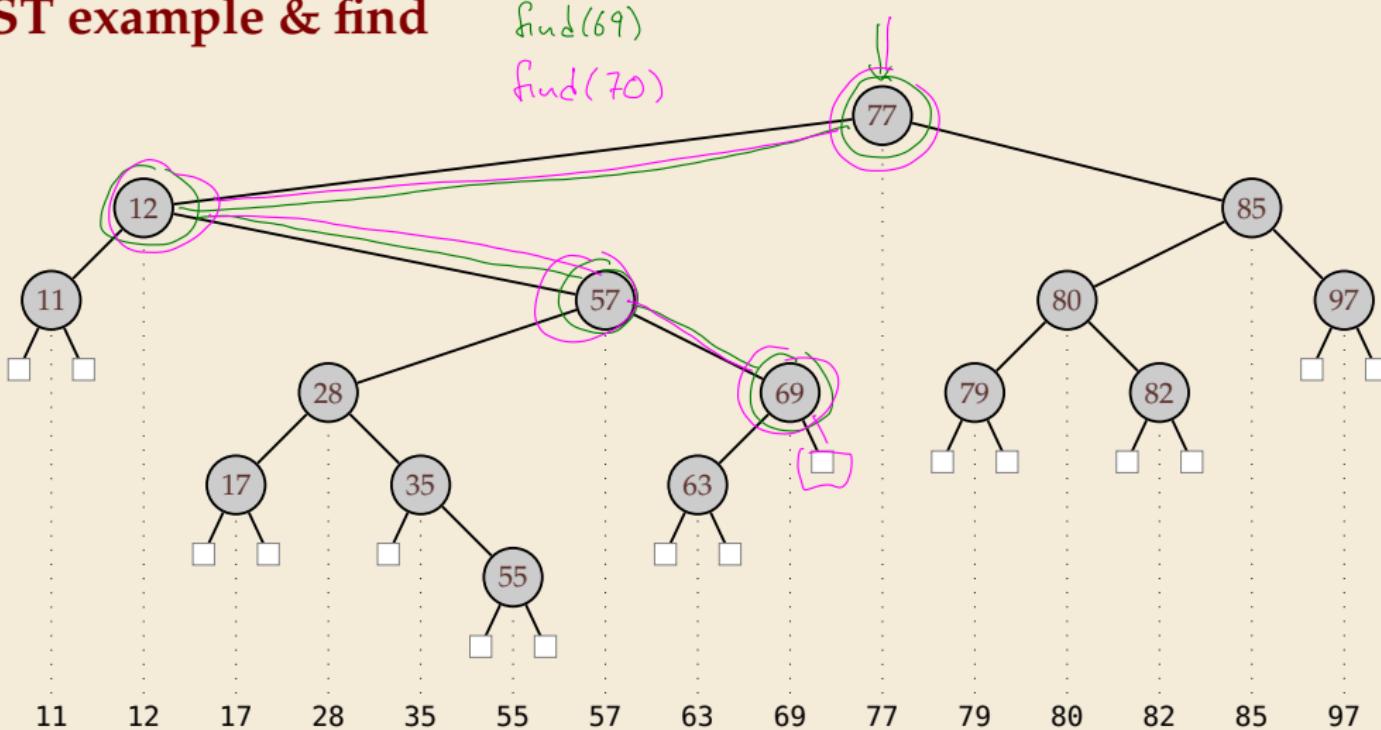
Binary search trees (BSTs) \approx dynamic sorted array

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

all keys in left subtree \leq root key \leq all keys in right subtree

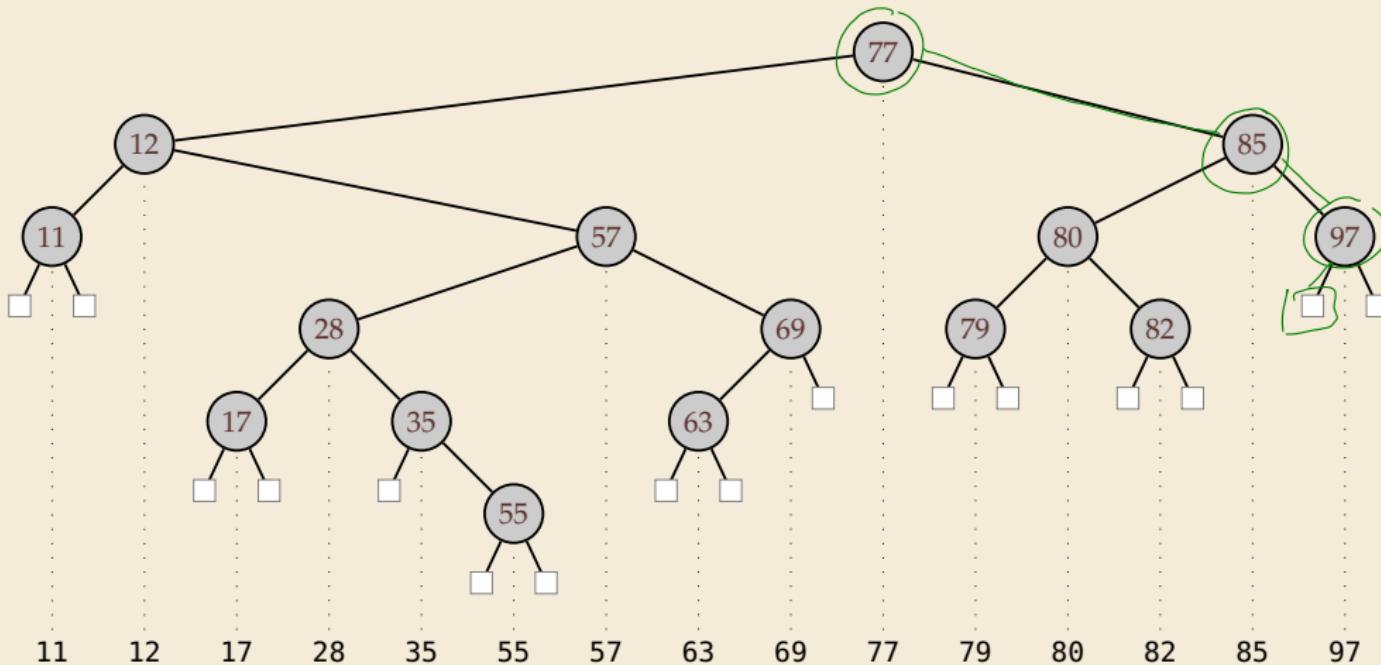


BST example & find



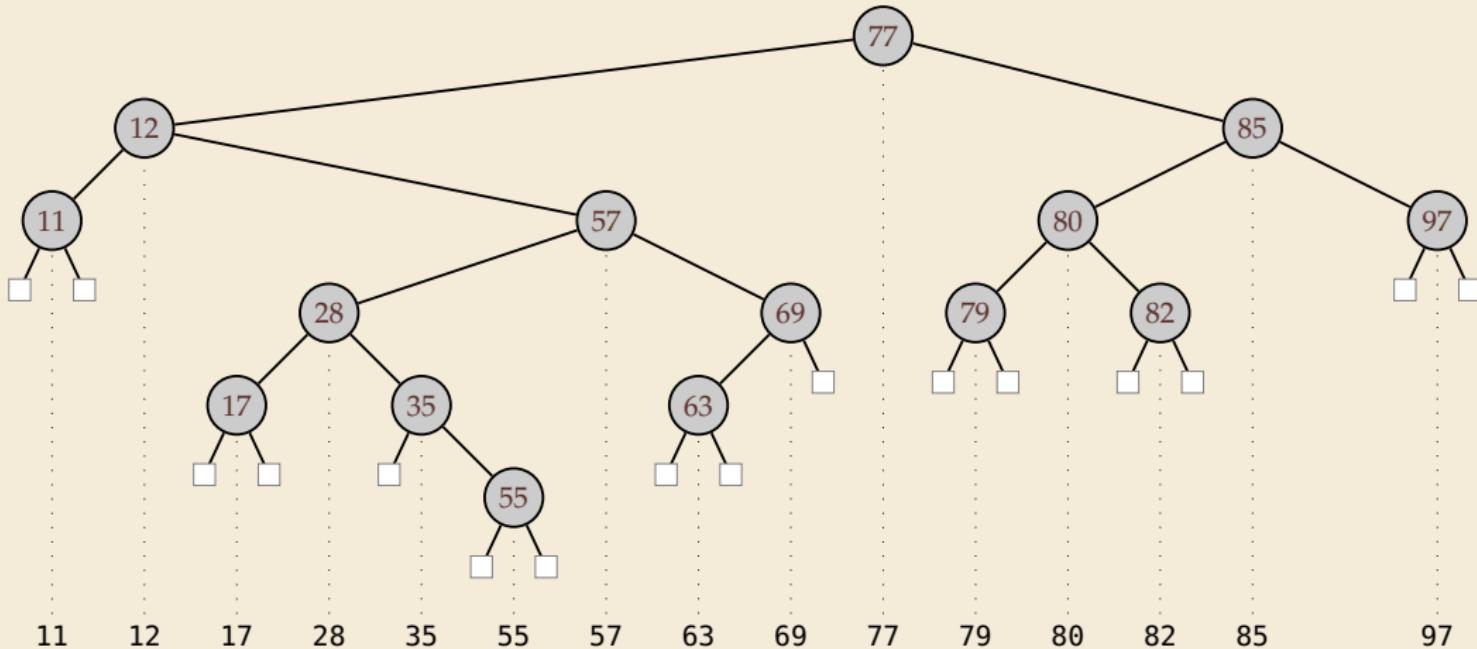
BST insert

Example: Insert 80



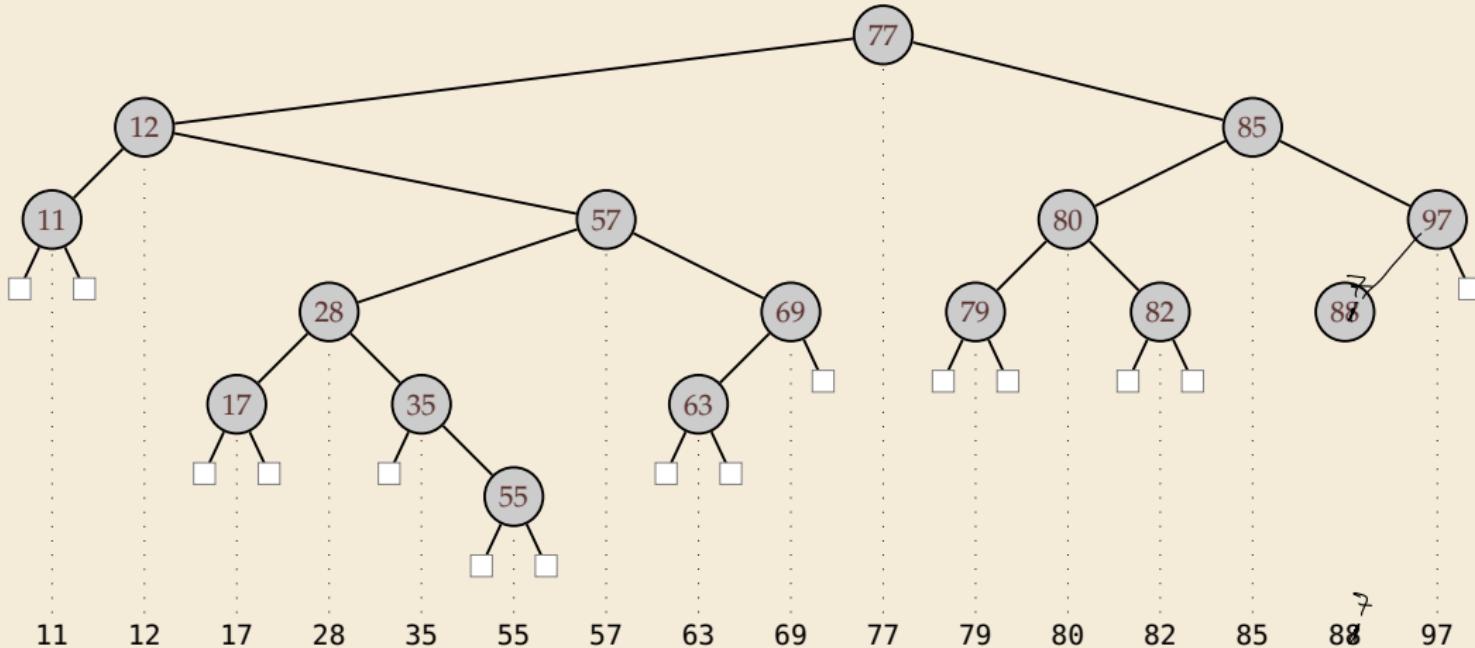
BST insert

Example: Insert 88



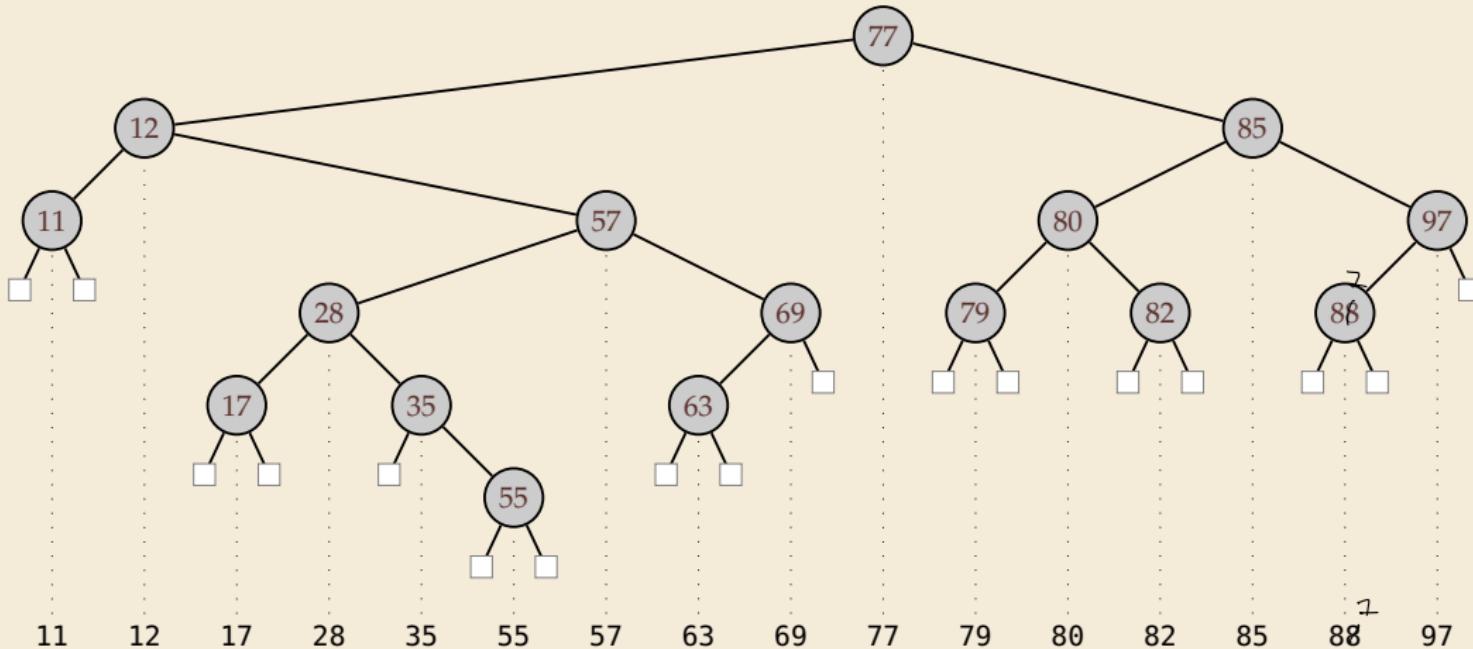
BST insert

Example: Insert 88



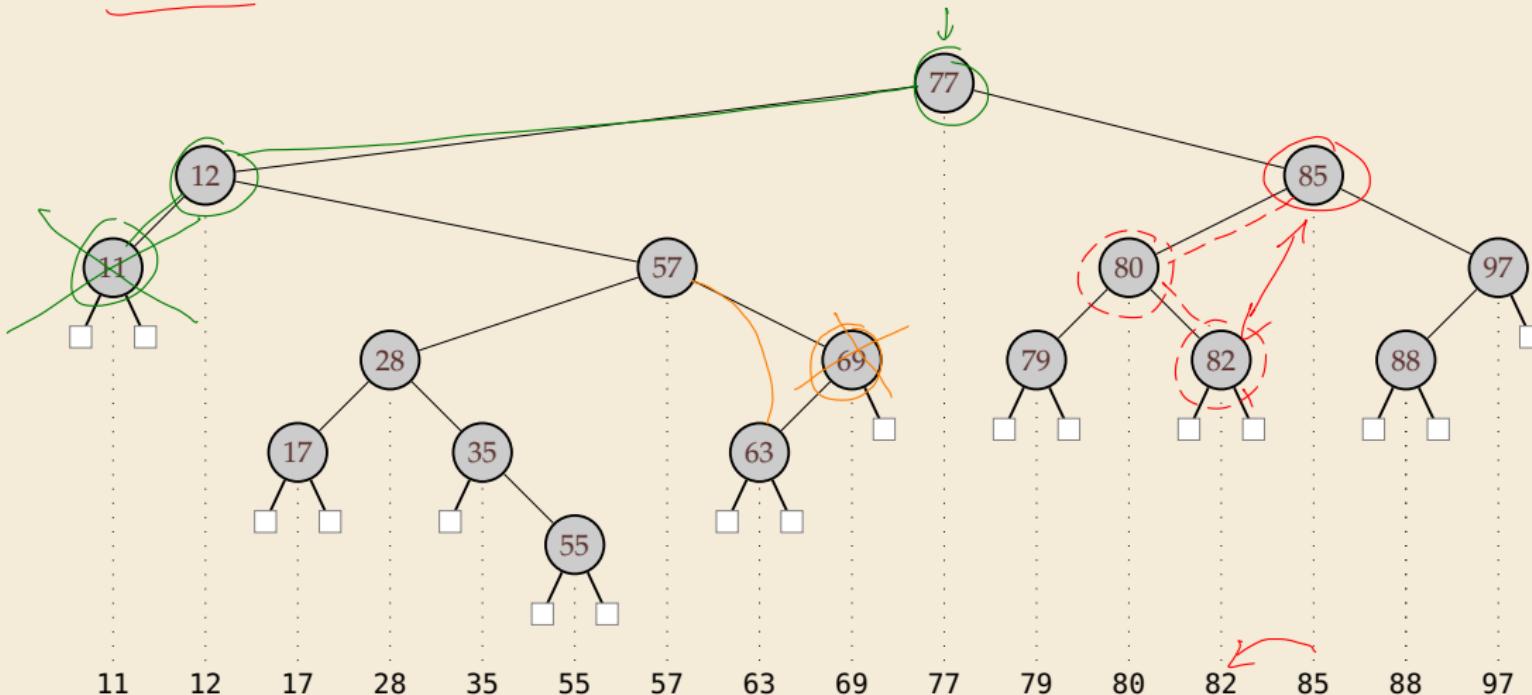
BST insert

Example: Insert 88



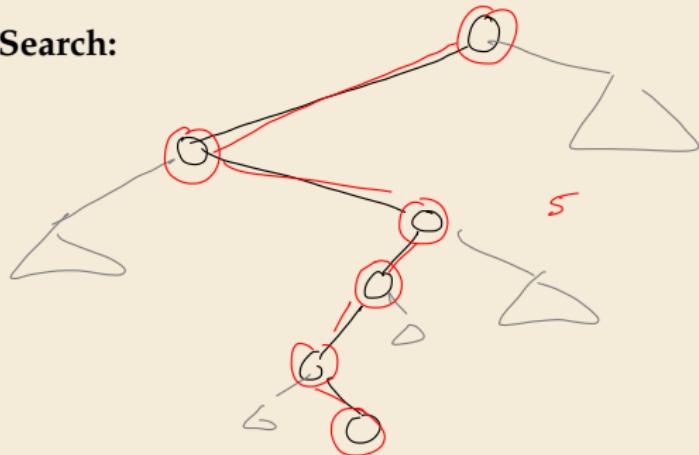
BST delete

- 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:



comps = length of
search path
+ 1
 $\leq \underbrace{\text{height of BST}}_h + 1$

total time $O(h)$

► Insert: $O(h)$ unsuccessful search \rightarrow one path + $O(1)$ effort

► Delete: $O(h)$ at most one root-to-leaf path

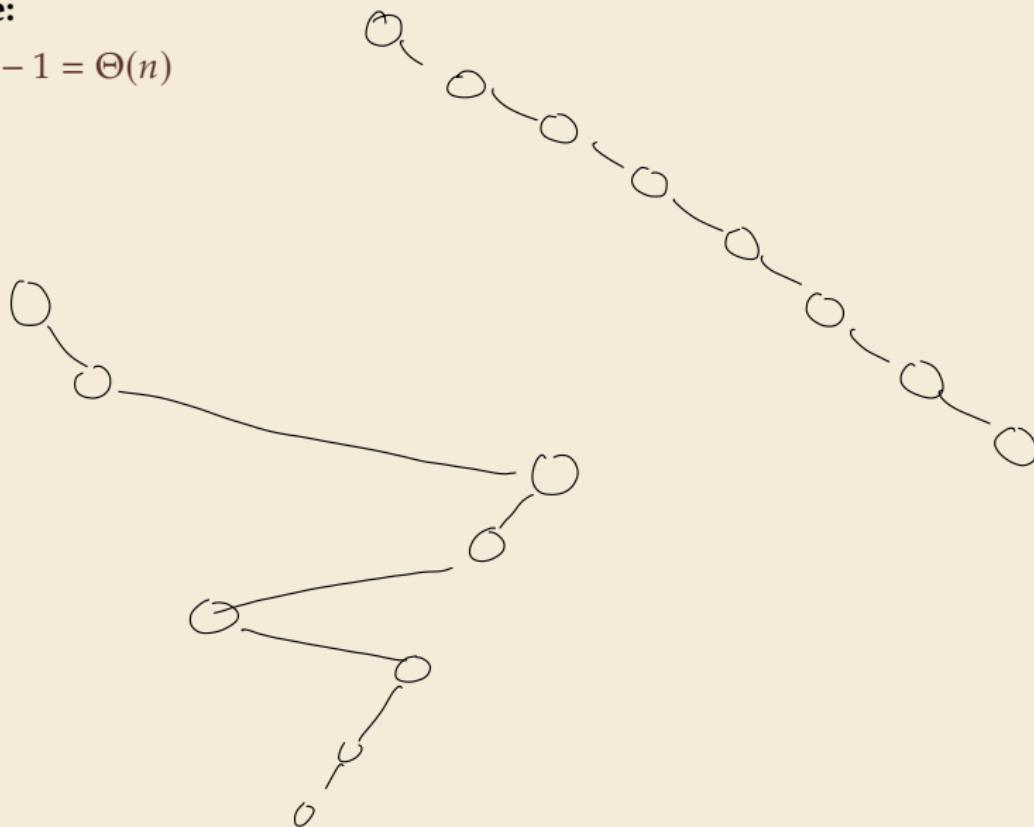
BST summary

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

What is the height of a BST?

Worst Case:

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



What is the height of a BST?

Worst Case:

► $h = n - 1 = \Theta(n)$

Average Case: \approx random permutation model

► Assumption: insertions come in random order
no deletions

$\leadsto h = \Theta(\log n)$ in expectation

even "with high probability":
 $\forall d \exists c : \Pr[h \geq c \lg(n)] \leq n^{-d}$

Precise analysis:

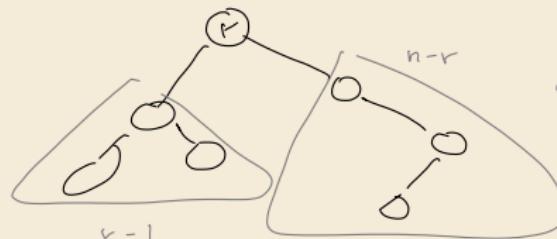
cost of finding random key in ST

easier: EXPECTED π cost of searching ALL keys in BST
||

C_n

$n = \# \text{keys}$

random: order of insertions
of $[n]$



again built
from random permutation

$$C_n = \sqrt{n} + \frac{1}{n} \sum_{r=1}^n [C_{r-1} + C_{n-r}] \quad (n \geq 1)$$

$$C_0 = 0$$

$$= 2 \sum_{i=0}^{n-1} C_i$$

① Full history

$$n \cdot C_n = n^2 + 2 \sum_{i=0}^{n-1} C_i$$

$$n C_n - (n-1) C_{n-1} = n^2 - (n-1)^2 + 2 C_{n-1}$$

② Telescoping

$$n C_n = (n+1) C_{n-1} + 2n - 1 \quad | : (n)(n+1) \quad (n \geq 1)$$

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2n-1}{n(n+1)}$$

$$D_n := \frac{C_n}{n+1} \quad D_n = D_{n-1} + \dots \quad (n \geq 1)$$

$$= \sum_{k=1}^n \frac{2k-1}{k(k+1)} \quad (\text{recursion} \rightsquigarrow \text{sum})$$

$$\frac{2k-1}{k(k+1)} = \frac{-1}{k} + \frac{3}{k+1}$$

$$D_n = \sum_{k=1}^n \left(\frac{3}{k+1} - \frac{1}{k} \right) = 3 \sum_{k=2}^{n+1} \frac{1}{k} - \left(\sum_{k=1}^n \frac{1}{k} \right)$$

$\approx H_n$

$$= \frac{3}{k+1} - 3 + 3H_n - H_n$$

$$= 2H_n - 3 + \frac{3}{k+1}$$

$$C_n = 2(n+1)H_n - 3(n+1) + 3 \underset{\text{TCSCS}}{\sim} 2n \ln n \approx 1.38 n \lg n$$

$$H_n \approx \ln n$$

3.7 Ordered Symbol Tables

Ordered symbol tables

- ▶ $\min()$, $\max()$

Return the smallest resp. largest key in the ST

- ▶ $\text{floor}(x)$, $\lfloor x \rfloor = \mathbb{Z}.\text{floor}(x)$

Return largest key k in ST with $k \leq x$.

- ▶ $\text{ceiling}(x)$

Return smallest key k in ST with $k \geq x$.

- ▶ $\text{rank}(x)$

Return the number of keys k in ST $k < x$.

- ▶ $\text{select}(i)$

Return the i th smallest key in ST (zero-based, i. e., $i \in [0..n]$)



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

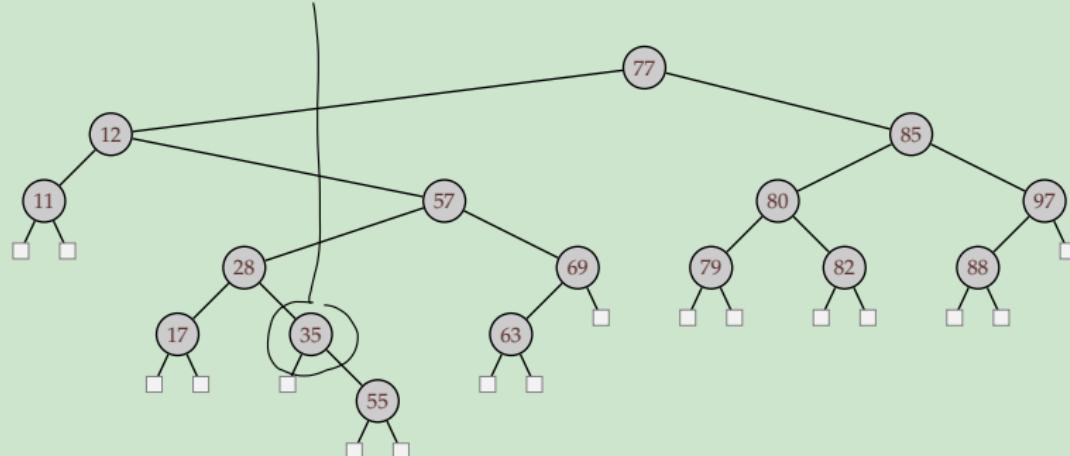
(Might not need any keys at all then!)

aside

nope: ds for stress

Clicker Question

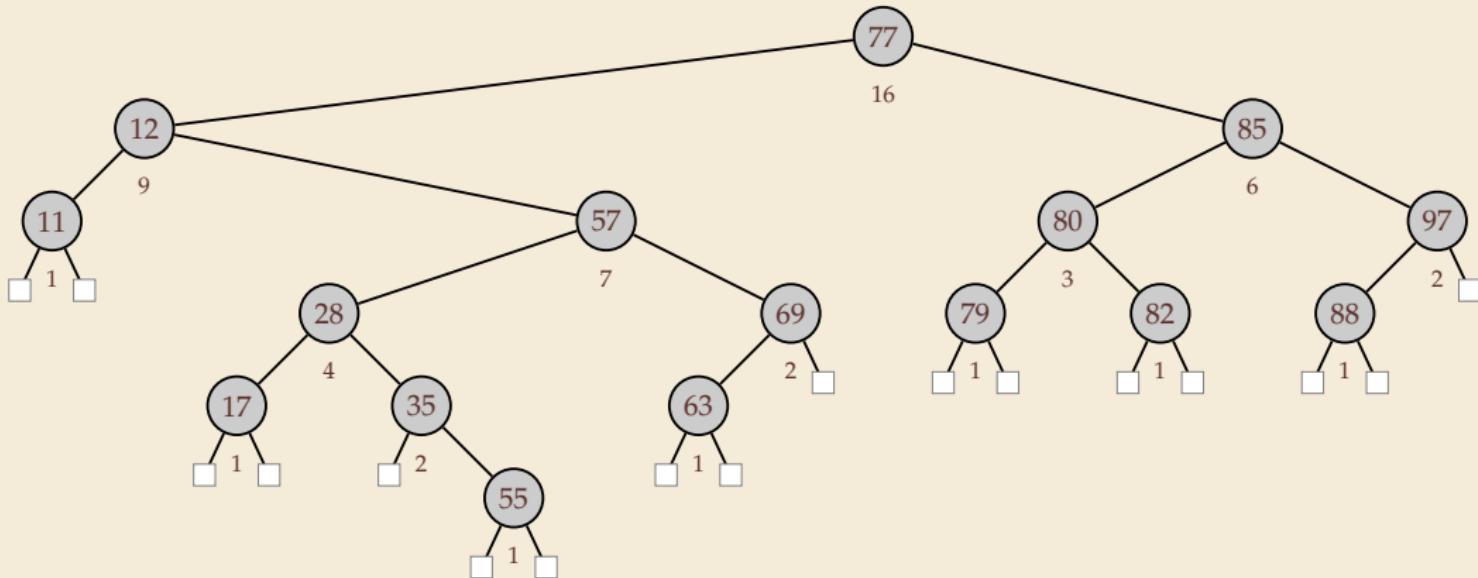
In the BST below, what would `rank(35)` return?



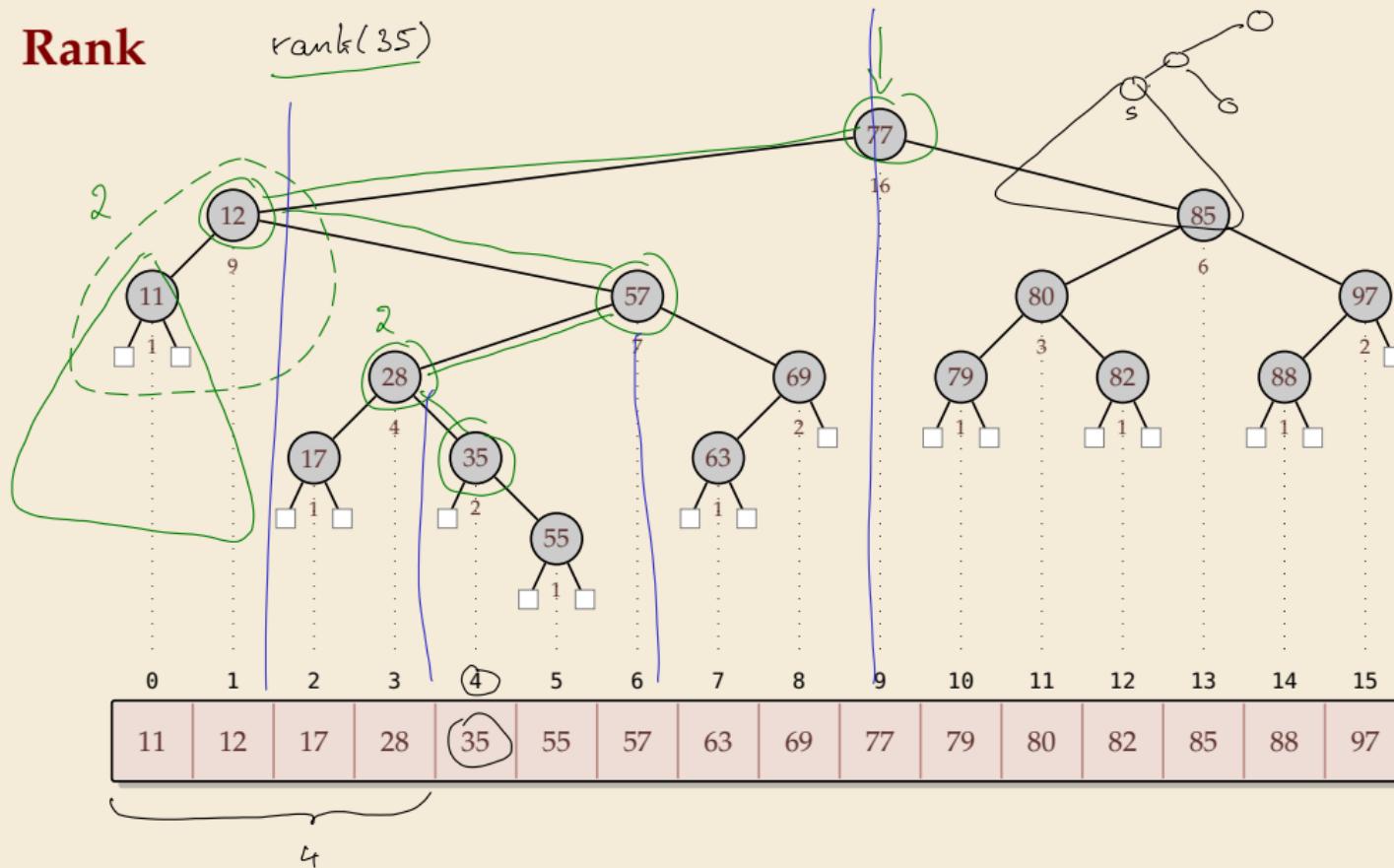
→ sli.do/cs566

Augmented BSTs

k
 $s = \text{subtree size}$

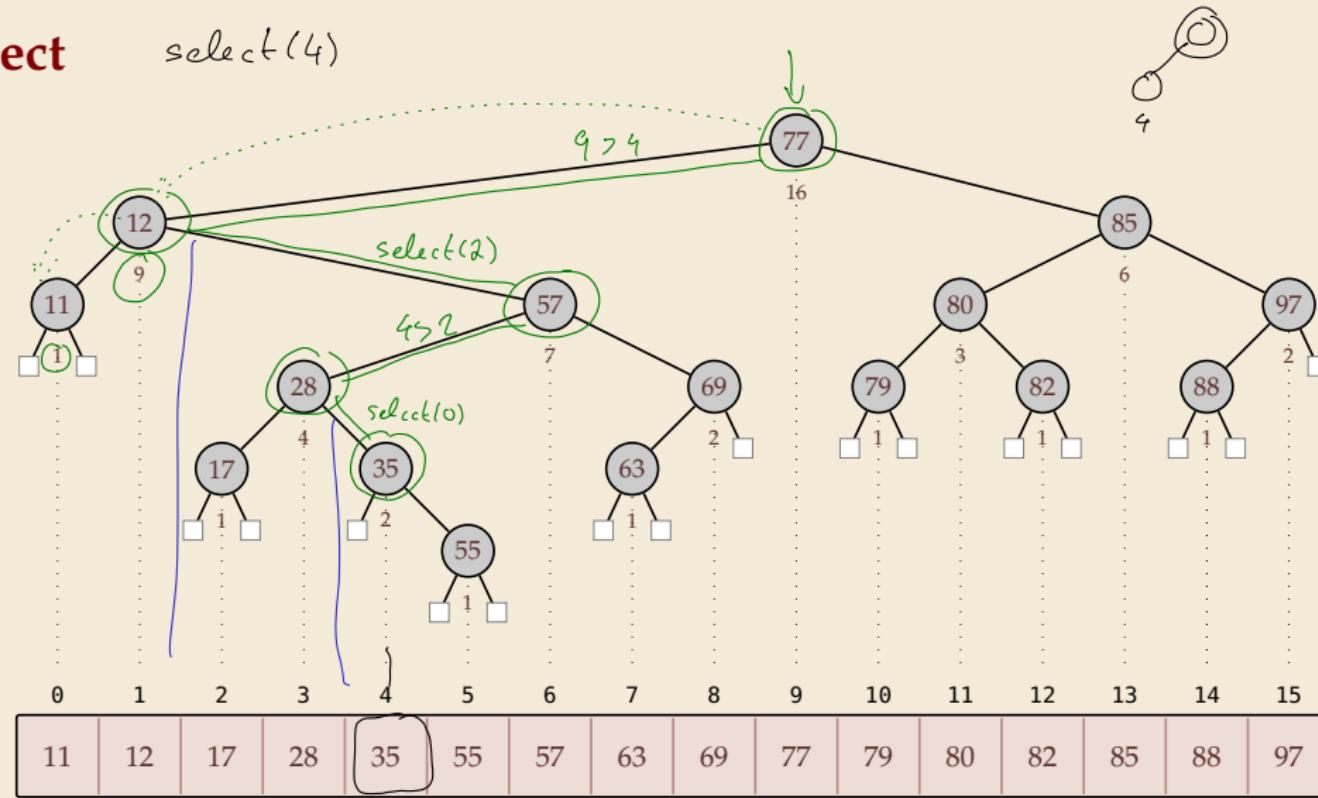


Rank



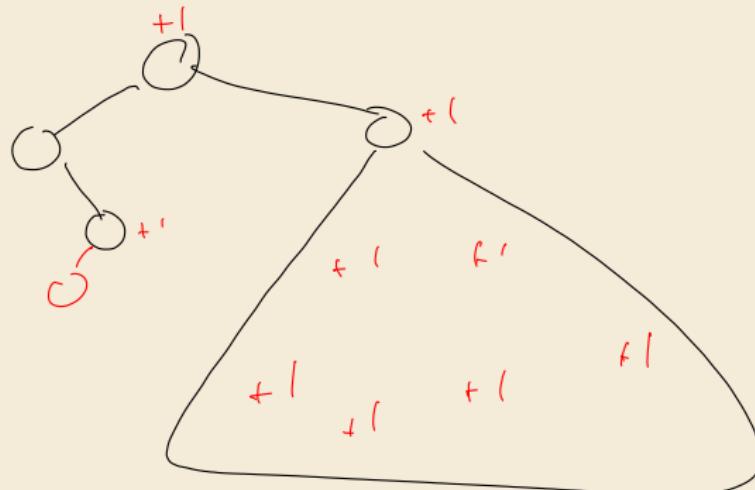
Select

select(4)



Why store subtree sizes?

- ▶ Note that in an augmented BST, each node stores the **size of its subtree**.
- ▶ ... why not directly store the **rank**? Would make rank/select much simpler!



Why store subtree sizes?

- ▶ Note that in an augmented BST, each node stores the **size of its subtree**.
 - ▶ ... why not directly store the **rank**? Would make rank/select much simpler!
 - ▶ Problem: Single insertion/deletion can change *all* node ranks!
 - ~~> Cannot efficiently maintain node ranks.
-  Subtree sizes only change along search path ~> $O(h)$ nodes affected

3.8 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/cs566*

Balanced BSTs

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* ($\text{BB}[\alpha]$ trees)
 - ▶ ...

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 (BB[α] trees)
 - ▶ ...

Other options:

- ▶ **amortization:** *splay trees, scapegoat trees*
COLA (cache oblivious lookahead array)
- ▶ **randomization:** *randomized BSTs, treaps, skip lists*

I'd love to talk more about all of these ...
(Maybe another time)

BSTs vs. Heaps

Balanced binary search tree

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

Binary heaps

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

BSTs vs. Heaps

Balanced binary search tree

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

Binary heaps

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

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

BSTs vs. Heaps

Balanced binary search tree

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

Binary heaps

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

- ▶ 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
<code>construct($A[1..n]$)</code>	$O(n \log n)$
<code>put(k, v)</code>	$O(\log n)$
<code>get(k)</code>	$O(\log n)$
<code>delete(k)</code>	$O(\log n)$
<code>contains(k)</code>	$O(\log n)$
<code>isEmpty()</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>min() / max()</code>	$O(\log n) \rightsquigarrow O(1)$
<code>floor(x)</code>	$O(\log n)$
<code>ceiling(x)</code>	$O(\log n)$
<code>rank(x)</code>	$O(\log n)$
<code>select(i)</code>	$O(\log n)$

~~Binary heaps~~ Strict Fibonacci heaps

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

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