

# 3 Fundamental Data Structures

21 October 2025

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];`



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



→ *sli.do/cs566*

# Clicker Question

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

Object[] A = new Object[n]; //  $n \cdot w$  bit



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



→ *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!)



Java

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 3)

# 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?



- 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*

# Clicker Question

Which of the following are examples of abstract data types?

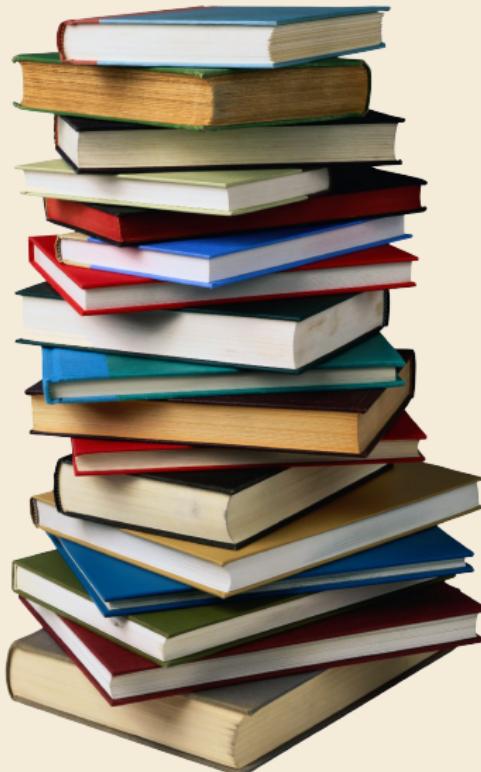


- A ~~ADT~~
- B Stack ✓
- C Deque ✓
- D ~~Linked list~~
- E ~~binary search tree~~
- F Queue ✓
- G ~~resizable array~~
- H ~~heap~~
- I priority queue ✓
- J dictionary/symbol table ✓
- K ~~hash table~~



→ *sli.do/cs566*

# Stacks



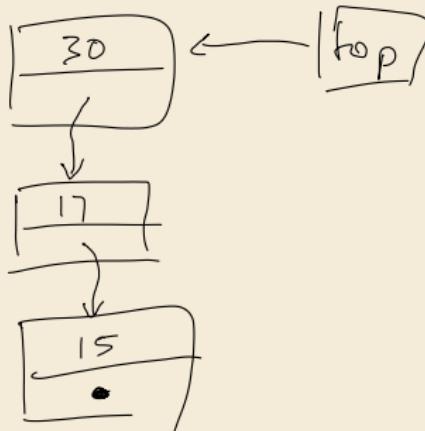
## Stack ADT

- ▶ `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

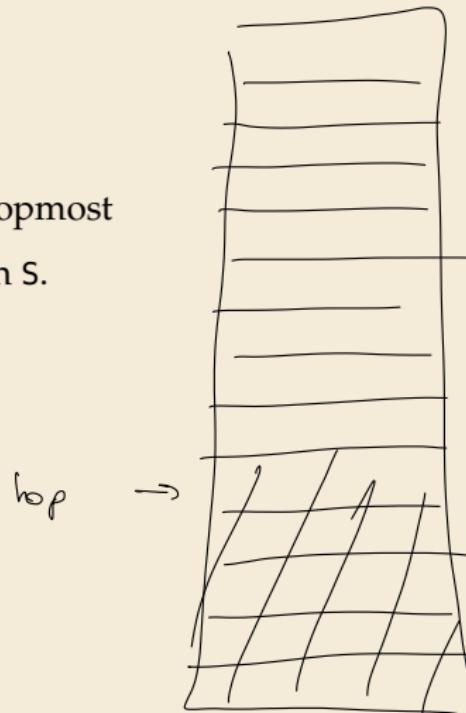
Can we avoid extra space for pointers?

# 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  $S$ .



# 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  $\underline{O(1)}$  time

# Queues

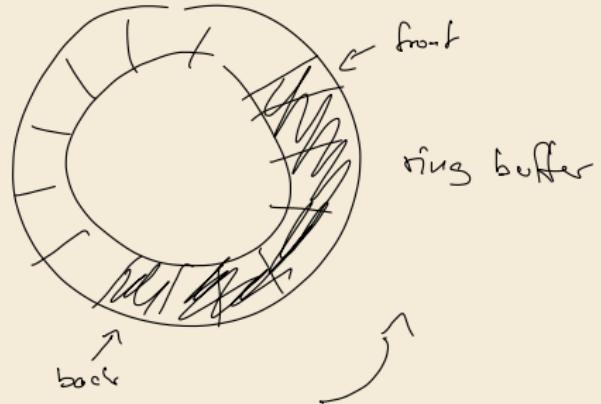
## Operations:

- ▶ 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**!

**Update 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`
- ▶ always support iterating over content (read only)

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  $A[0..n) = A[0..n - 1]$
  - ▶ `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!

*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?

*always*

- 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*

# 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!

Amortisierte Analyse       $c_i$  = echte Kosten von Operation  $i$

$\Phi_i$  = "Potential" nach Operation  $i$

$a_i$  = amortisierte Kosten

$$:= c_i + \alpha \cdot (\Phi_i - \Phi_{i-1})$$

Ziel: Zeigen, dass  $a_i \leq A$

$$m \cdot A \geq \sum_{i=1}^m a_i = \sum_{i=1}^m \underbrace{(c_i + \alpha(\Phi_i - \Phi_{i-1}))}_{\text{Teleskopsumme!}} = \sum_{i=1}^m c_i + \alpha(\Phi_m - \Phi_0)$$

$$\Rightarrow \sum_{i=1}^m c_i \leq m \cdot A - \alpha (\Phi_m - \Phi_0)$$

||      ||

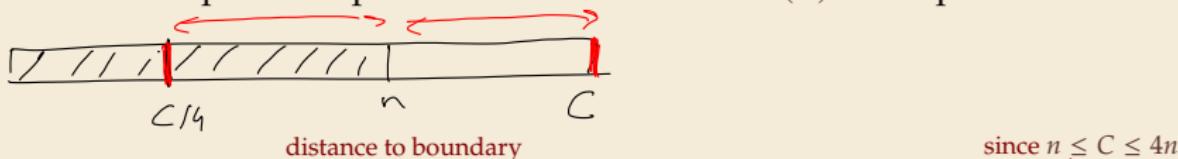
5      -4

for representing arrays

$$\sum_{i=1}^m c_i \leq 5 \cdot m + 9 \cdot \Phi_m \leq \underline{5m + 24 \cdot n}$$

# 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"  $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

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

$\rightsquigarrow$  sequence of  $m$  operations: total actual cost  $\leq$  total amortized cost + final credits

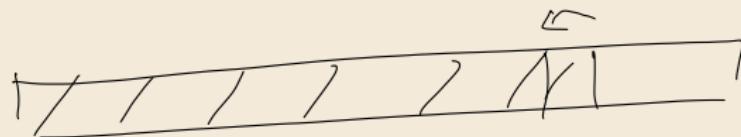
$$\text{here: } \leq \quad 5m \quad + \quad 4 \cdot 0.6n \quad = \Theta(m + n)$$

gushes push / pop



$$\Delta \Phi = -1$$

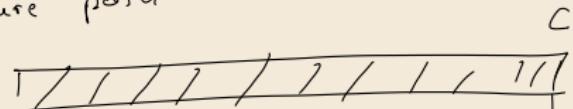
$$a_2 = \frac{1}{2} - 4 \cdot \Delta \Phi = 5$$



$$\Delta \overline{\oplus} = 1$$

$$a_1 = 1 - 4 \cdot 1 = -3 \leq 5$$

fence push



værlur

$$\Phi_{\tilde{c}-1} = \emptyset$$



$$\approx \frac{n}{G}$$

$$n' = n + 1$$

$$\phi_{\mu} =$$

2

watch

## 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  $O(n + m)$ . ✓
- 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  $O(\sqrt{n})$  blocks of  $O(\sqrt{n})$  elements throughout and fast calculation of address for an index.

imagine “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

bisher,  $\Theta(n)$

/

notin 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** 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** 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~~<sup>smallest</sup> priority. (Does not modify the PQ.)

► `delMin()`

Remove the item with ~~largest~~<sup>smallest</sup> 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

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

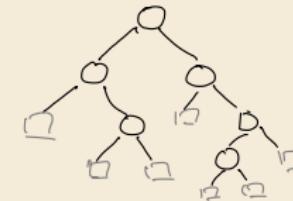
# PQ implementations

## Elementary implementations

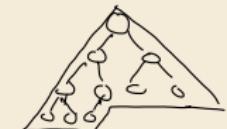
- unordered list  $\rightsquigarrow \Theta(1)$  insert, but  $\Theta(n)$  delMax
- sorted list  $\rightsquigarrow \Theta(1)$  delMax, but  $\Theta(n)$  insert

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

Yes! *Binary heaps*.



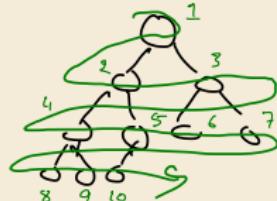
binary tree  
w/ null pointers



complete binary tree

### Array view

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



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

### Tree view

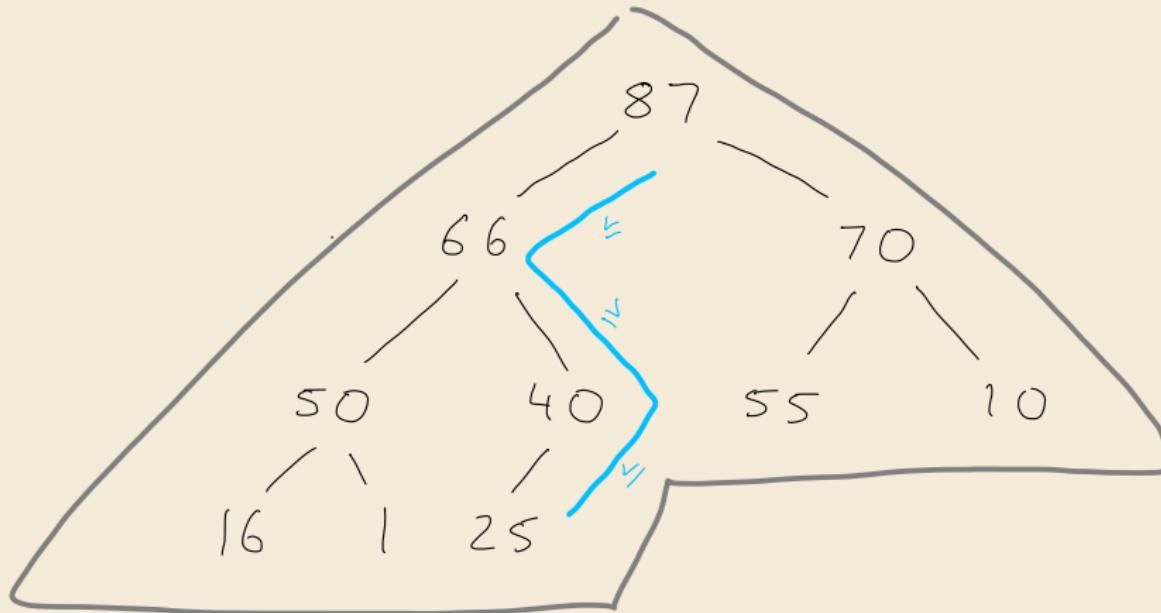
Heap = tree that is  
(i) a complete binary tree  
(ii) heap ordered

all but last level full  
last level flush left

parent  $\geq$  children

$$\text{parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor$$

## Binary heap example



# Why heap-shaped trees?

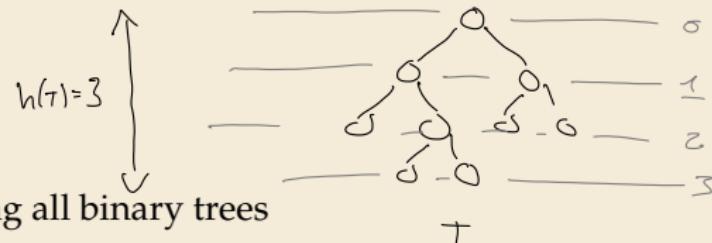
## Why complete binary tree shape?

- ▶ 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.~~

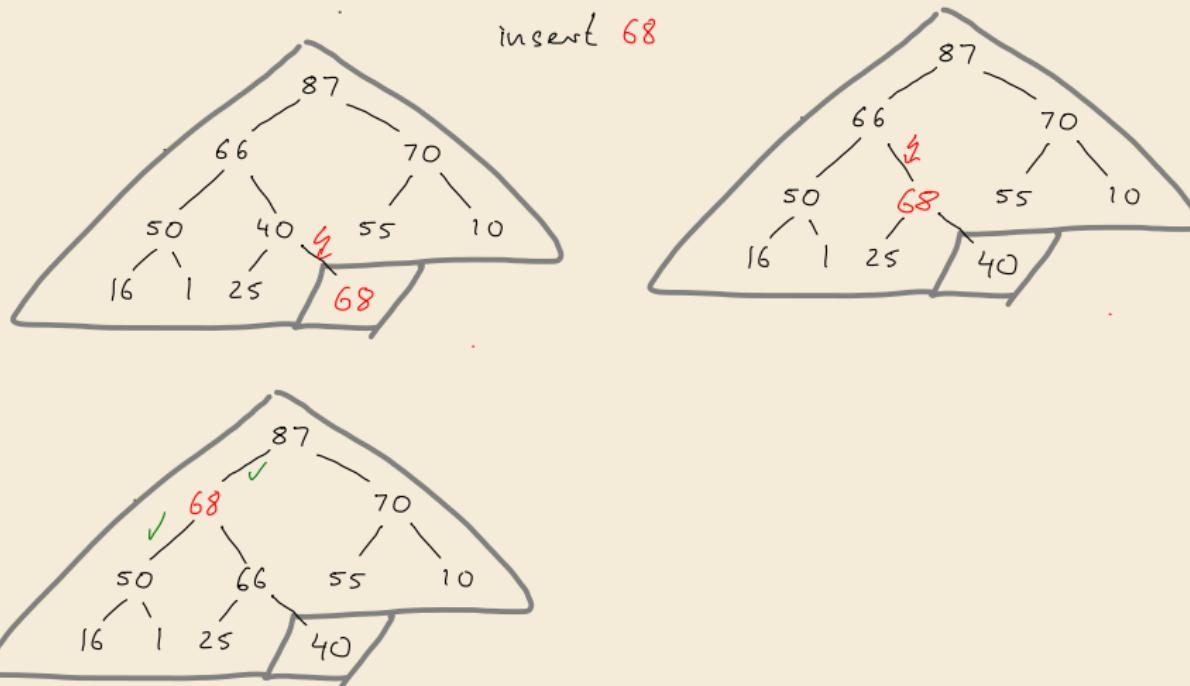


→ *sli.do/cs566*

## 3.4 Operations on Binary Heaps

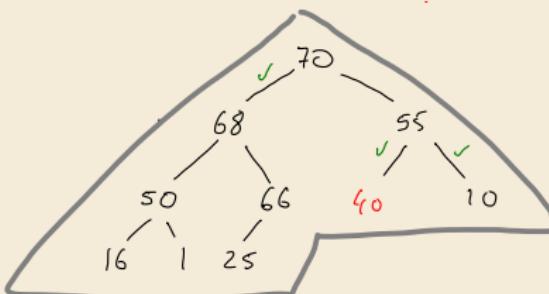
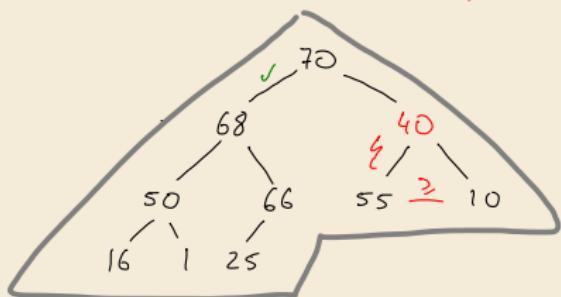
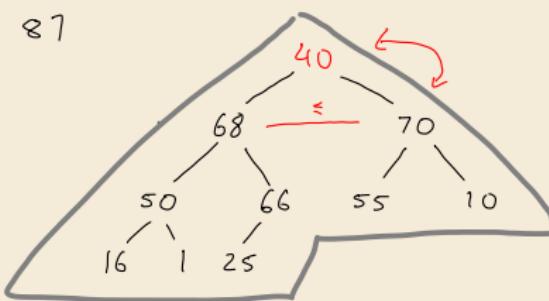
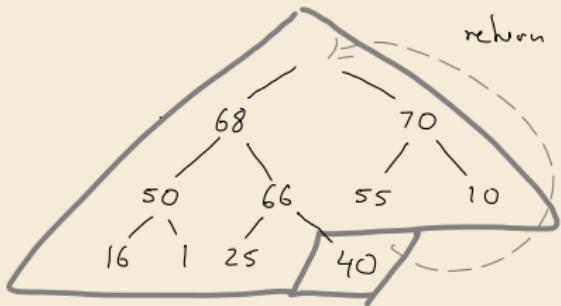
# Insert

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.



Running time:

both dominated  
by one root-to-leaf  
path

~ cost proportional  
to height of tree

---

height  $h \Rightarrow \# \text{nodes}$

$$\geq \sum_{l=0}^{h-1} 2^l + 1$$

$$l=0$$

$$= 2^h$$

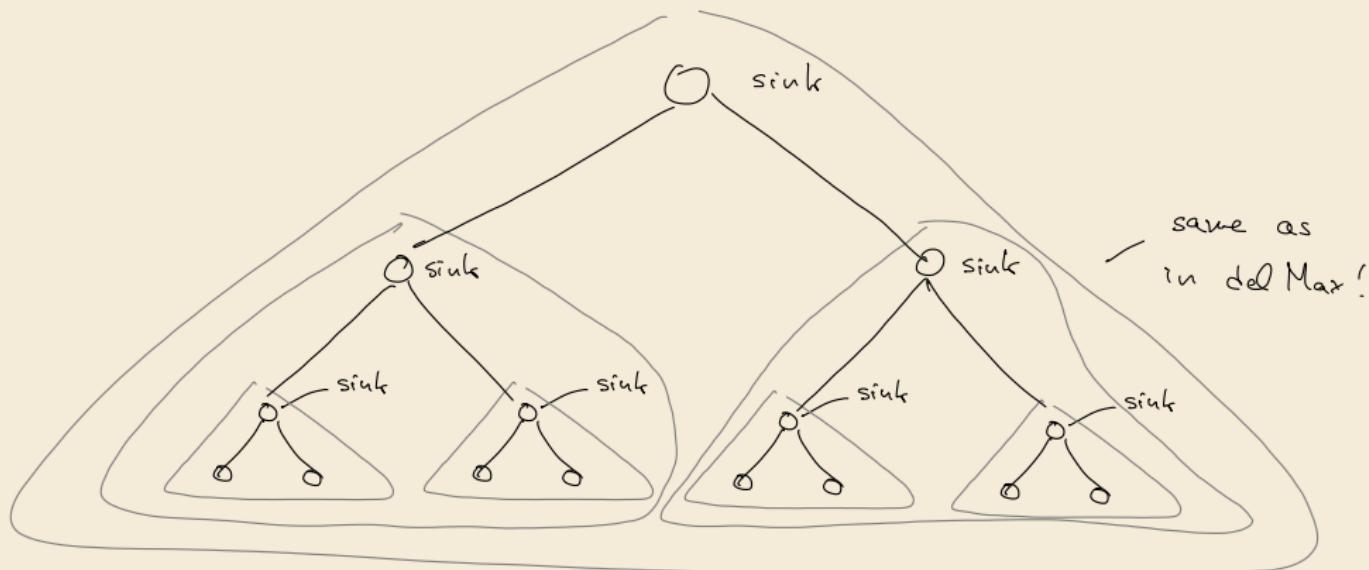
$$\Rightarrow h \leq \log_2(n)$$

# 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!

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

# Binary heap summary

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

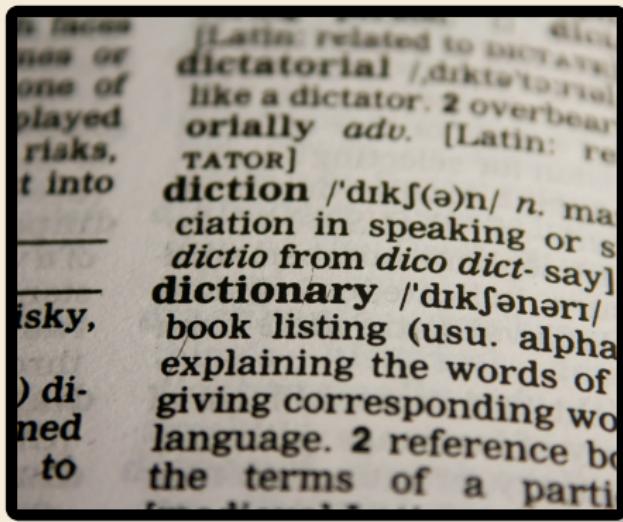
## 3.5 Symbol Tables

# 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:

  $\Theta(n)$  time for put

  $\Theta(n)$  time for get

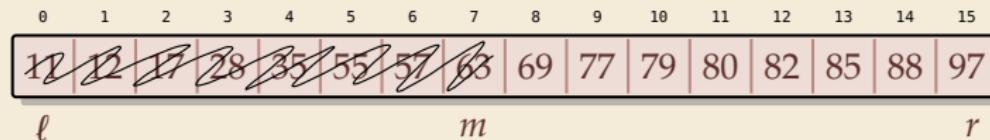
~~ Too slow to be useful

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

# Binary search

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

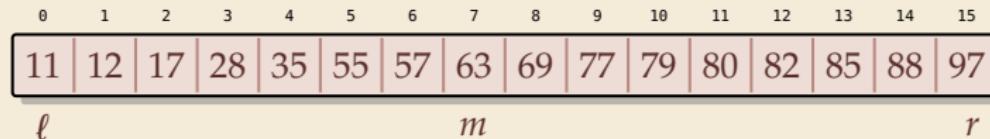
**Example:** search for 69



# Binary search

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

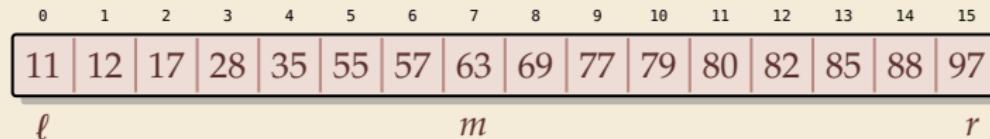
**Example:** search for 69



# Binary search

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

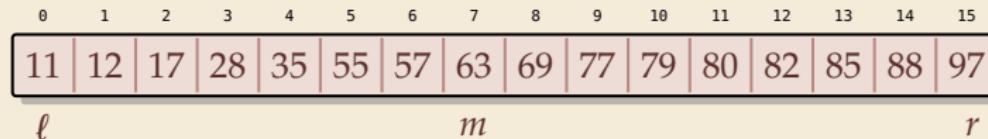
**Example:** search for 69



# Binary search

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

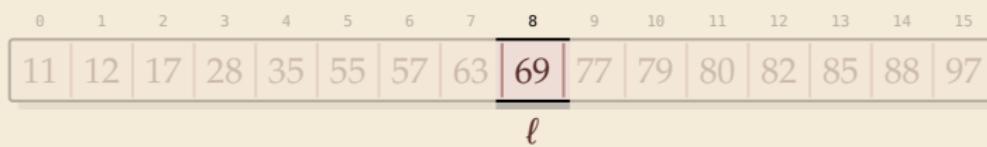
**Example:** search for 69



# Binary search

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

**Example:** search for 69



**Binary search:**

- ▶ halve remaining list in each step

$\rightsquigarrow \leq \lfloor \log_2 n \rfloor + 1$  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** 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*

# 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.~~



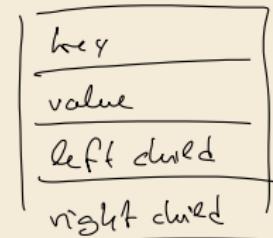
→ *sli.do/cs566*

# Binary search trees

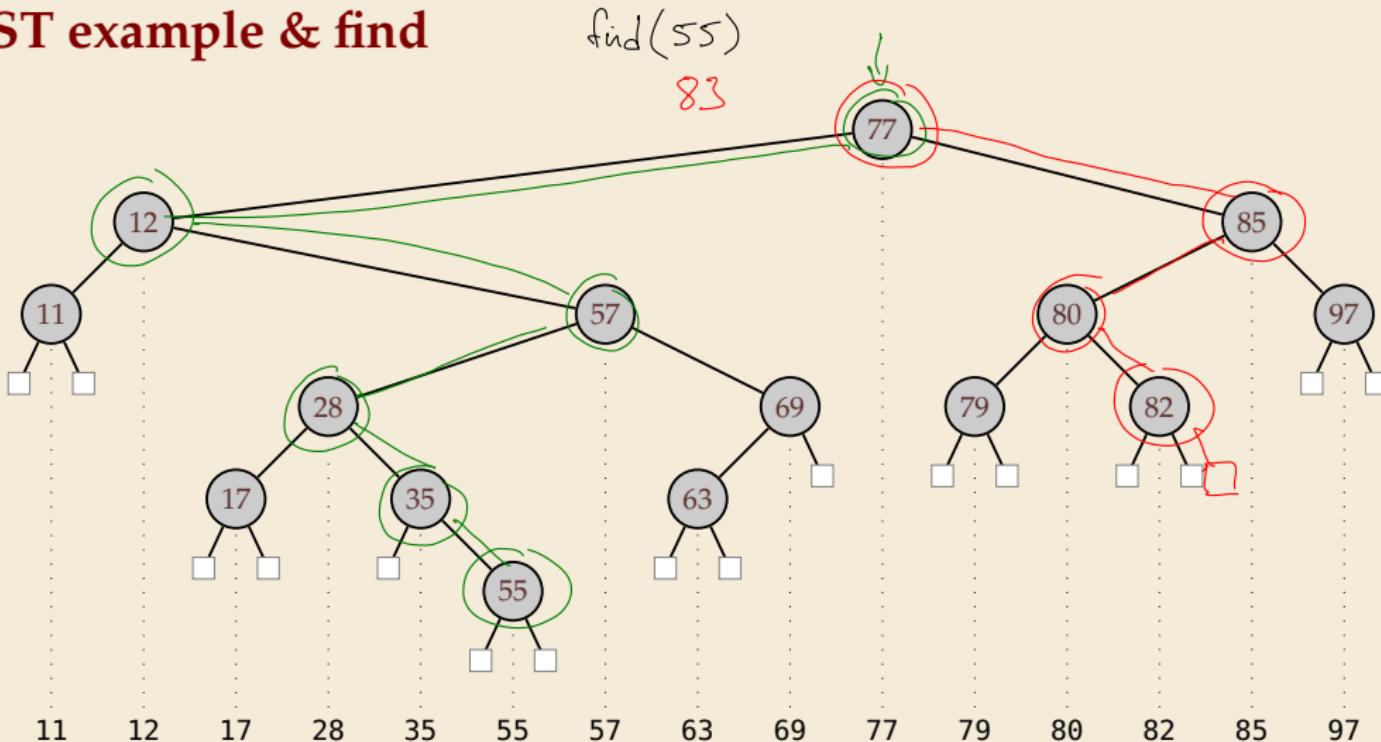
Binary search trees (BSTs)  $\approx$  dynamic sorted array

- ▶ 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

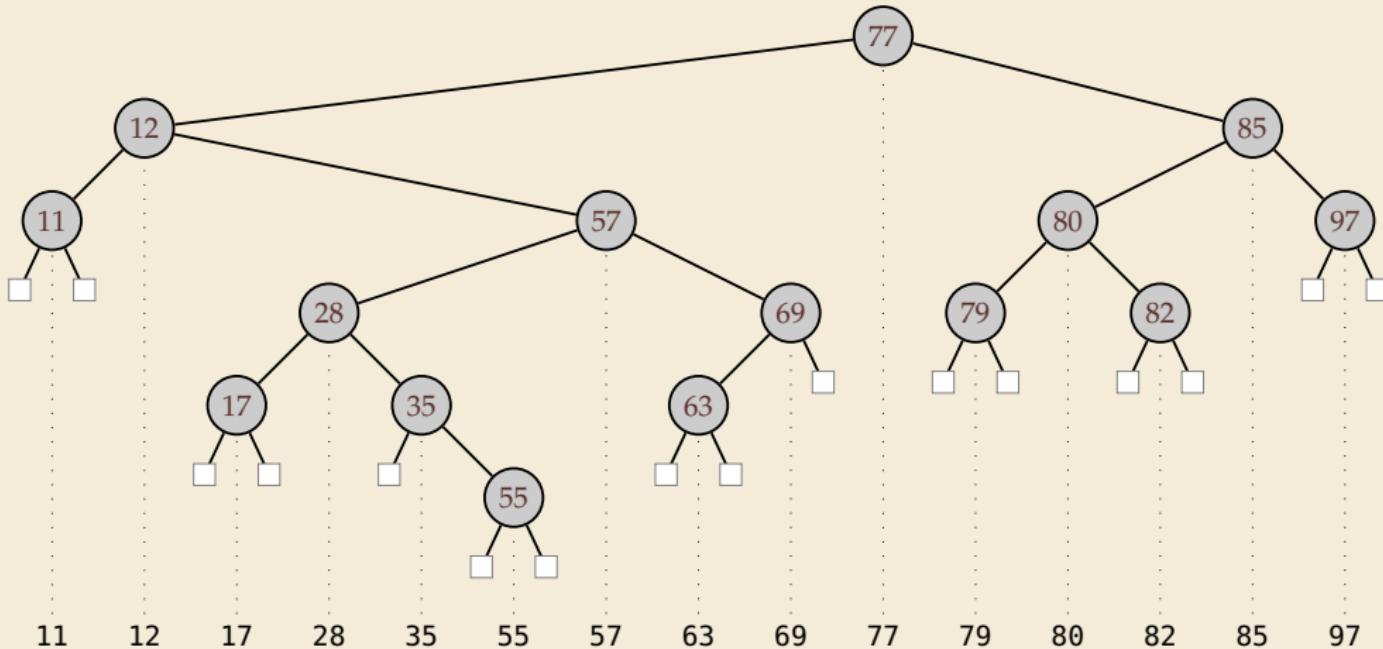


## BST example & find



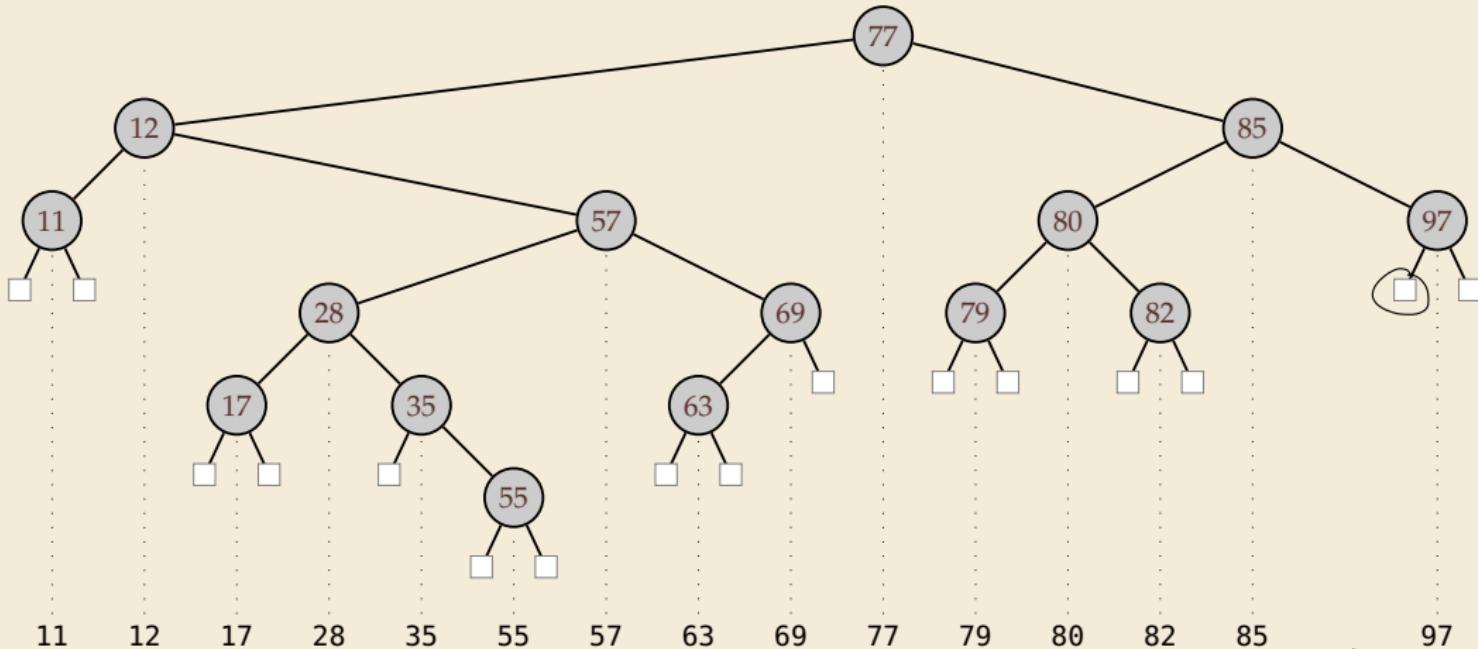
# BST insert

Example: Insert 88



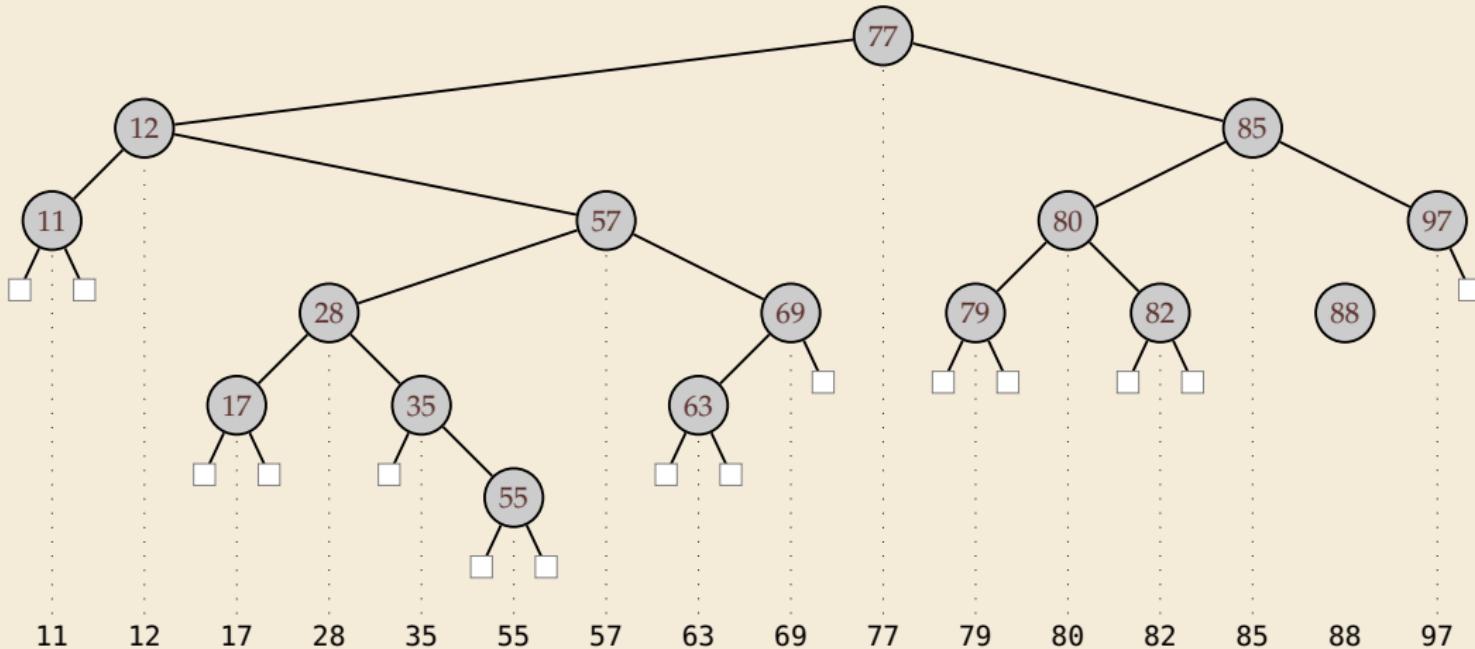
# BST insert

Example: Insert 88



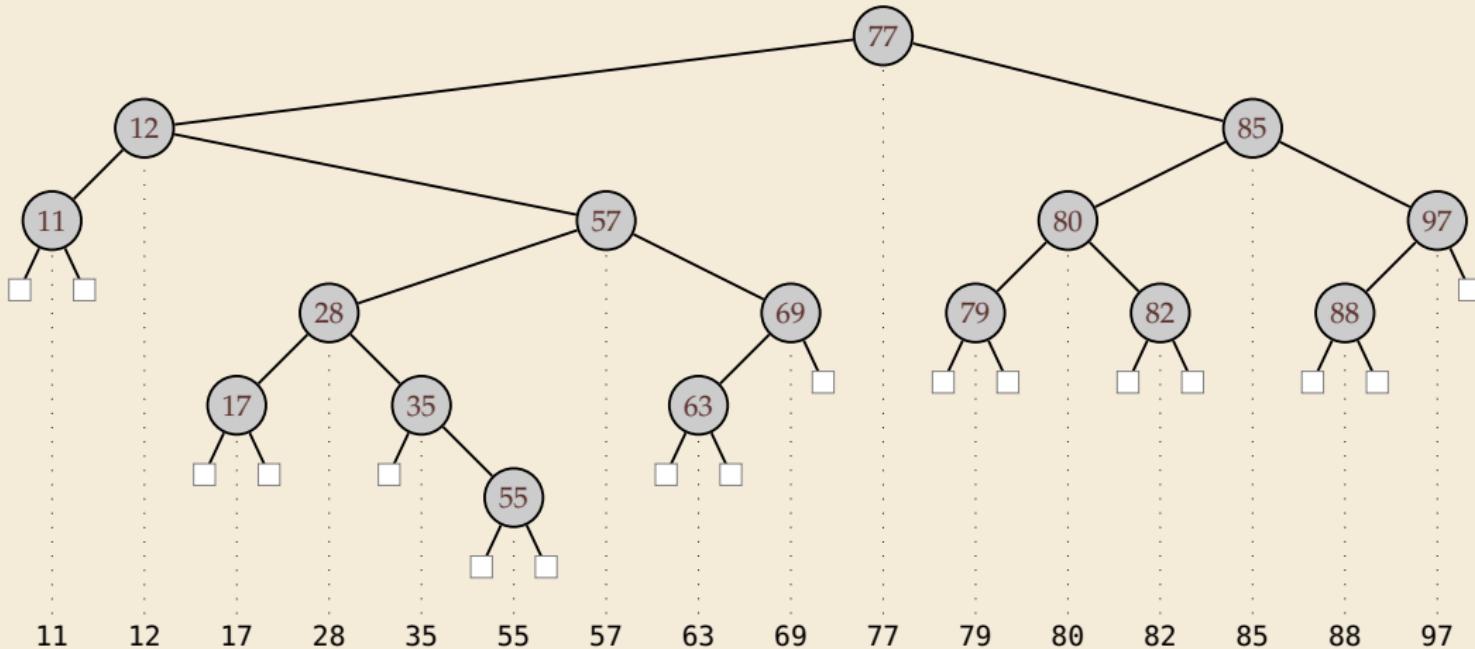
# BST insert

Example: Insert 88



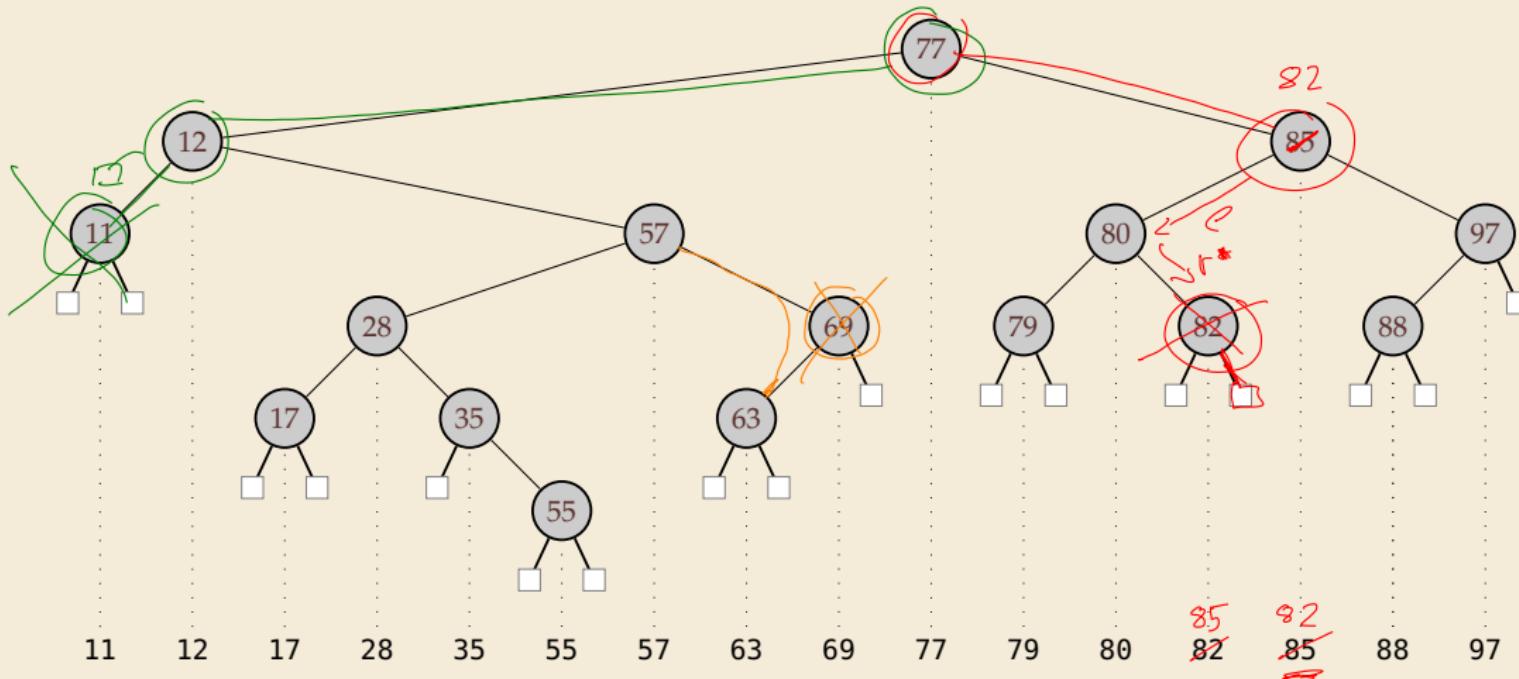
# BST insert

Example: Insert 88



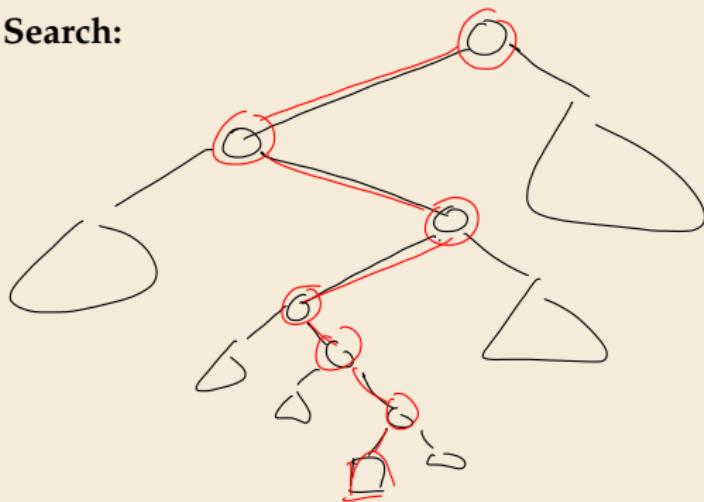
# BST delete

- 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:



#cups = length of  
search path  
+ 1

$\leq \underbrace{\text{height of BST}}_h + 1$

- ▶ Insert:  $O(h)$

- ▶ Delete: all cases  $O(h)$

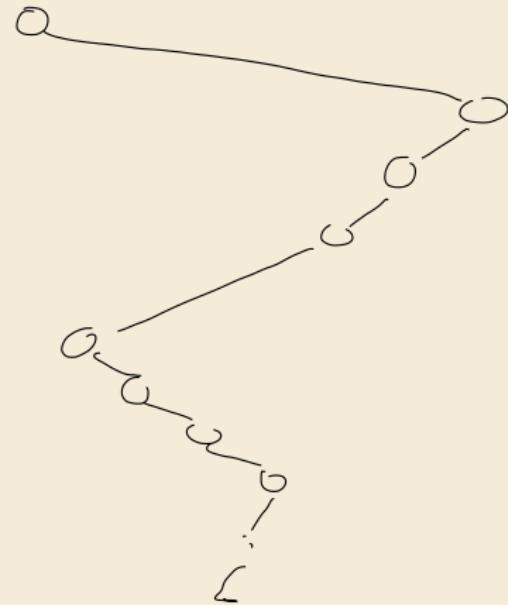
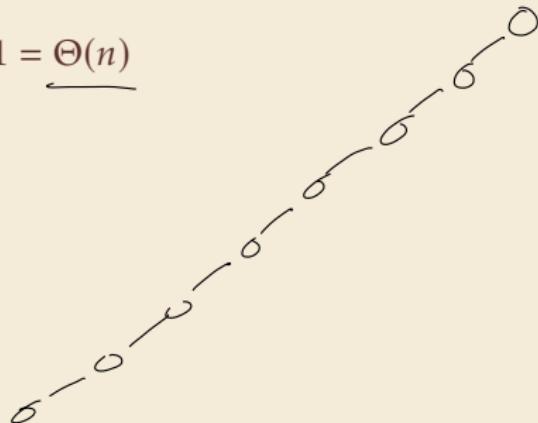
## BST summary

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

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

Precise analysis for

EXPECTED

$C_n$  = TOTAL cost of finding ALL keys in BST

$[n]$  inserted in random order

"

$[1..n] = \{1, \dots, n\}$

$$C_n = n + \frac{1}{n} \sum_{r=1}^n (C_{r-1} + C_{n-r})$$

comp w/ root

Average Case:

$\blacktriangleright$  Assumption: insertions come in random order  
no deletions

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

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

}  $\notin$  exam

root key = first inserted key

