

2

Fundamental Data Structures

17 February 2021

Sebastian Wild

Outline

2 Fundamental Data Structures

- 2.1 Stacks & Queues
- 2.2 Resizable Arrays
- 2.3 Priority Queues
- 2.4 Binary Search Trees
- 2.5 Ordered Symbol Tables
- 2.6 Balanced BSTs

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 operations
 - ▶ **what** should happen
 - ▶ **not:** how to do it
 - ▶ **not:** how to store data
- ≈ Java interface
(with Javadoc 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/comp526

Click on “Polls” tab

Clicker Question

Which of the following are examples of abstract data types?

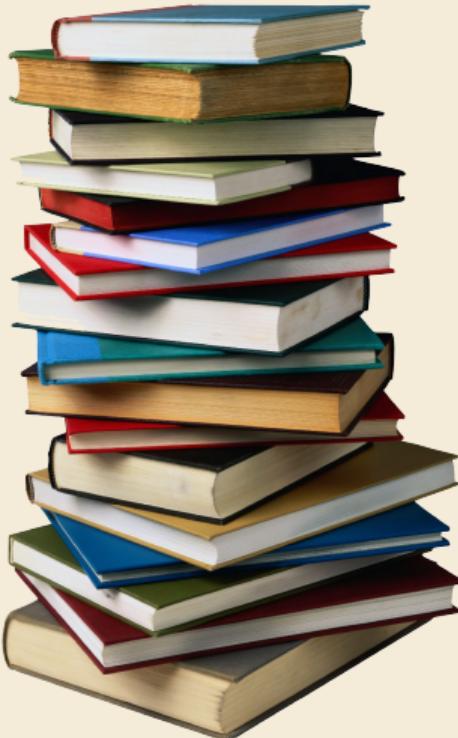


- A ADT
- B Stack ✓
- C Deque ✓
- D Linked list
- E binary search tree
- F Queue ✓
- G resizable array
- H heap
- I priority queue ✓
- J dictionary/symbol table ✓
- K hash table

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

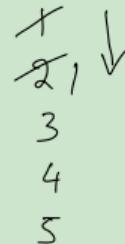
Suppose a stack initially contains the numbers $\overbrace{1, 2, 3, 4, 5}$ with 1 at the top.

What is the content of the stack after the following operations:

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



- A** $\overbrace{1, 2, 3, 1}$
- 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

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~~

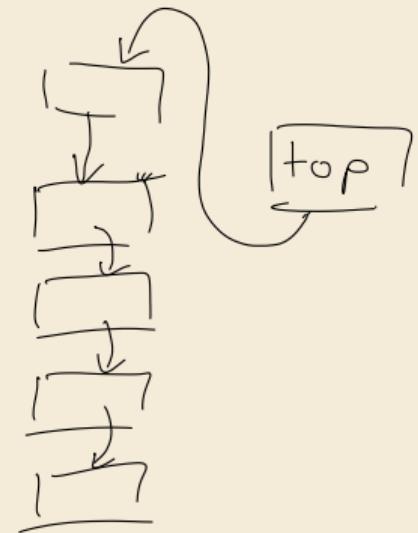
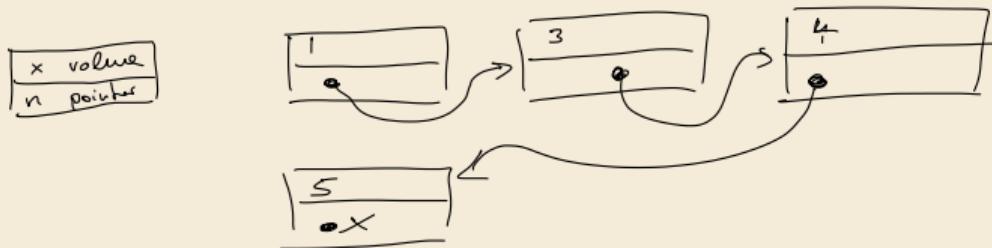
sli.do/comp526

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)

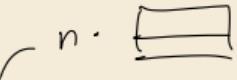


Linked-list implementation for Stack

Invariants:

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

Linked stacks:



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

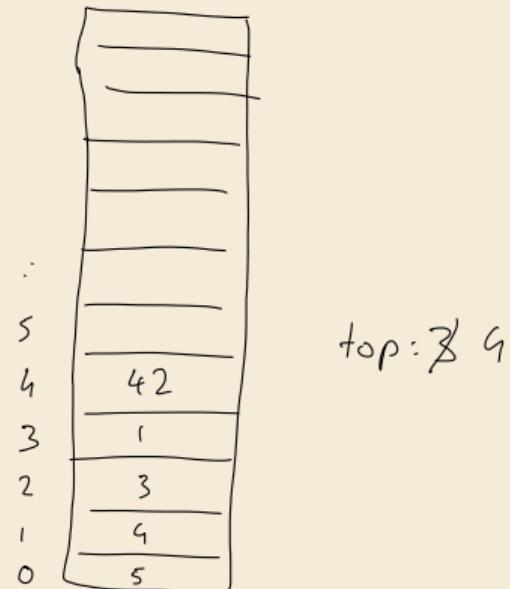
Array-based implementation for Stack

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 .



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 $0, 1, \dots, n - 1$

- ▶ `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  $0, 1, \dots, n - 1$
  - ▶ `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

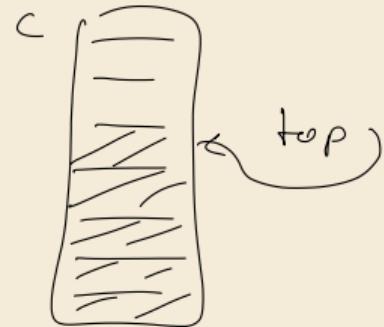
*Can we have unbounded stacks based on arrays?*      Yes!

# Doubling trick

Can we have unbounded stacks based on arrays? Yes!

## Invariants:

- ▶ maintain array  $S$  of elements, from bottommost to topmost
- ▶ maintain index  $\text{top}$  of position of topmost element in  $S$
- ▶ maintain capacity  $C = S.\text{length}$  so that  $\frac{1}{4}C \leq n \leq C$
- ~~ can always push more elements!



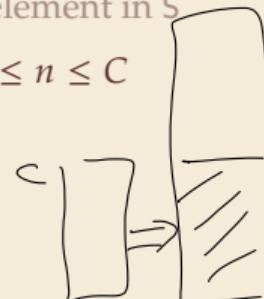
# Doubling trick

Can we have unbounded stacks based on arrays? Yes!

## Invariants:

- ▶ maintain array  $S$  of elements, from bottommost to topmost
- ▶ maintain index  $\text{top}$  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?

- 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](https://sli.do/comp526)

Click on “Polls” tab

## Amortized Analysis

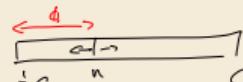
- ▶ Any individual operation push / pop can be expensive!  
 $\Theta(n)$  time to copy all elements to new array.
- ▶ **But:** An one expensive operation of cost  $T$  means  $\Omega(T)$  next operations are cheap!

# 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 · 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

$$a_i = c_i - 4(\Phi_i - \Phi_{i-1}) \leq 5 \quad \text{here: } \leq \quad 5m \quad + \quad 4 \cdot 0.6n = \underline{\Theta(m+n)}$$

$$\sum_{i=1}^m a_i \leq 5m \geq \sum_{i=1}^m a_i = \sum_{i=1}^m c_i - 4 \sum_{i=1}^m (\Phi_i - \Phi_{i-1}) = \sum_{i=1}^m c_i - 4(\Phi_m - \Phi_0)$$

$$\sum_{i=1}^m c_i \leq 5m + 4\Phi_m - 4\Phi_0 \leq 5m + 4\Phi_7$$

## 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](https://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](http://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** A 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 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.)

- ▶ ~~`delMax()`~~

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.



# 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](https://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

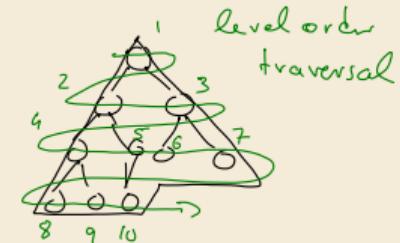
*Can we get something between these extremes? Like a “slightly sorted” list?*

# 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

prevent  
father  $\geq$  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  $A$

- ▶ 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  $\rightsquigarrow$  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  $A$

- ▶ parent at  $\lfloor k/2 \rfloor$
- ▶ 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



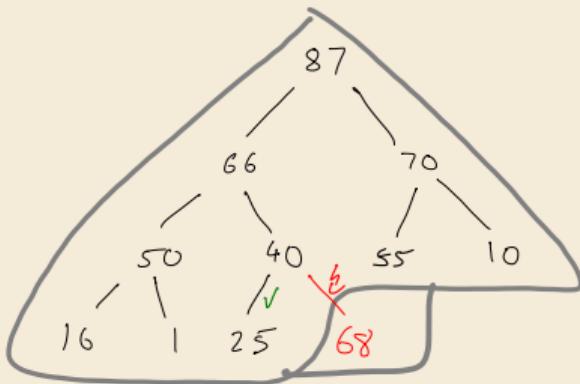
What is a heap-ordered tree?

- 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 bigger than the key at the root. ✓
- D ~~An tree that is stored in the heap area of the memory.~~

[sli.do/comp526](http://sli.do/comp526)

Click on “Polls” tab

# Insert

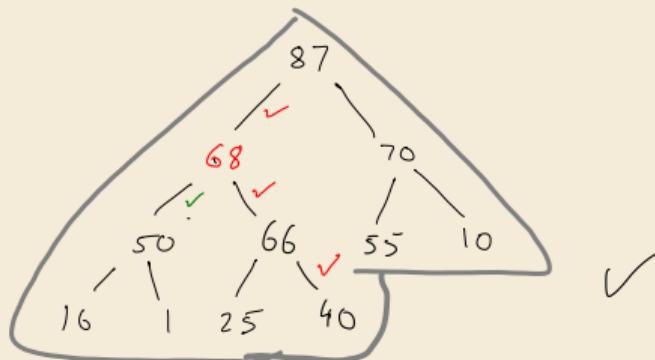
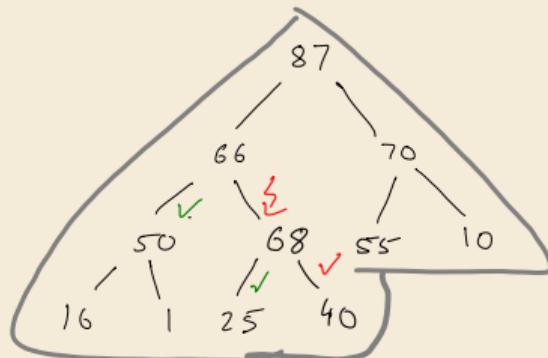


68

shape : only 1 possible position

BUT heap-order ↴

⇒ swim up the heap



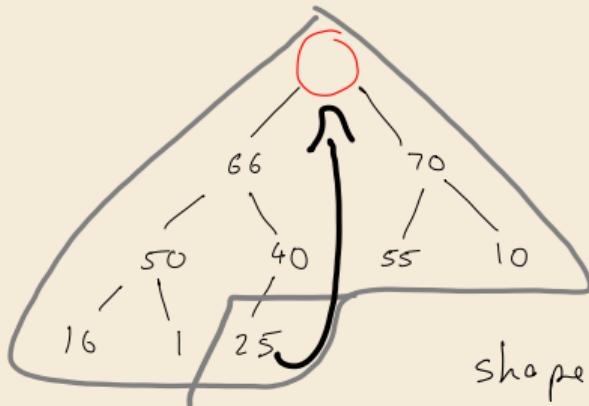
## Delete Max

87

find max is easy

after removing if

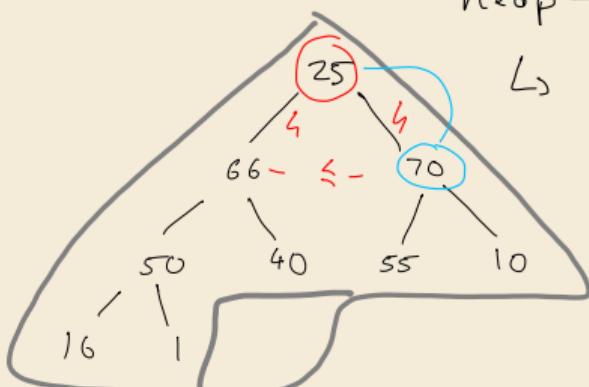
↳ complete binary tree

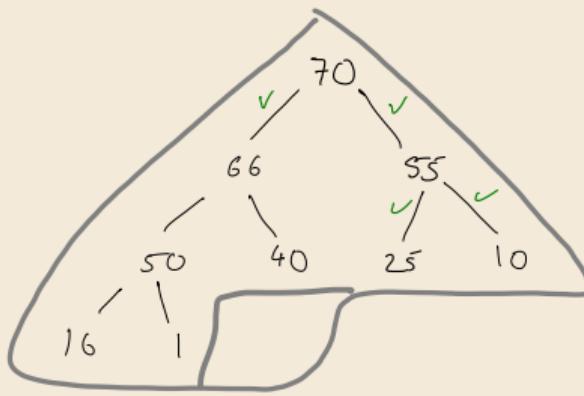
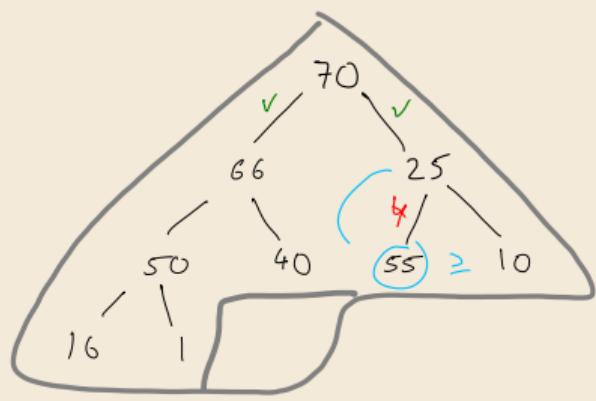


shape ✓

heap-order ↴

↳ let 25 sink in heap



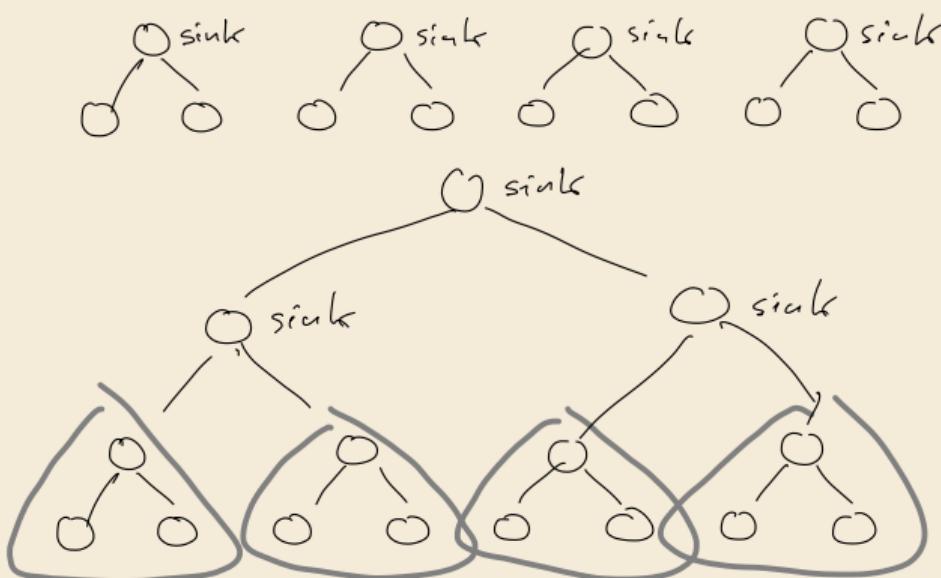


## Heap construction

n. insert

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

can do better:



$\frac{n}{4}$ . sink in heap of size  $\leq 3$

$\frac{n}{8}$ .  $\sim$   $\leq 7$

$\frac{n}{2^k}$   
 $k = \log n$

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

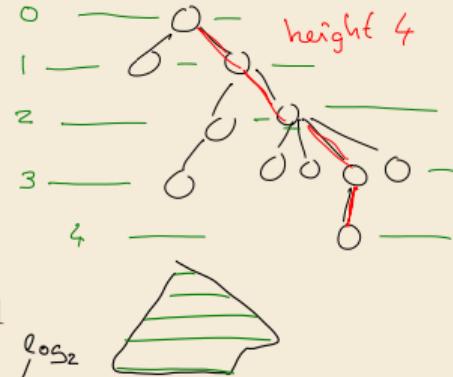
# Analysis

## Height of binary heaps:

- ▶ height of a tree: # edges on longest root-to-leaf path
- ▶ depth/level of a node: # edges from root  $\rightsquigarrow$  root has depth 0

- ▶ How many nodes on first  $k$  full levels? 
$$\sum_{\ell=0}^k 2^\ell = 2^{k+1} - 1$$

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



# 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? 
$$\sum_{\ell=0}^k 2^\ell = 2^{k+1} - 1$$
 $\rightsquigarrow$  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!
$$\begin{aligned} &\leq 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^\ell \cdot (h-\ell) + \cdots + 2^{h-1} \cdot 1 + 2^h \cdot 0 \\ &= \sum_{\ell=0}^h 2^\ell (h-\ell) = \sum_{i=0}^h \frac{2^h}{2^i} i = 2^h \sum_{i=0}^h \frac{i}{2^i} \leq 2 \cdot 2^h \leq 4n \end{aligned}$$

## 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>                          | $O(1)$       |

## 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.~~

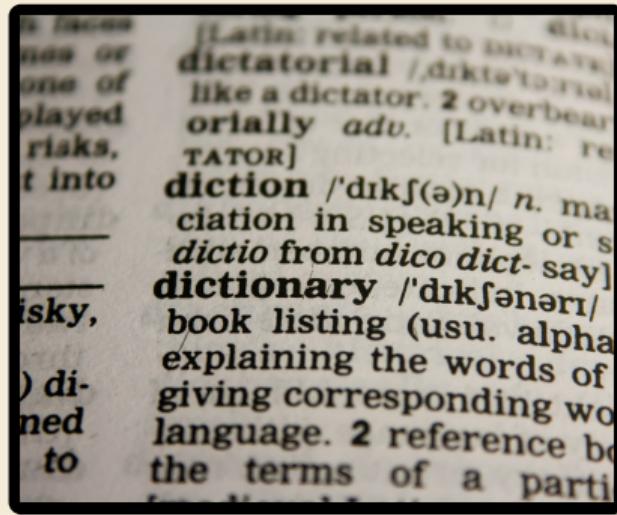
[sli.do/comp526](http://sli.do/comp526)

Click on “Polls” tab

# Symbol table ADT

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

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



- ▶ `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) from 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

~~~ 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

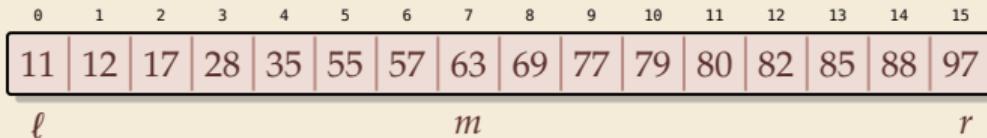
~~ Too slow to be useful

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

Binary search

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

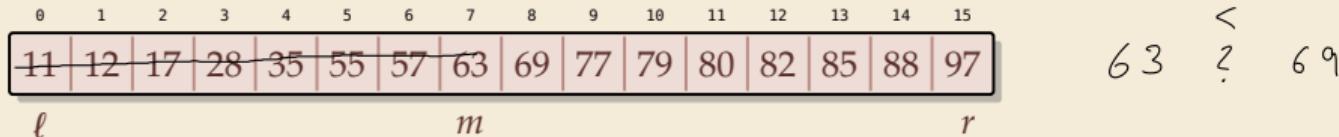
Example: search for 69



Binary search

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

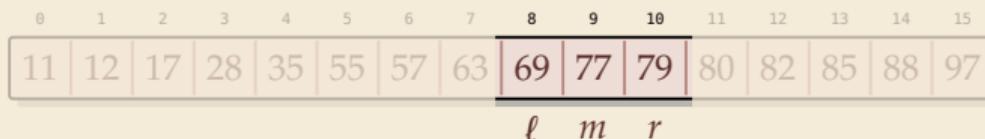
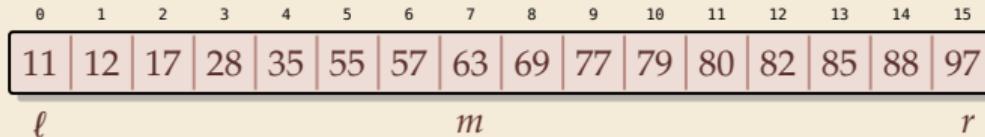
Example: search for 69



Binary search

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

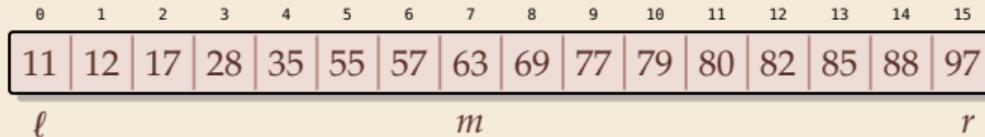
Example: search for 69



Binary search

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

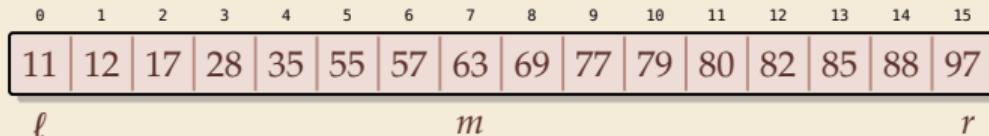
Example: search for 69



Binary search

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

Example: search for 69



Binary search:

- ▶ halve $\nearrow \pm 1$ remaining list in each step

$\rightsquigarrow \leq \lfloor \lg n \rfloor + 1$ 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

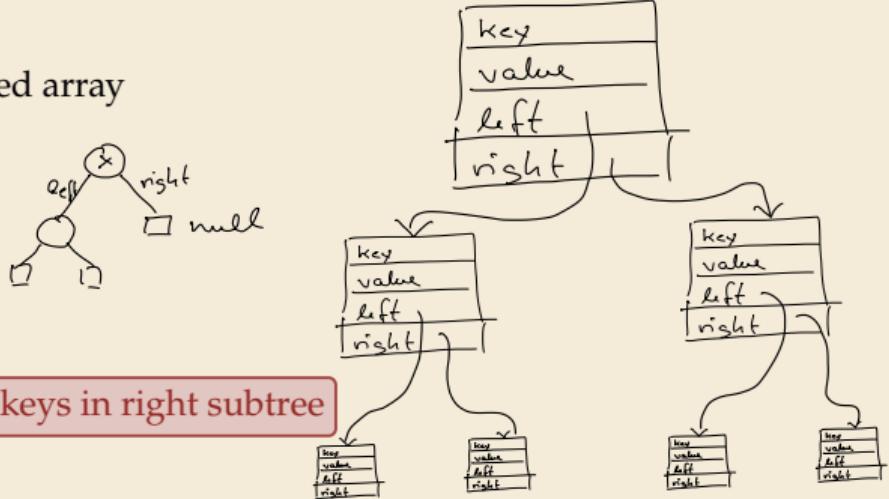
Click on "Polls" tab

Binary search trees

Binary search trees (BSTs) \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

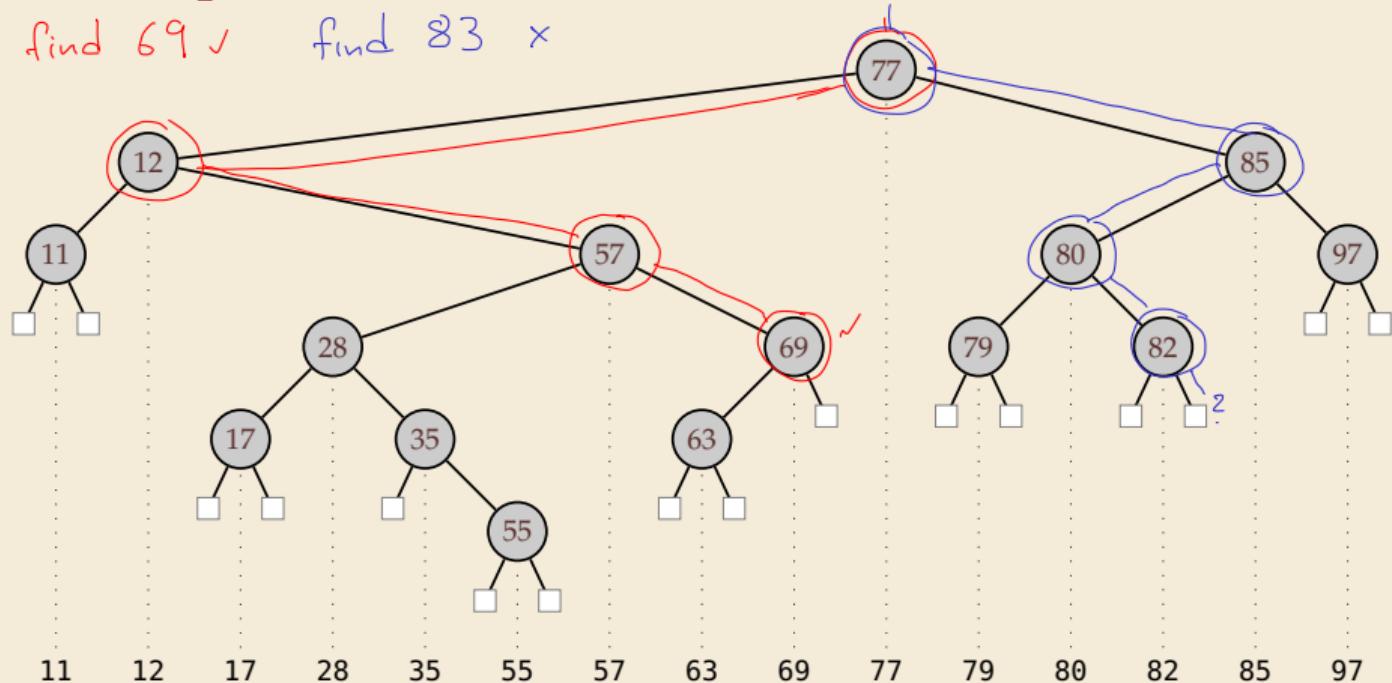


$$\triangle = \square + \text{tree}$$

BST example & find

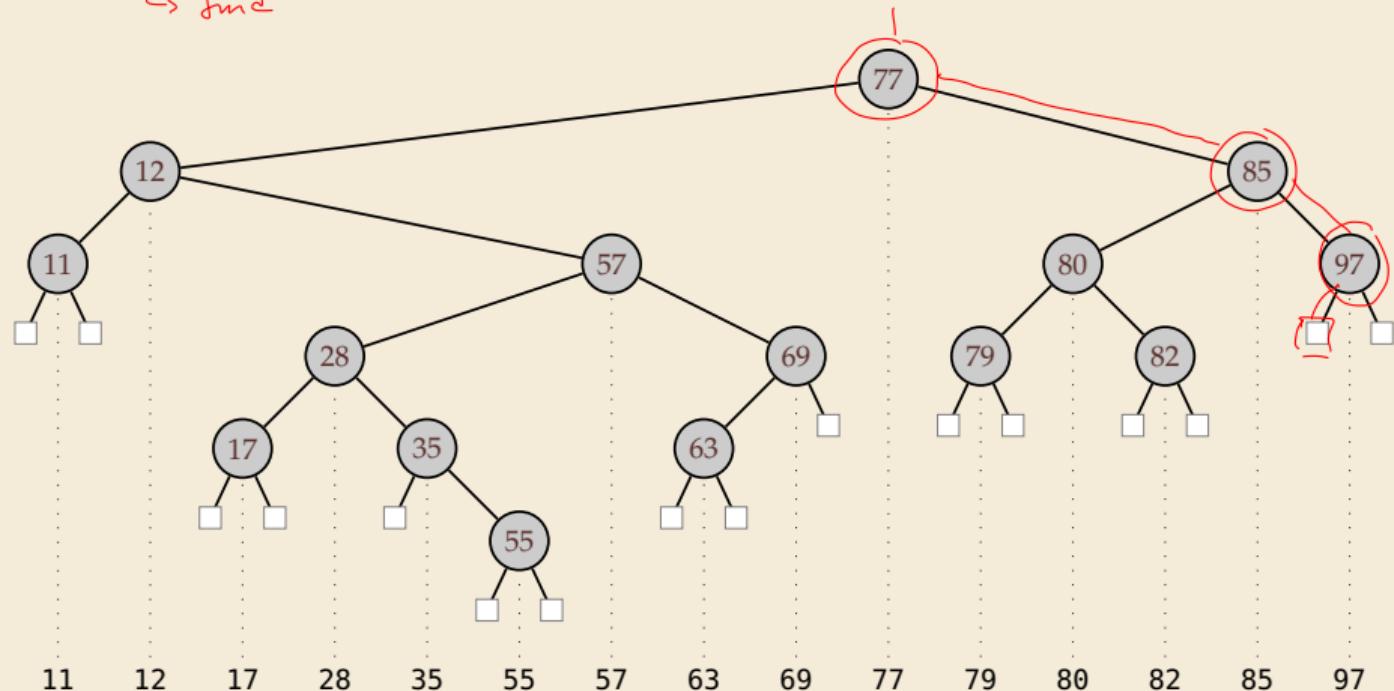
find 69 ✓

find 83 ✗



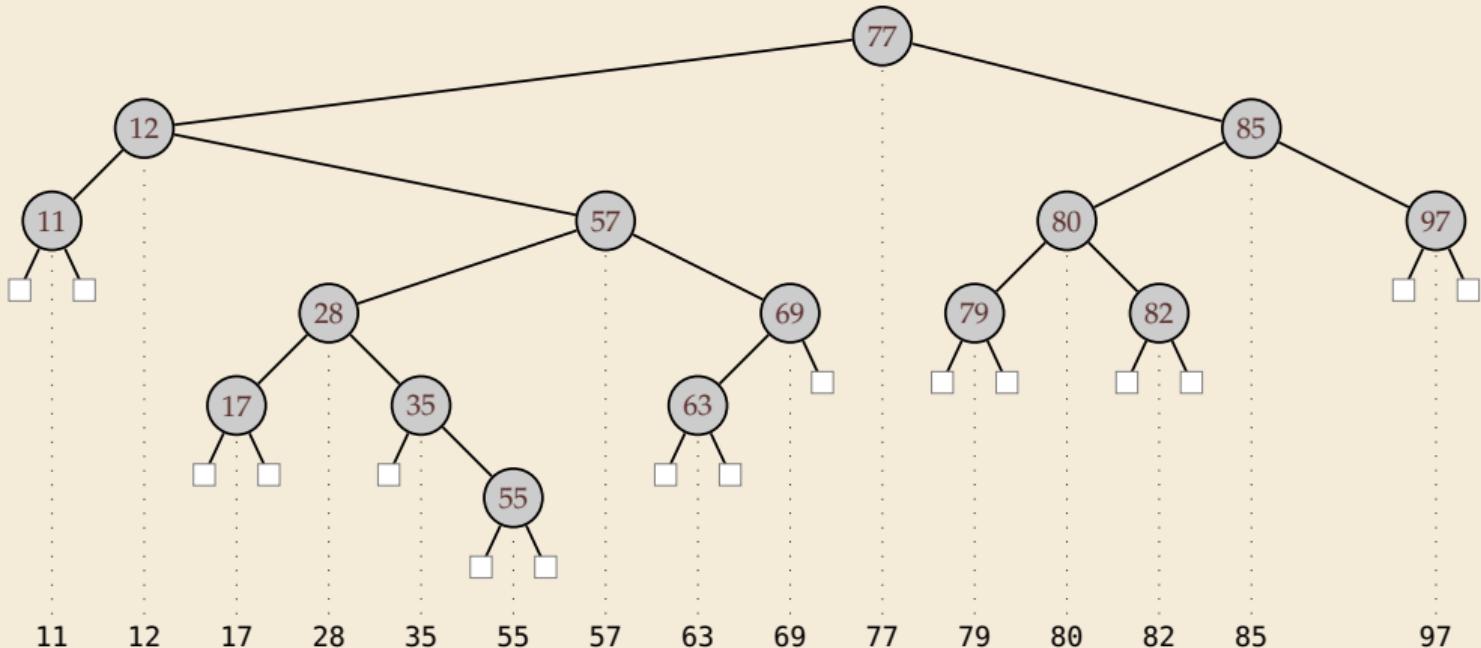
BST insert

Example: Insert 88



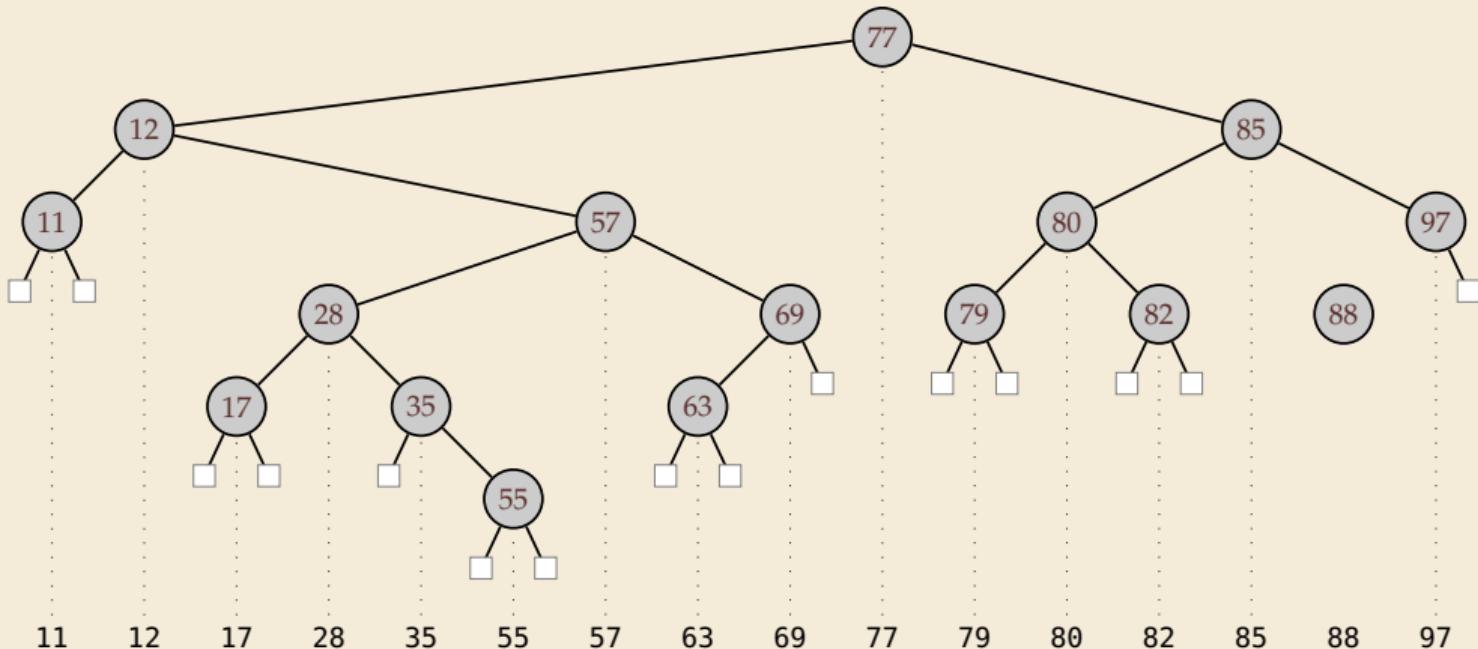
BST insert

Example: Insert 88



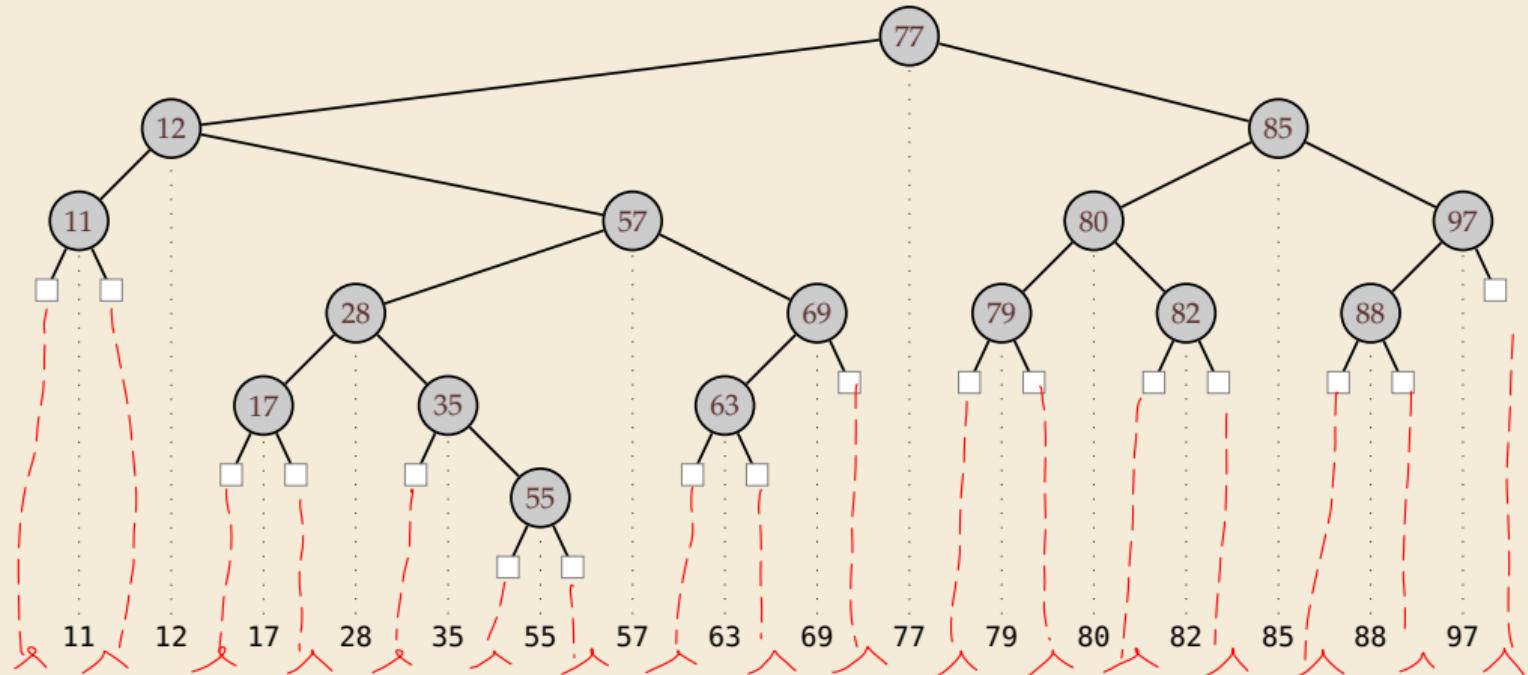
BST insert

Example: Insert 88



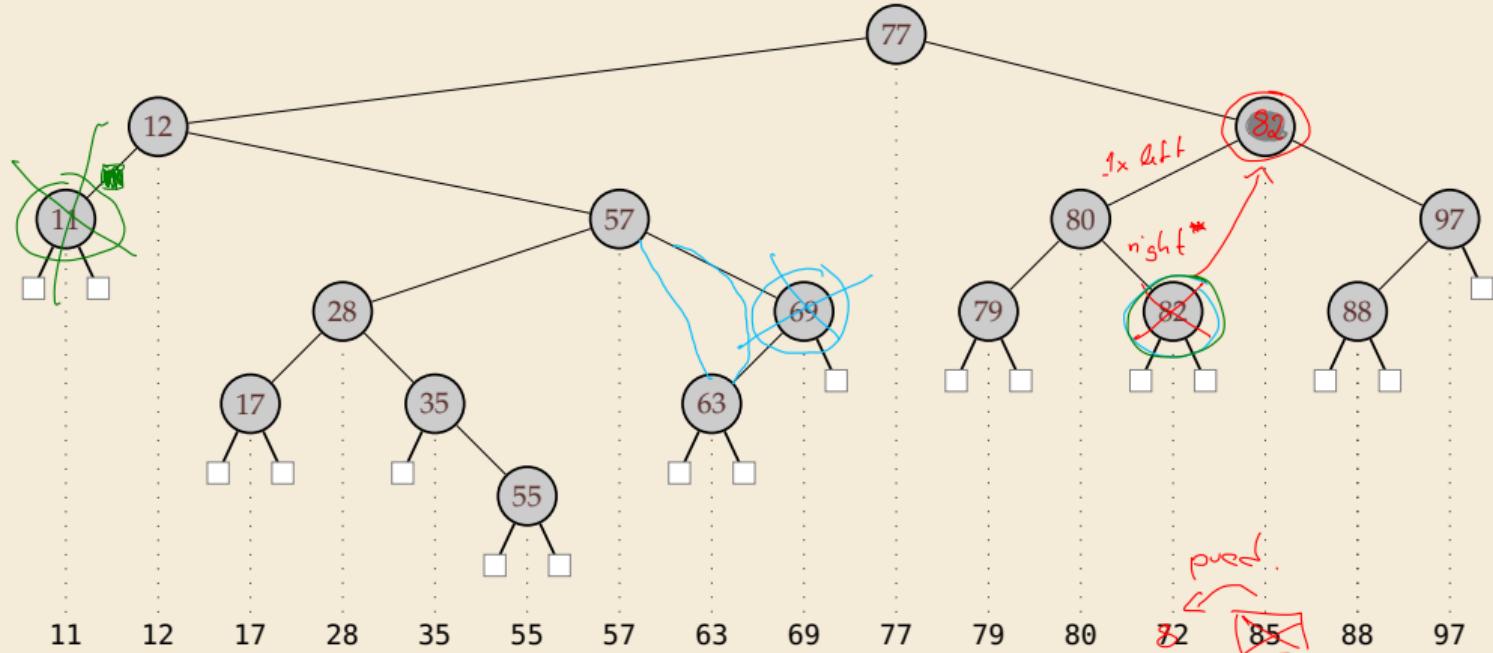
BST insert

Example: Insert 88



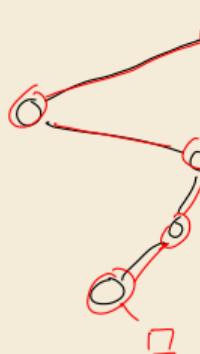
BST delete

- ▶ 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 :



\leq height of BST

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

insert : same as search $O(h)$

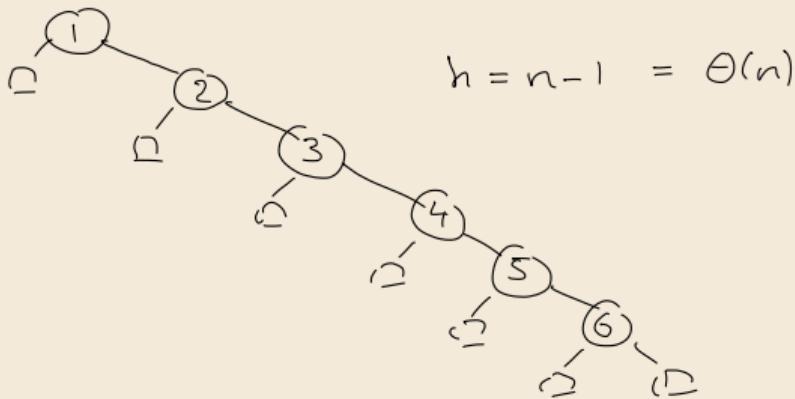
delete : search for k plus its predecessor $O(h)$

BST summary

| Operation | Running Time |
|--|--------------|
| <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?

a) worst case



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

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

- ▶ `ceiling(x)` $\lceil x \rceil$

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

- ▶ `rank(x)`

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

- ▶ `select(i)` i $A[{:}]$

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



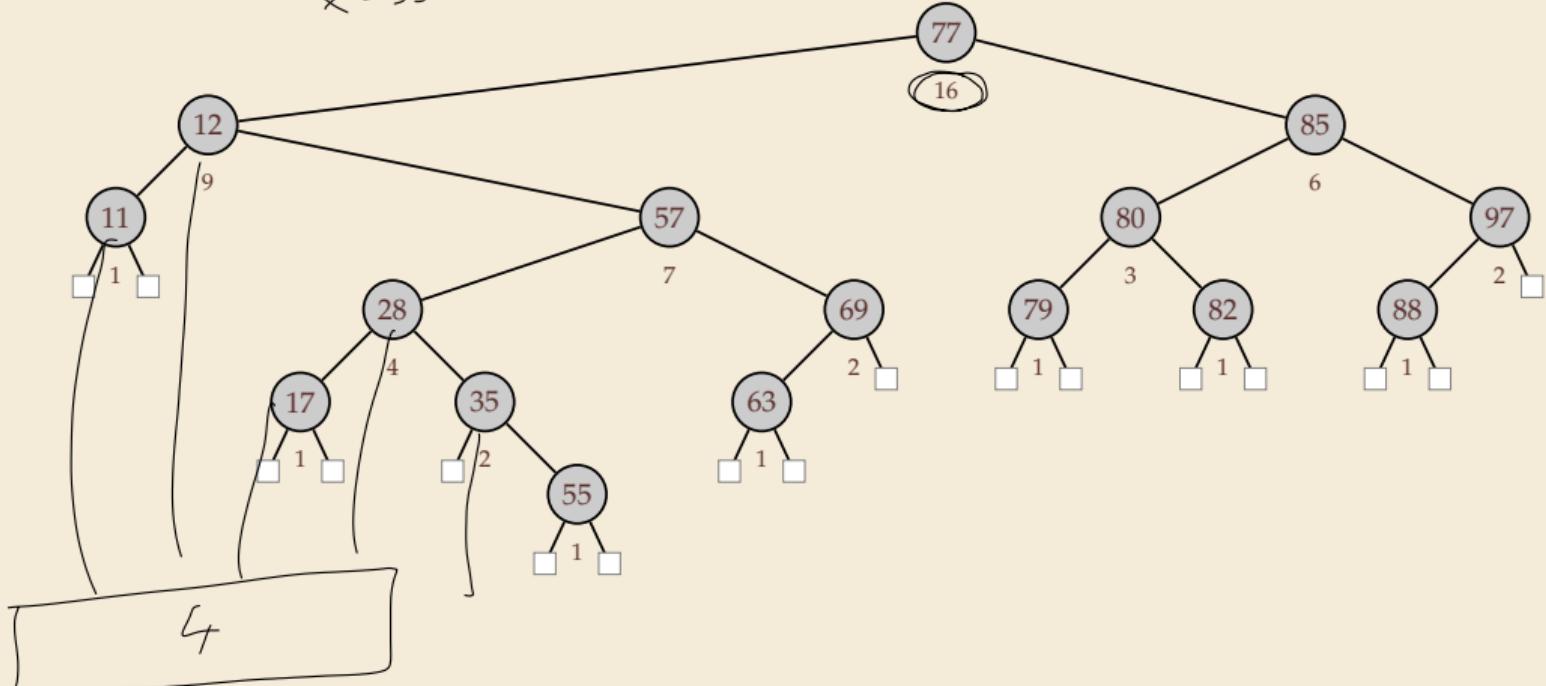
With `select`, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

Augmented BSTs

slide: rank(35)

$x = 35$



Rank

$$\gamma = 0$$

rank(35)

`v += v.left.st + 1`

1

14

9

1

7

28

8

3

9

8

6

•

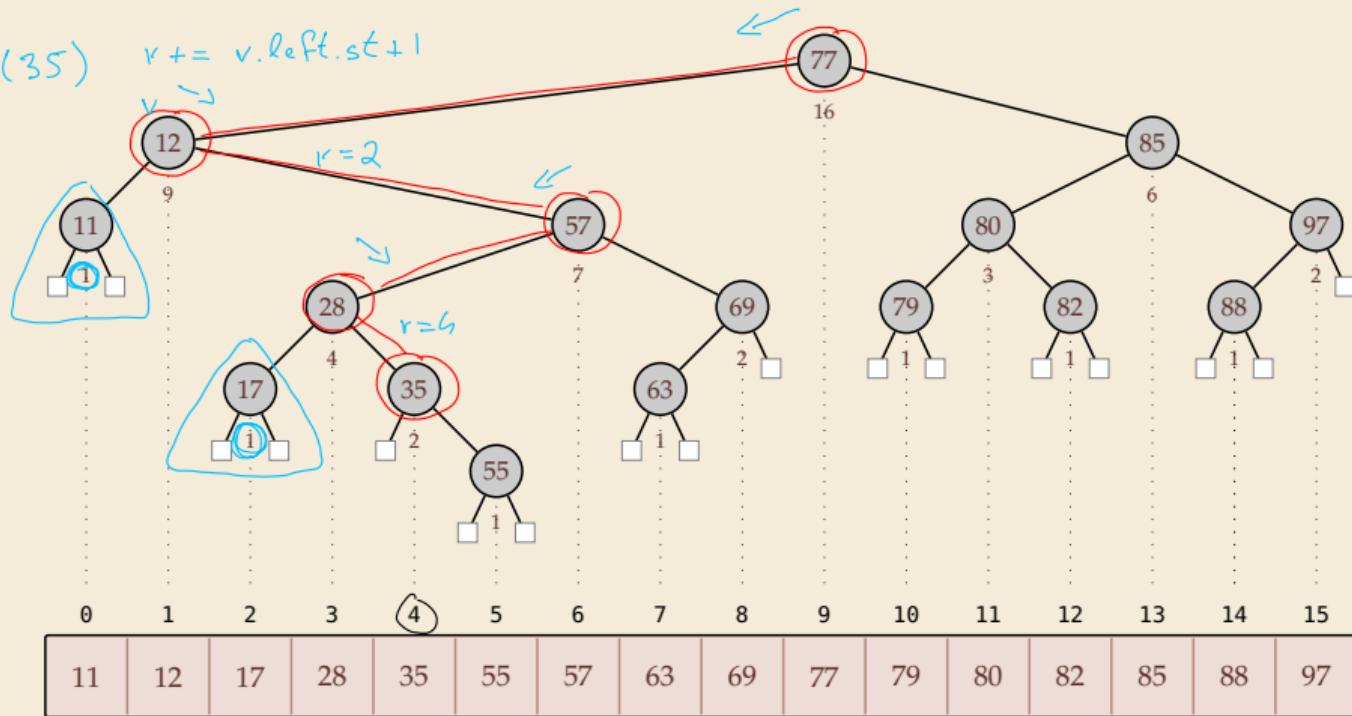
8

8

3

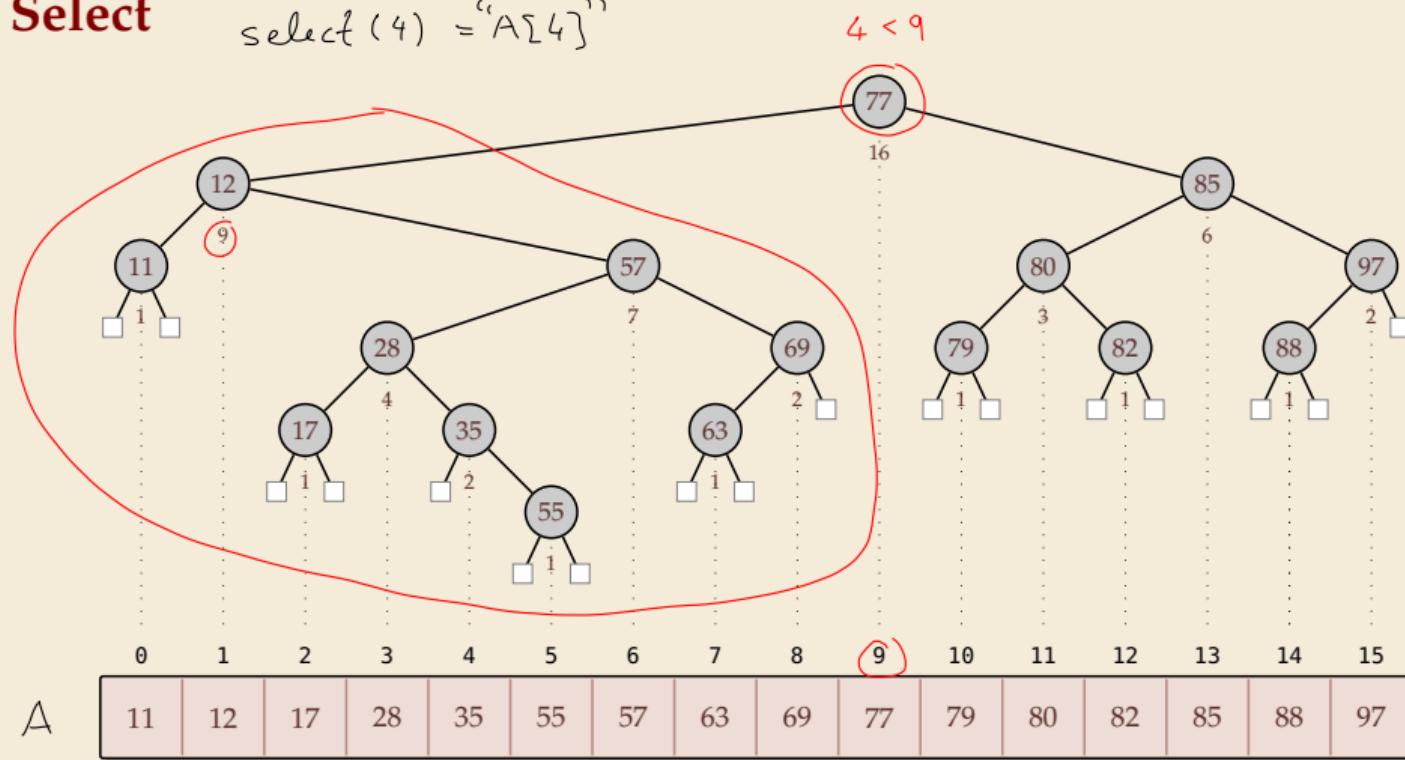
1

12

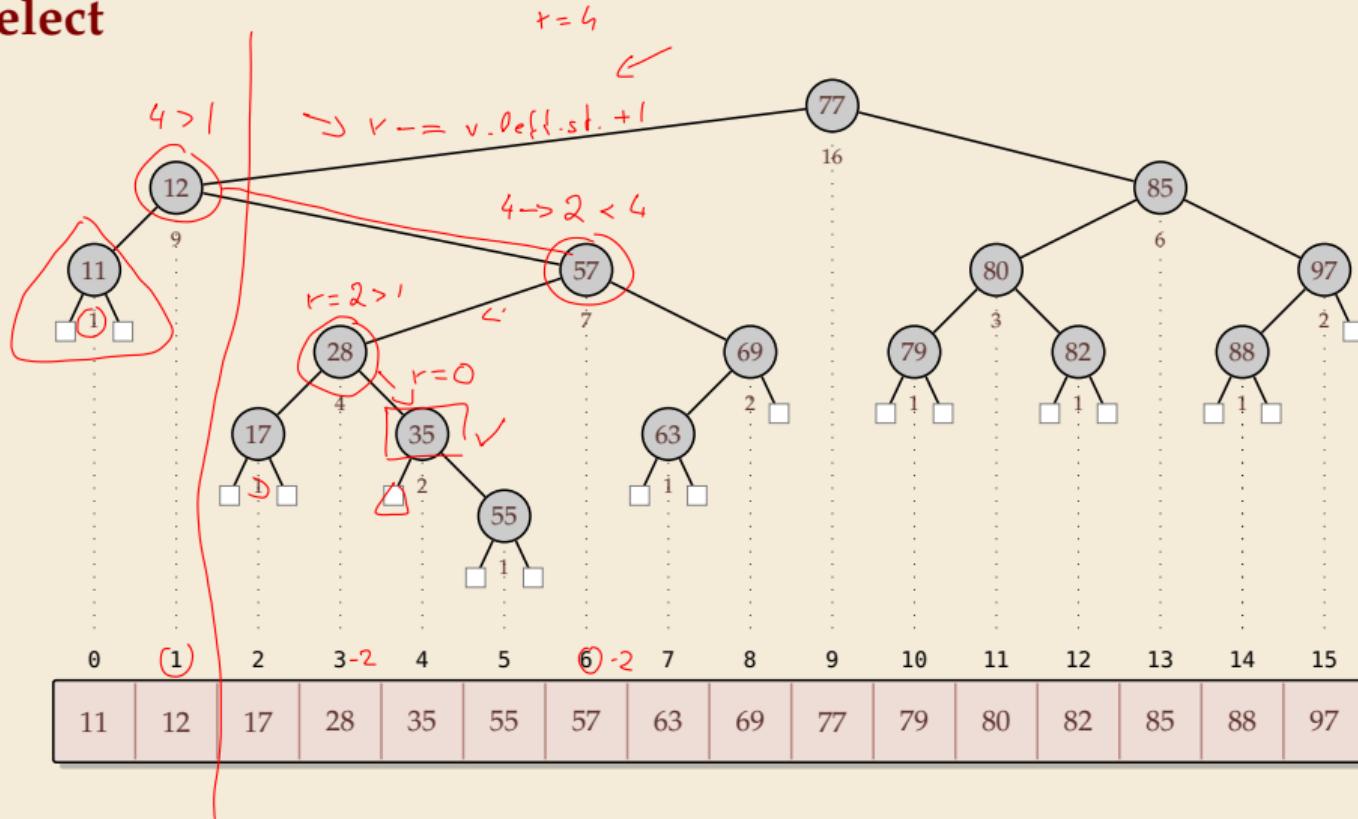


Select

`select(4) = "A[4]"`



Select



each node stores its subtree size (not rank)

can be maintained upon updates

