

2

Fundamental Data Structures

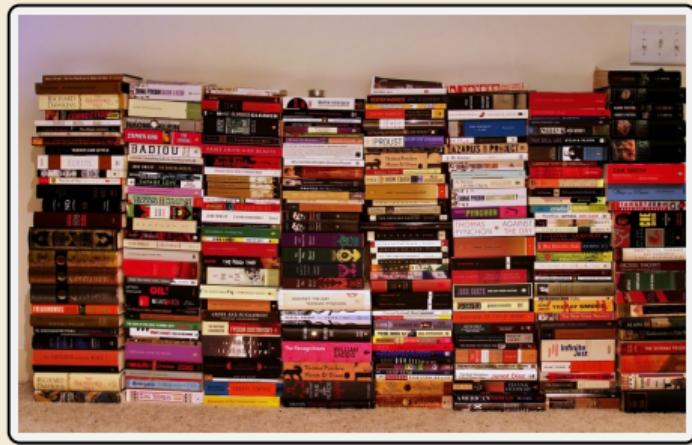
5 October 2022

Sebastian Wild

Learning Outcomes

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

Unit 2: Fundamental Data Structures



Outline

2 Fundamental Data Structures

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

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 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!

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, 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



data structures

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

≈ Java/Python class
(non abstract)

VS.

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

abst

≈ Java interface, Python ABC
(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/comp526*

Clicker Question

Which of the following are examples of abstract data types?

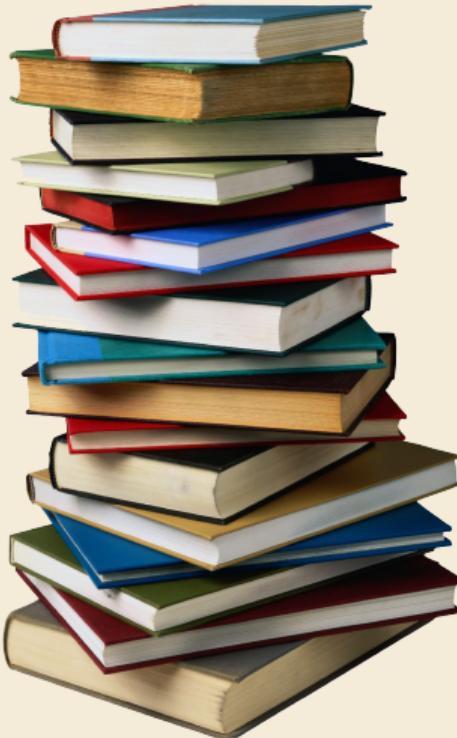


- A ADT
- B Stack ✓
- C Deque ✓
- D Linked list
- E binary search tree
- F Queue ✓
- G resizable array
- H heap
- I priority queue ✓
- J dictionary/symbol table ✓
- K hash table



→ *sli.do/comp526*

Stacks



Stack ADT

- ▶ **top()**

Return the topmost item on the stack
Does not modify the stack.

- ▶ **push(x)**

Add x onto the top of the stack.

- ▶ **pop()**

Remove the topmost item from the stack
(and return it).

- ▶ **isEmpty()**

Returns true iff stack is empty.

- ▶ **create()**

Create and return an new empty stack.

Clicker Question

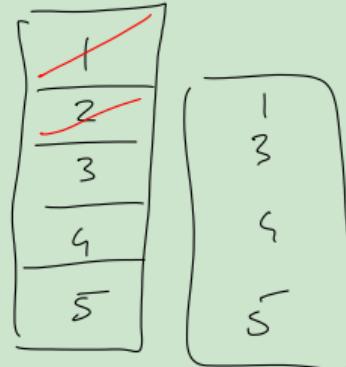
Suppose a stack initially contains the numbers 1, 2, 3, 4, 5 with 1 at the top.

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

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



- A 1,2,3,1
- B 3,4,5,1
- C $\overbrace{1,3,4,5}$
- D empty
- E 1,2,3,4,5



→ sli.do/comp526

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~~

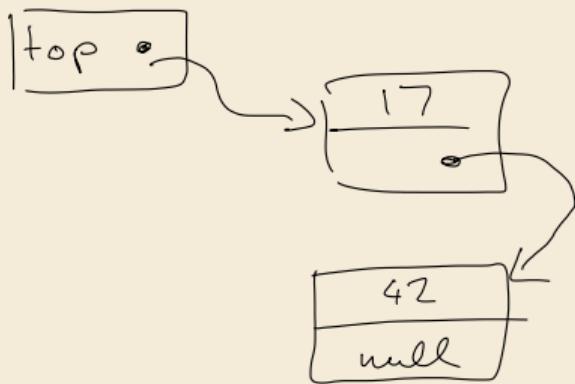


→ sli.do/comp526

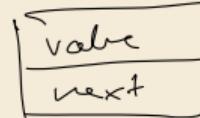
Linked-list implementation for Stack

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
```



↓ is Empty

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

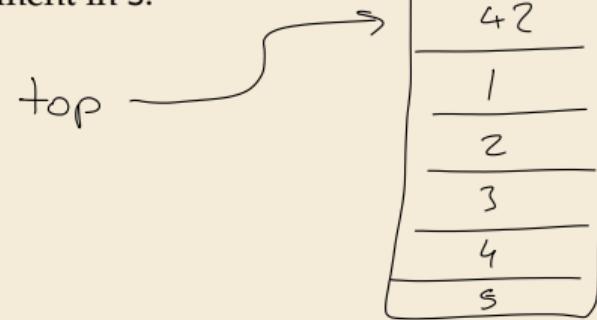
Can we avoid extra space for pointers?

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 $O(1)$ 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];` 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!

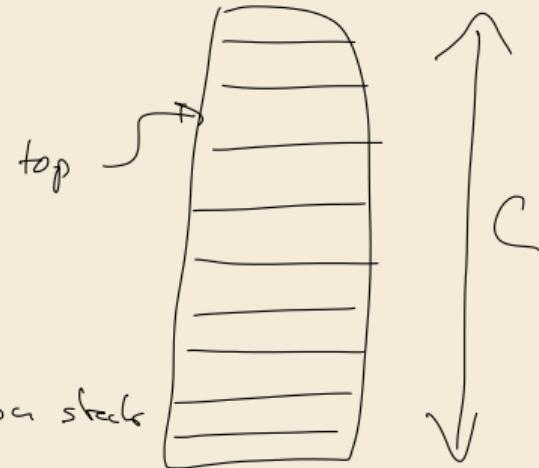
Doubling trick

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!

elements on stack



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?

- 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.



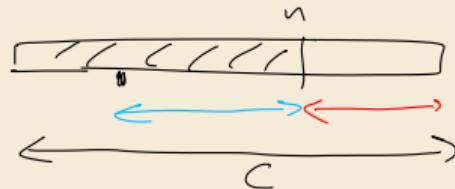
→ *sli.do/comp526*

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

- ▶ 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! $\frac{1}{4}C \leq n \leq C$

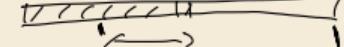


$$\frac{1}{4}C \leq n \leq C$$

distance to boundary

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

since $n \leq C \leq 4n$



- ▶ amortized cost of an operation = actual cost (array accesses) $- 4 \cdot \text{change in } \Phi$

▶ cheap push/pop: actual cost 1 array access, consumes ≤ 1 credits \rightsquigarrow amortized cost ≤ 5

▶ copying push: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n + 1$ credits \rightsquigarrow amortized cost ≤ 5

▶ copying pop: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n - 1$ credits \rightsquigarrow amortized cost 5

~~more~~



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

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

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

$$\sum_{i=1}^m a_i \leq 5m \geq \sum_{i=1}^m a_i = \sum_{i=1}^m c_i - 4 \underbrace{\sum_{i=1}^m (\phi_i - \phi_{i-1})}_{\phi_m - \phi_0}$$

$$\begin{aligned} \sum_{i=1}^m c_i &\leq 5m + 4\phi_m - 4\phi_0 \\ &\leq 5m + 4\phi_m \end{aligned}$$

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*

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*

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 ↗ Bag

Implementation of Bag usually just a Stack or a Queue

2.3 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/comp526*

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 (MaxPQ):~~ Min
(MaxPQ):

- ▶ `construct(A)`
Construct from elements in array A .
- ▶ `insert(x, p)`
Insert item x with priority p into PQ.
- ▶ ~~min~~
~~max()~~
Return item with ~~largest~~ ^{smallest} priority. (Does not modify the PQ.)
- ▶ ~~delMax~~
^{Min}
`delMax()`
Remove the item with ~~largest~~ ^{smallest} priority and return it.
- ▶ `changeKey(x, p')`
Update x 's priority to p'
~~de~~
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, 9, 1 in that order. What is the result of `delMax()`?



A $-\infty$

B 1

C 2

D 4

E 7

F 9

G not allowed



→ sli.do/comp526

Clicker Question

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



A ∞

D 4

G ~~not allowed~~

B 1

E 7

C 2

F 9 ✓



→ sli.do/comp526

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

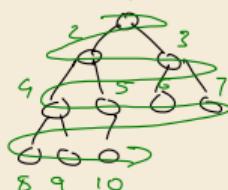
- unordered list $\rightsquigarrow \Theta(1)$ insert, but $\Theta(n)$ delMax
- sorted list $\rightsquigarrow \Theta(1)$ delMax, but $\Theta(n)$ insert

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

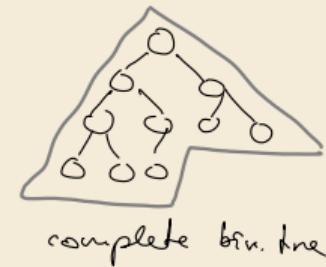
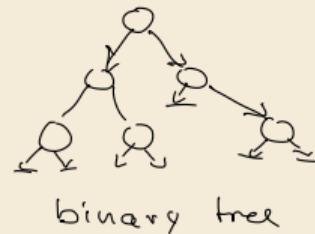
Yes! *Binary heaps*.

Array view

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



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



index 1 2 3 4 5 6 7 8 9 10
value 87 66 70 50 40 55 10 16 1 25

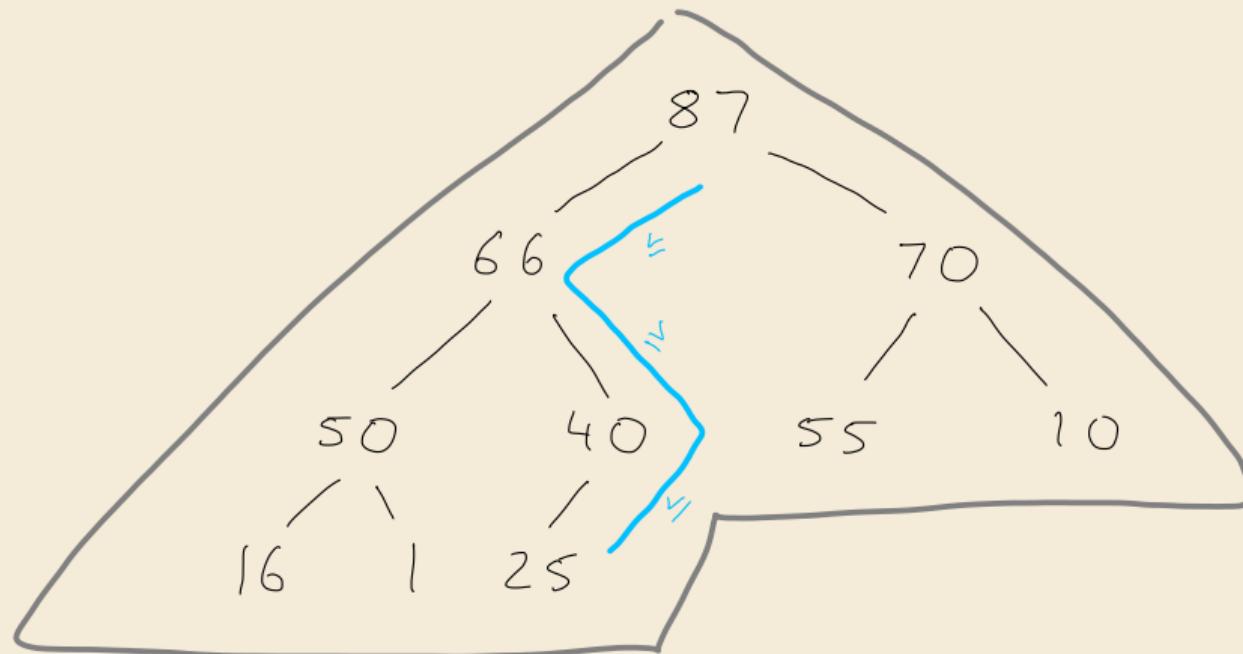
Tree view

all but last level full
last level flush left

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

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

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

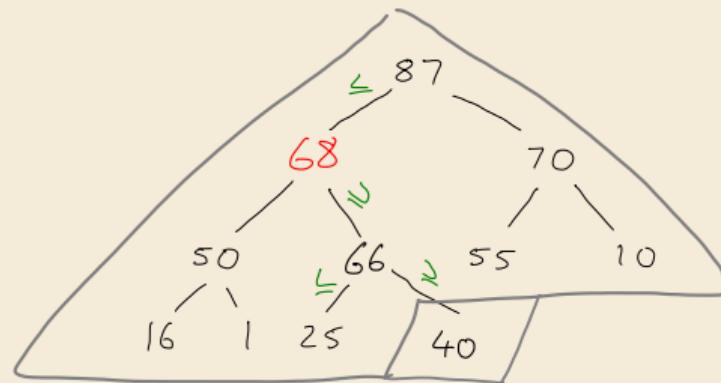
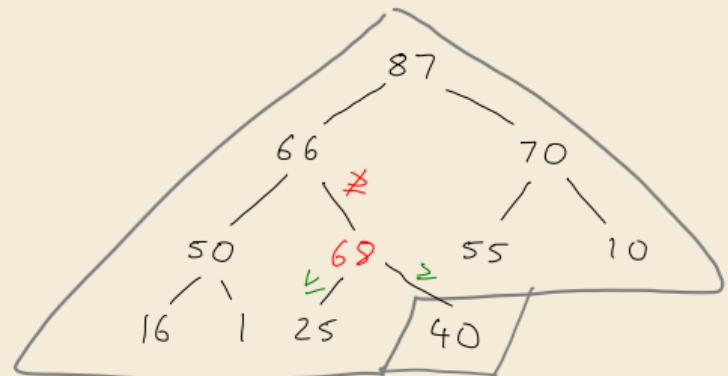
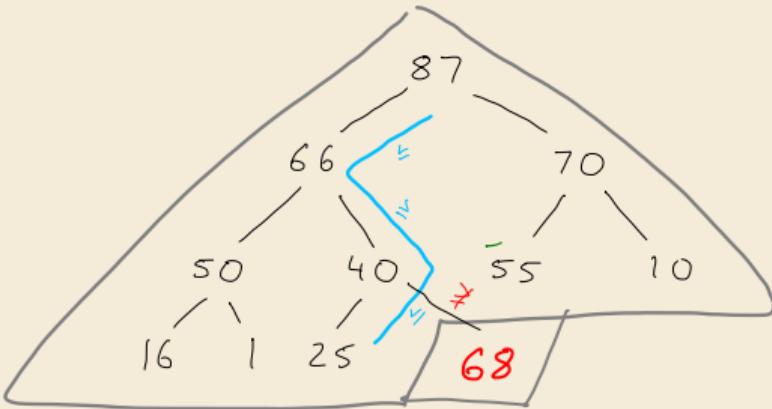


→ *sli.do/comp526*

2.4 Operations on Binary Heaps

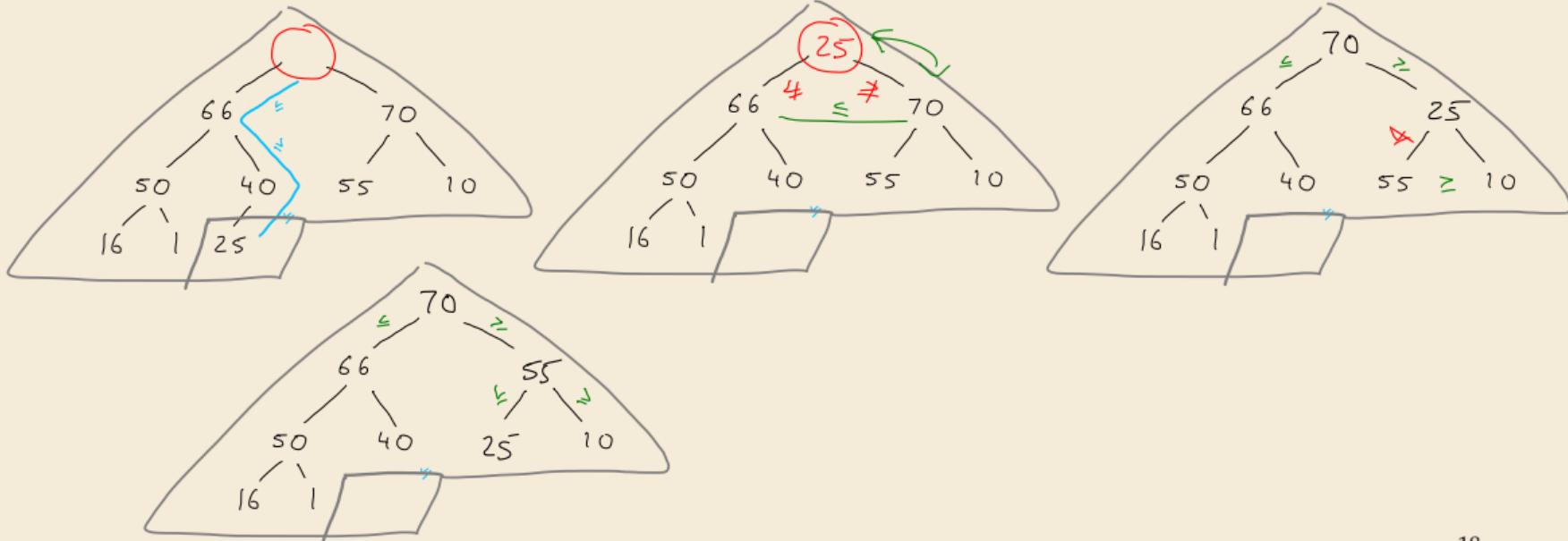
Insert

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). $\rightarrow 87$
2. Move last element (bottom-most, rightmost) into root.
3. Let root key *sink* in heap to repair heap order.

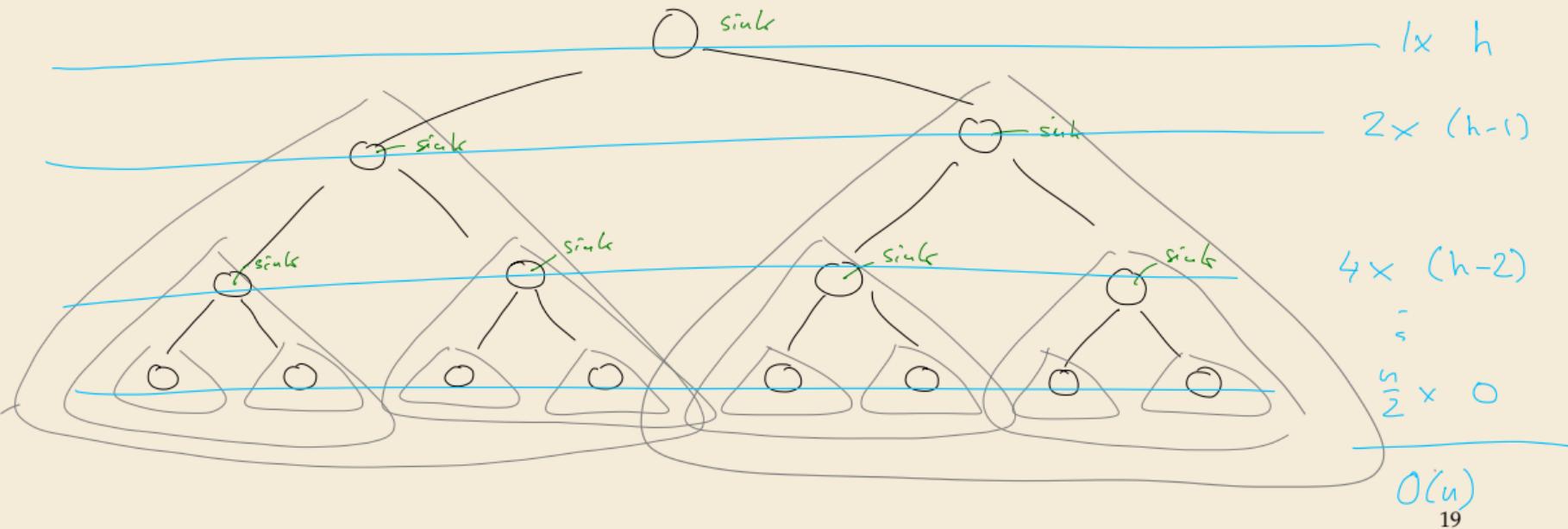


Heap construction

► n times insert $\rightsquigarrow \Theta(n \log n)$ 

► instead:

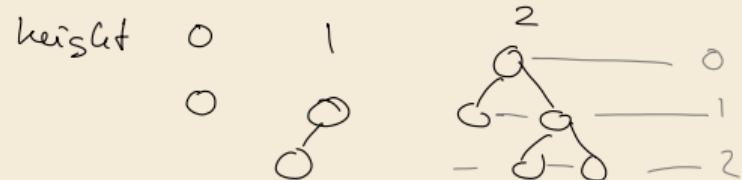
1. Start with singleton heaps (one element)
2. Repeatedly merge two heaps of height k with new element into heap of height $k + 1$



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

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>	$O(1)$