Heaps

Priority Queues

Problem.

- Let S={(s1,p1), (s2,p2),…,(sn,pn)} where s(i) is an object and p(i) is the priority of s(i).

How to design a data structure/algorithm to support the following operations over S?

GetMin: Returns the element of S with minimum priority

Insert(s,p): Insert a new element (s,p) in S

DeleteMin: Remove the element in S with minimum priority

Priority Queues

Solution 1. Store S in a sorted linked list

GetMin :
Insert:
DeleteMin:

Solution 2. Use a linked list with the pair with minimum p at the first position

GetMin :
Insert:
DeleteMin:

Can we do better? How?

Priority Queues

Solution 1. Store S in a sorted linked list

GetMin : $O(1)$ time
Insert: $O(n)$ time
DeleteMin: $O(1)$ time

Solution 2. Use a linked list with the pair with minimum p at the first position

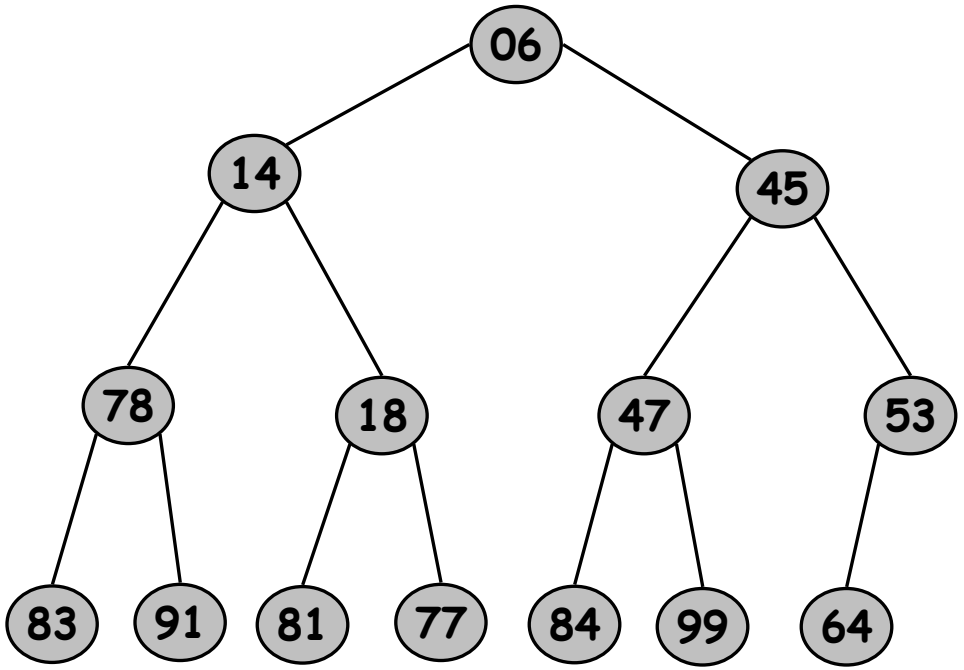
GetMin : $O(1)$ time
Insert: $O(1)$ time
DeleteMin: $O(n)$ time

Can we do better? How?

Binary Heap: Definition

Binary heap.

- Almost complete binary tree
  - filled on all levels, except last, where filled from left to right
- Min-heap ordered
  - every child greater than (or equal to) parent



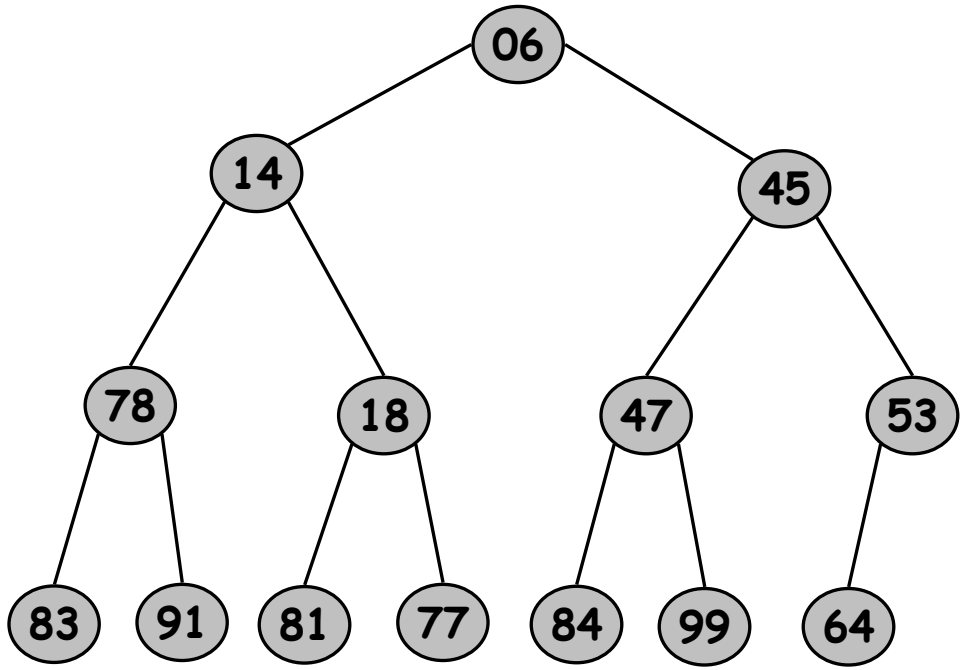
Max-heap is analogous (every child is smaller than its parent)

Binary Heap: Properties

Properties.

- Min element is in
- Heap with N elements has height

In particular can obtain minimum element (GetMin) in O(1)



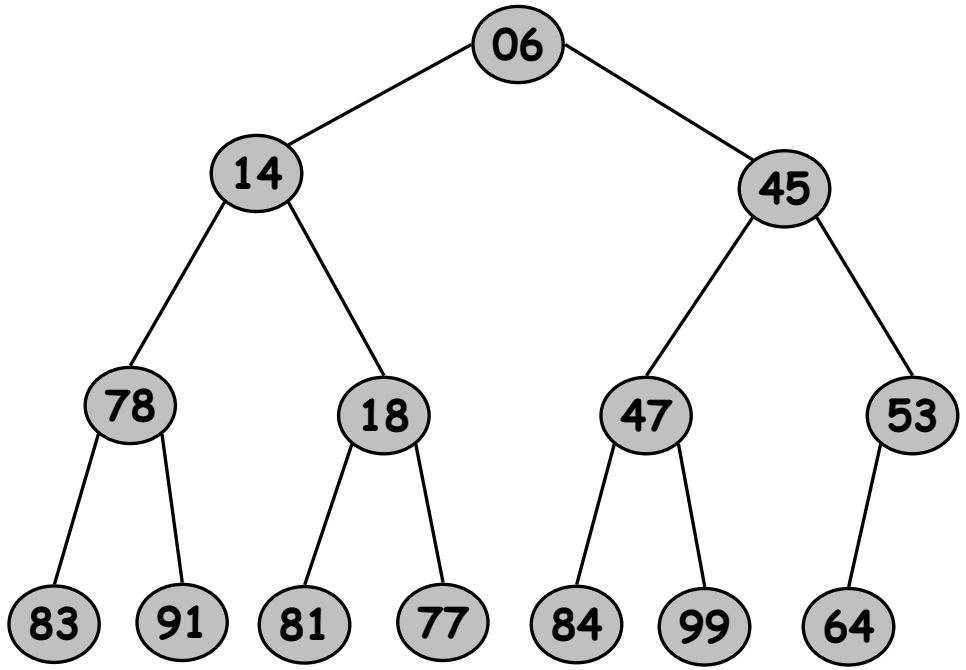
N = 14
Height = 3

Binary Heap: Properties

Properties.

- Min element is in root.
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$.

In particular can obtain minimum element (GetMin) in O(1)



N = 14
Height = 3

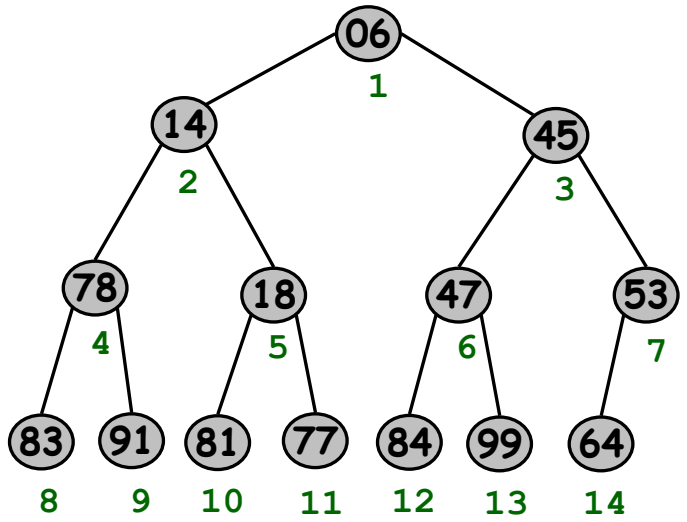
Binary Heaps: Array Implementation

Implementing binary heaps

- Use an array: no need for explicit parent or child pointers.
  - `Parent(i) = ⌊i/2⌋`
  - `Left(i) = 2i`
  - `Right(i) = 2i + 1`

Only important properties for us:

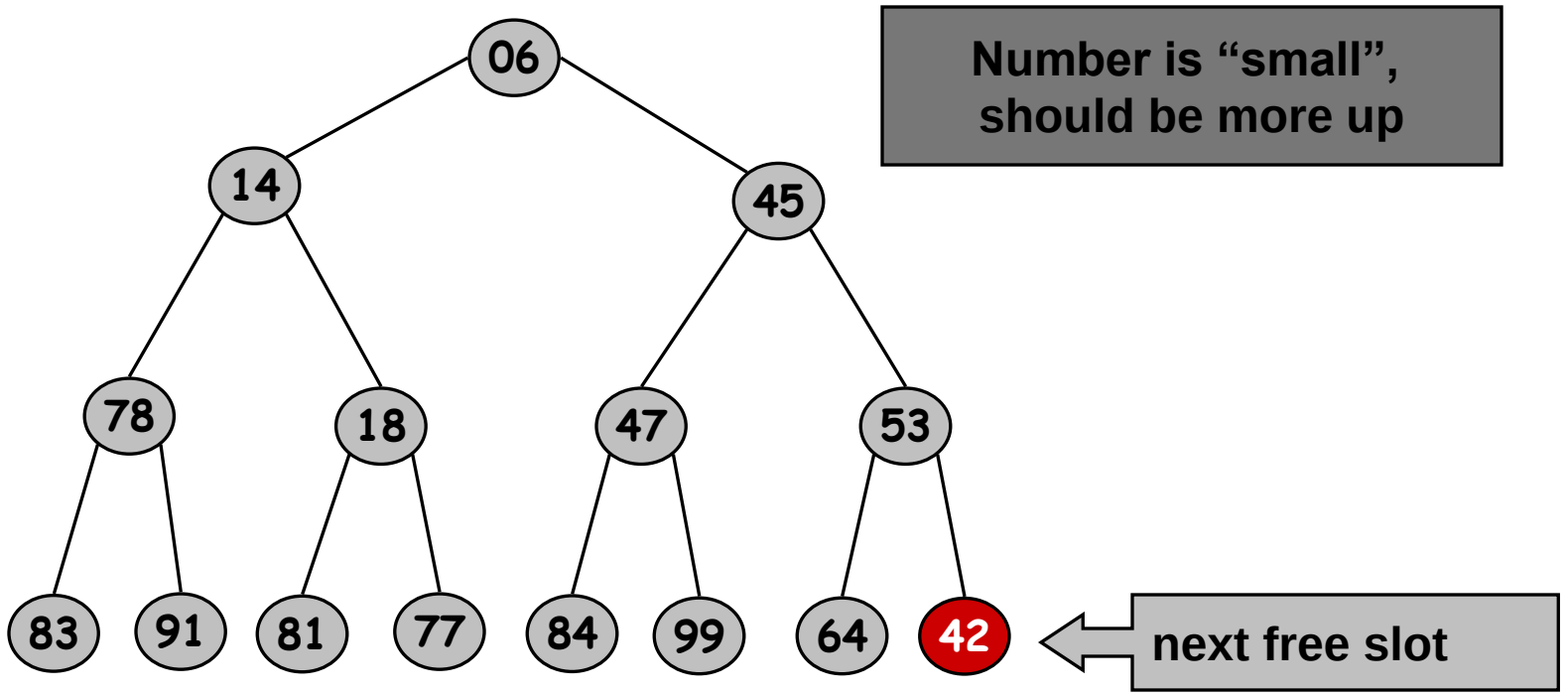
1. Last level of the tree is occupied from left to right
2. We can find the first (leftmost) empty space in constant time
   (keep track of first empty space with an integer firstEmpty)



Binary Heap: Insertion

Insert element x into heap.

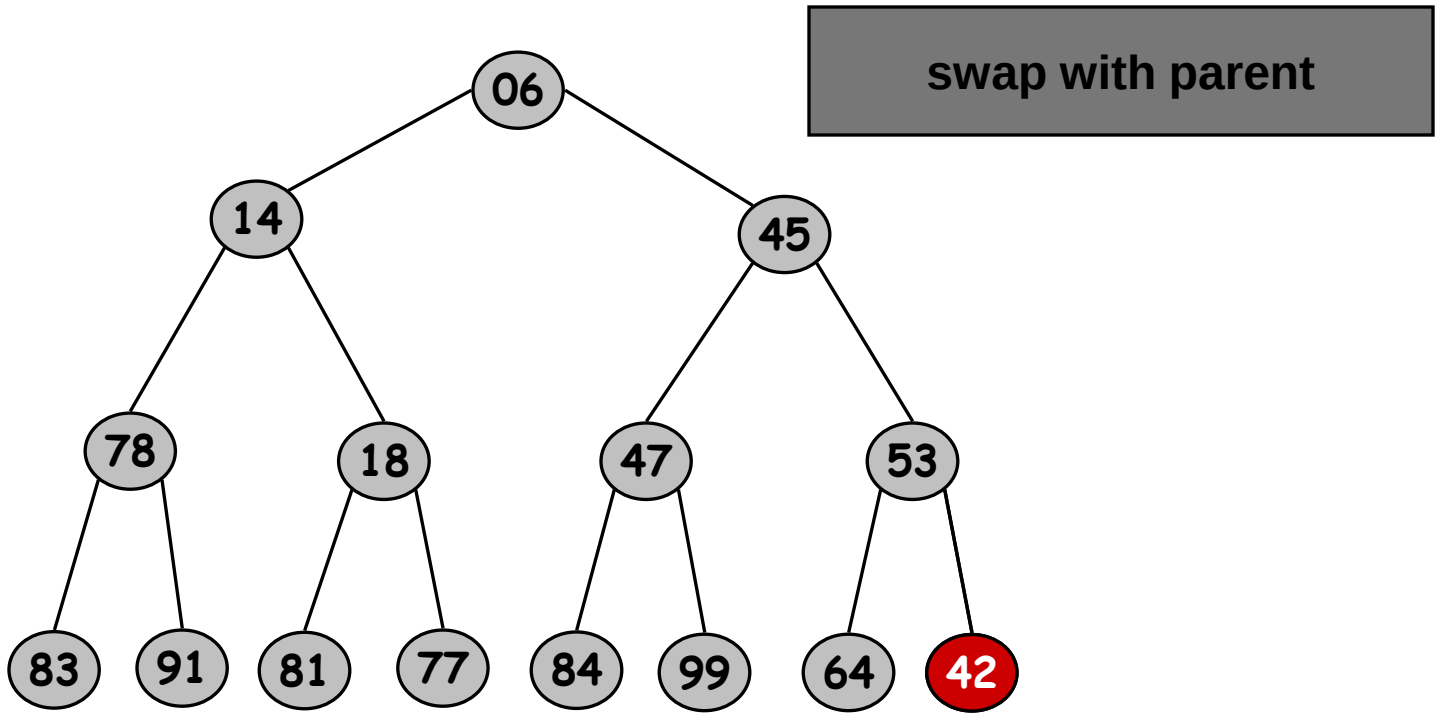
- Insert into first available slot.
- Bubble up until it's heap ordered.



Binary Heap: Insertion

Insert element x into heap.

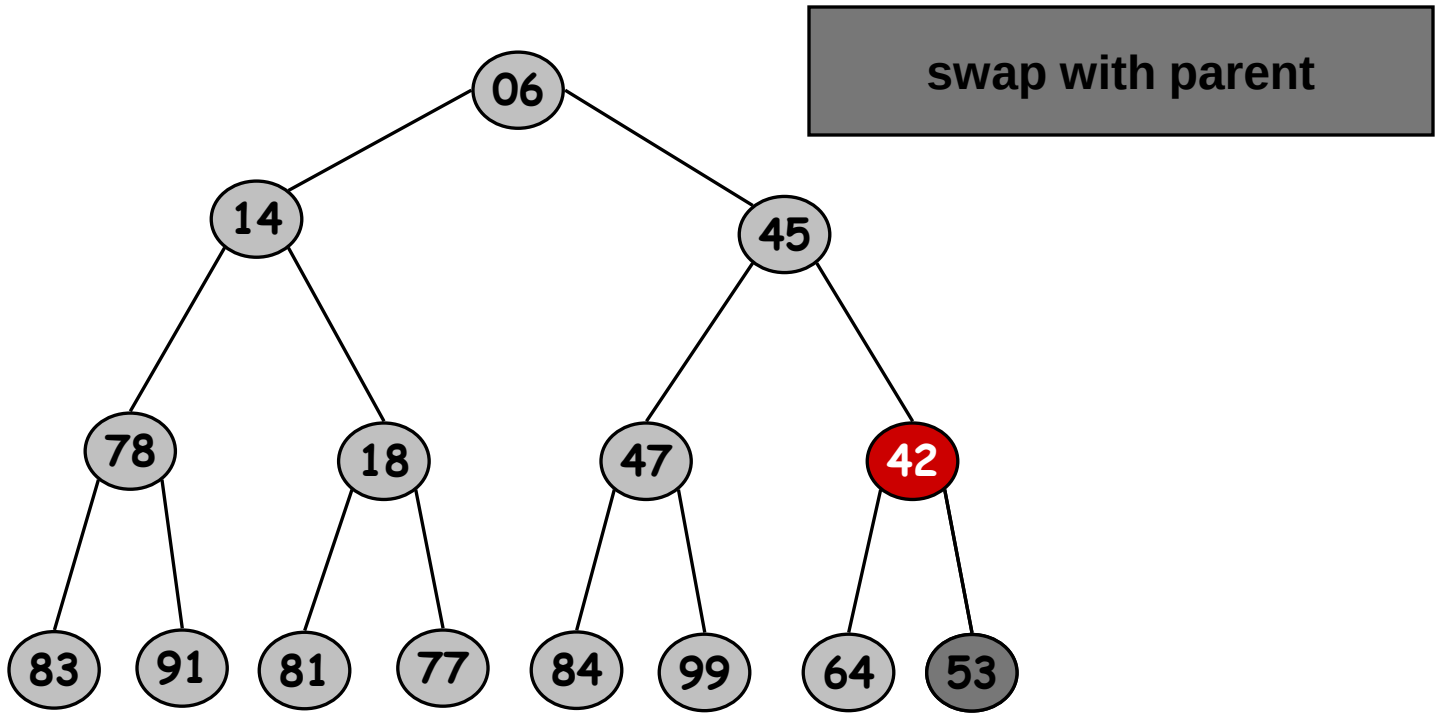
- Insert into first available slot.
- Bubble up until it's heap ordered.



Binary Heap: Insertion

Insert element x into heap.

- Insert into first available slot.
- Bubble up until it's heap ordered.



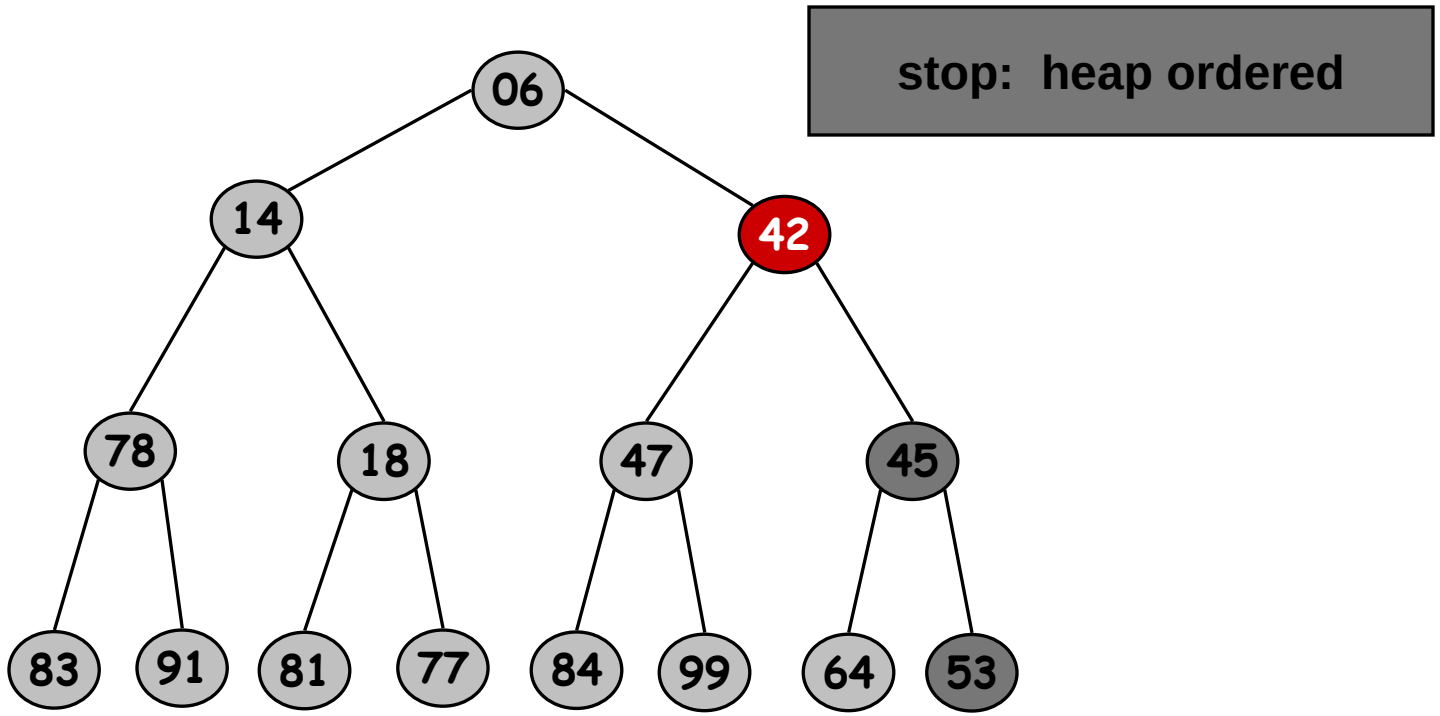
Binary Heap: Insertion

Insert element x into heap.

- Insert into first available slot.
- Bubble up until it's heap ordered.

Q: What is the time complexity of Insert?

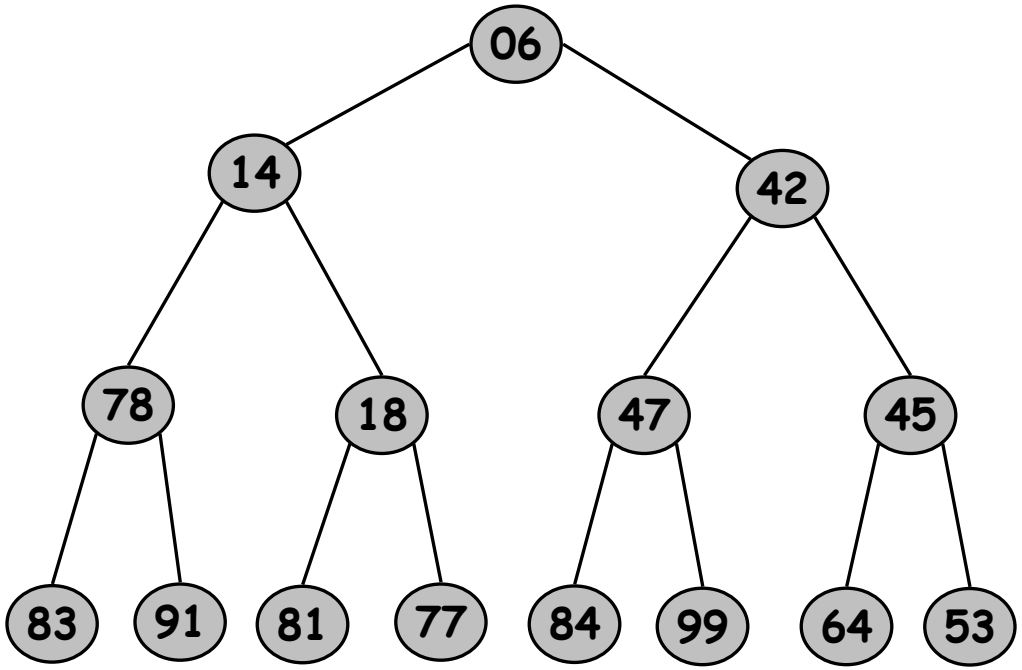
A: O(log N) operations



Binary Heap: Decrease Key

Decrease key of element x to k.

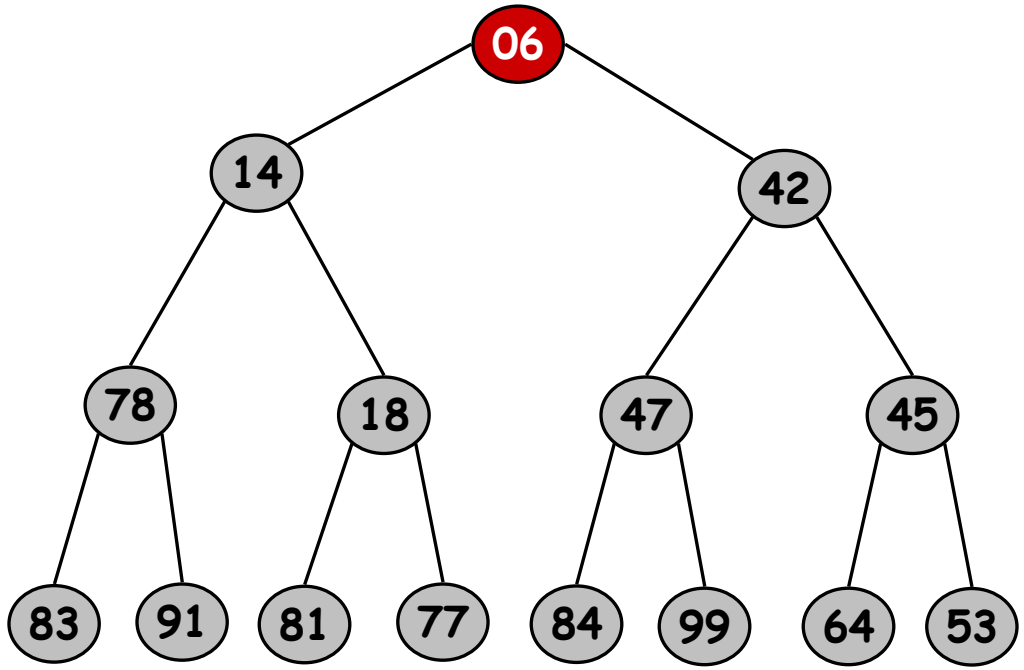
- Bubble up until it's heap ordered.
- O(log N) operations.



Binary Heap: Delete Min

Delete minimum element from heap.

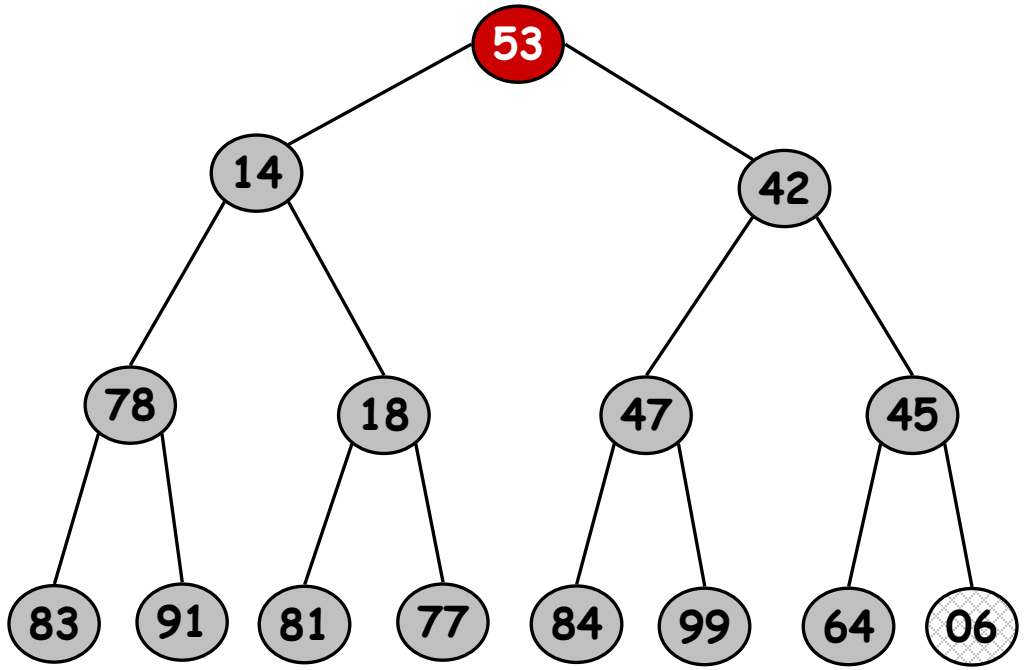
- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child



Binary Heap: Delete Min

Delete minimum element from heap.

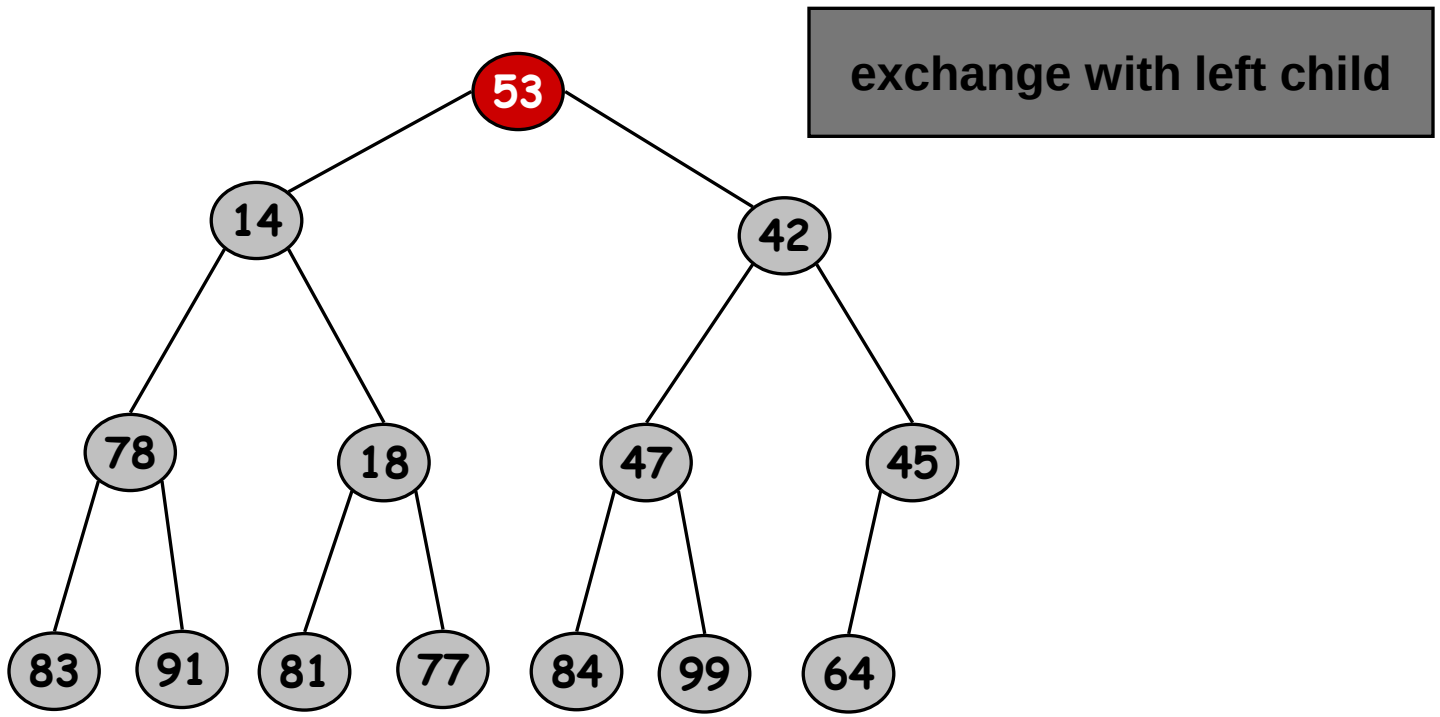
- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child



Binary Heap: Delete Min

Delete minimum element from heap.

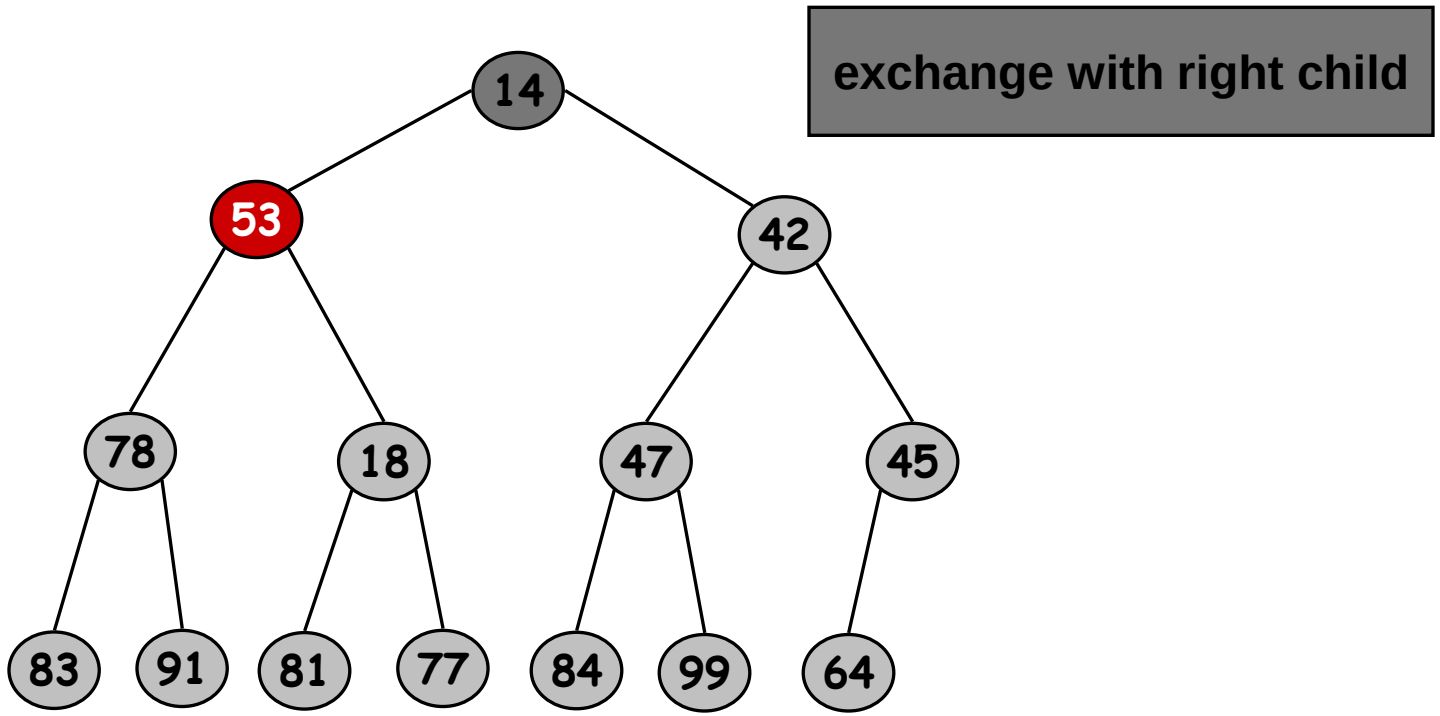
- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child



Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child



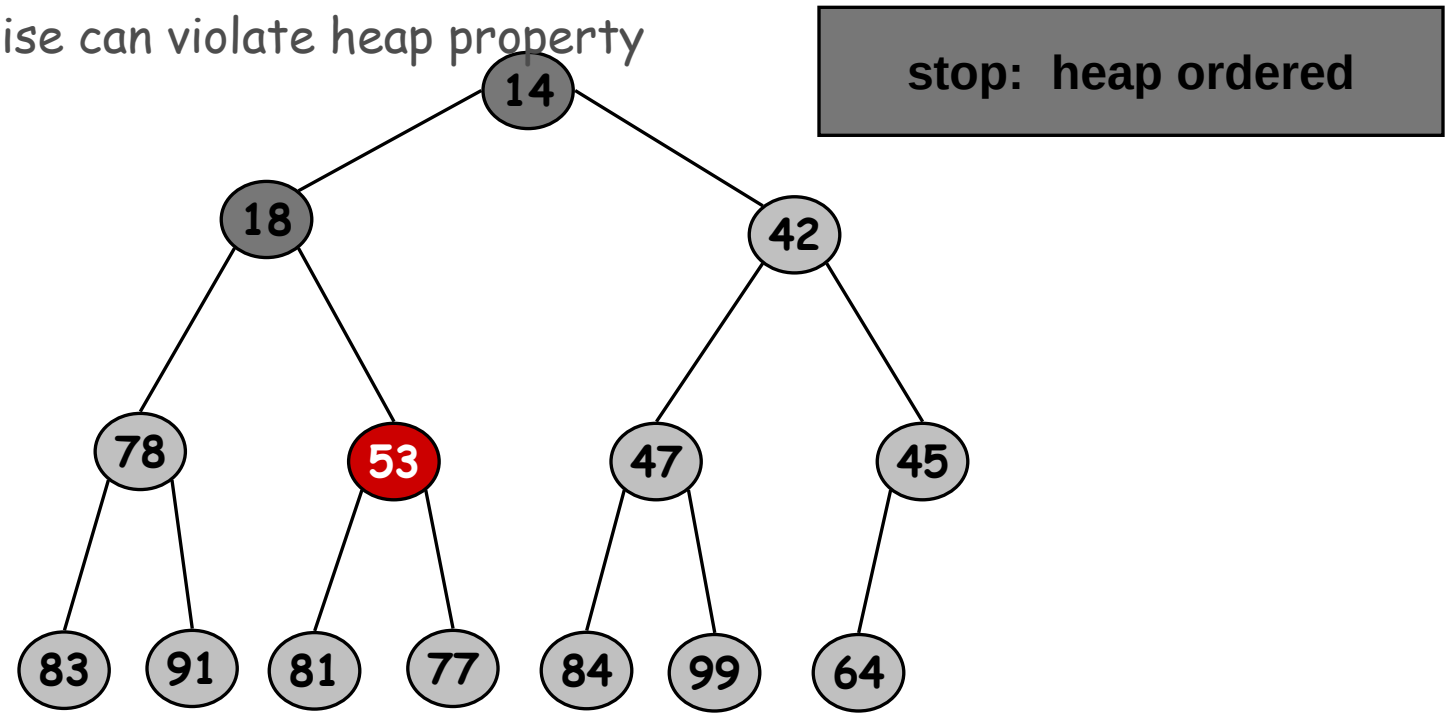
Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child

Q: Why do we exchange with smallest child?

A: Otherwise can violate heap property



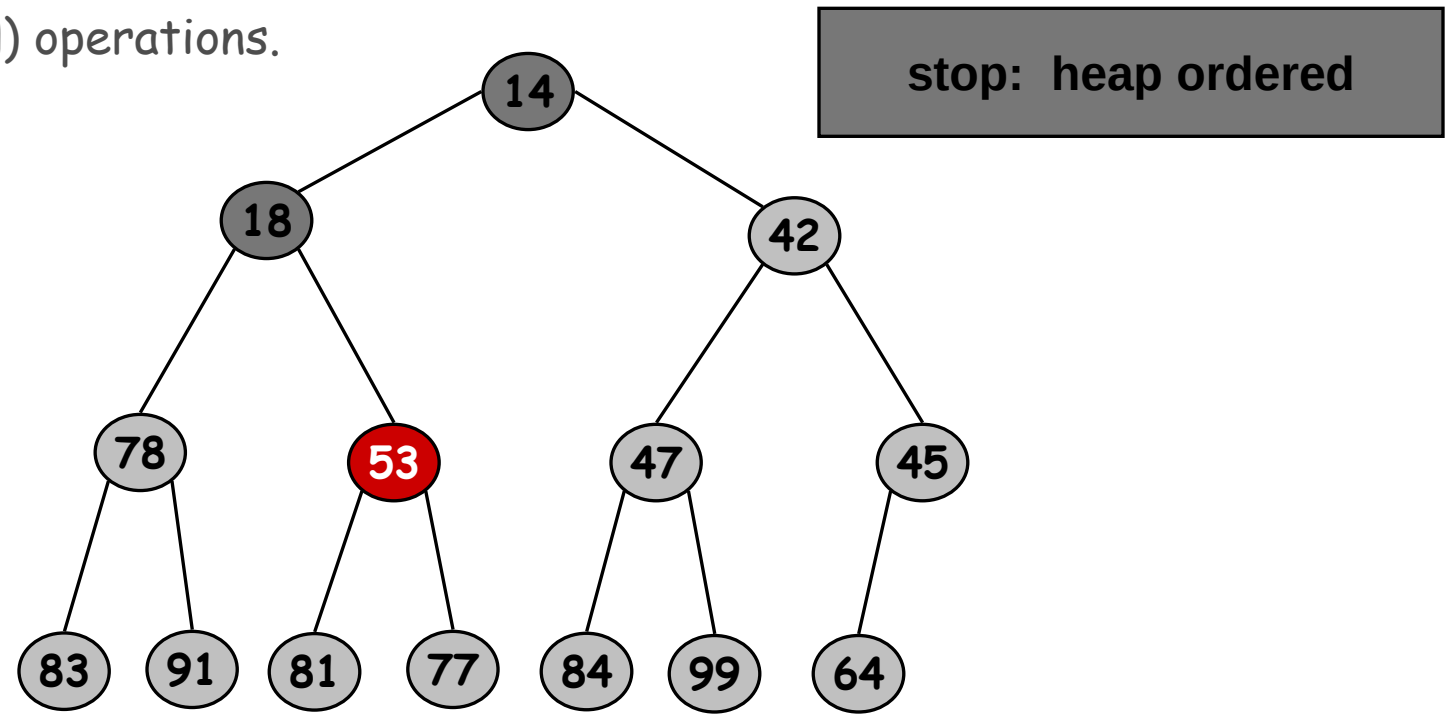
Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf in the last level
- Delete this leaf
- Bubble root down until it's heap ordered, exchanging it with smallest child

Q: What is the time complexity of DeleteMin?

A: O(log N) operations.



Priority Queues

Solution 1. Use a sorted list
GetMin: $O(1)$ time
Insert: $O(n)$ time
DeleteMin: $O(1)$ time

Solution 2. Use a list with the pair with minimum p at the first position
GetMin: $O(1)$ time
Insert: $O(1)$ time
DeleteMin: $O(n)$ time

Solution 3. Binary Heap
GetMin: $O(1)$ time
Insert: $O(\log n)$ time
DeleteMin: $O(\log n)$ time

Application: Heapsort

Q: Can we sort N numbers using a Binary Heap?

- Insert N items into a binary min-heap
  - O (N log N)
- Perform N GetMin and DeleteMin operations.
  - O(N log N)
- Overall
  - O(N log N) sort.

Building a Heap

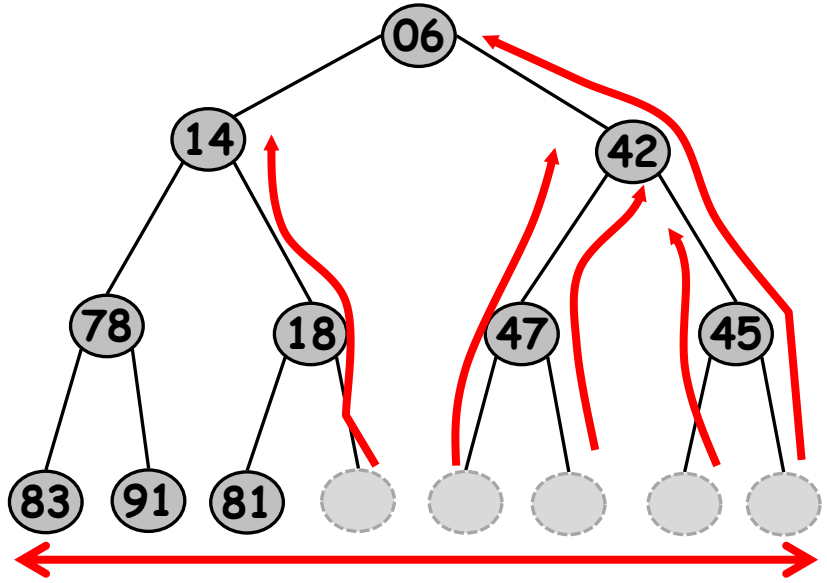
In Heapsort, we just built a heap for n numbers by inserting them one at a time

Time complexity of this procedure: $O(n \log n)$

Q: Can we build a heap faster?

Bottleneck:

- The bottom layers have many nodes…
- …when adding, each of them may move a lot up in the heap



Building a Heap in O(n)

Idea: Move nodes down instead of up

- Few node in the top of the tree, only they can move down a lot

Operation down(node): keeps exchanging node with smallest child until it is in the right position
(that is, has smaller priorities than its children)

Building a Heap of numbers a1, a2, … an in O(n):

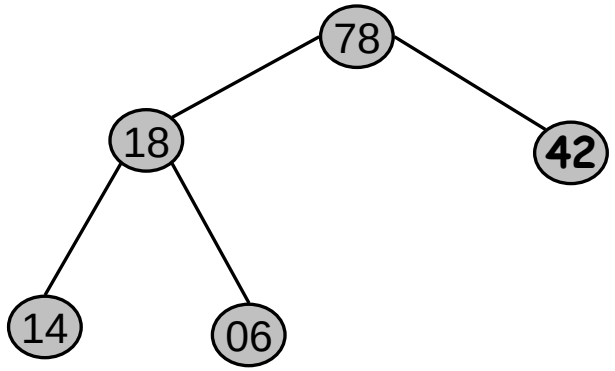
HeapBuild1 (Cormen)

- Make an almost complete binary tree with numbers in any position
- Traverse tree from bottom up applying down(node)

Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

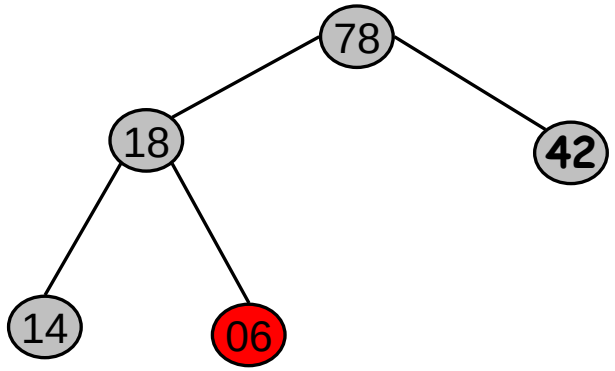
Apply *down()* to node with 06



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

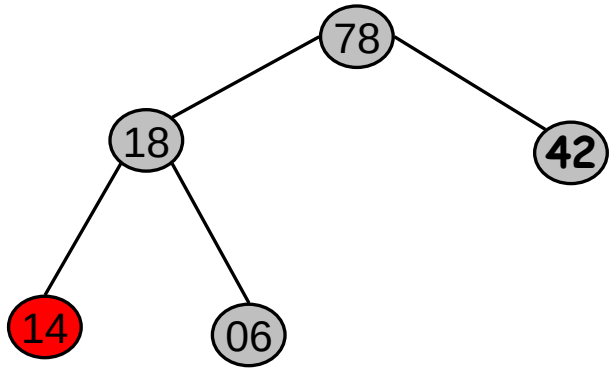
Apply *down()* to node with 06



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

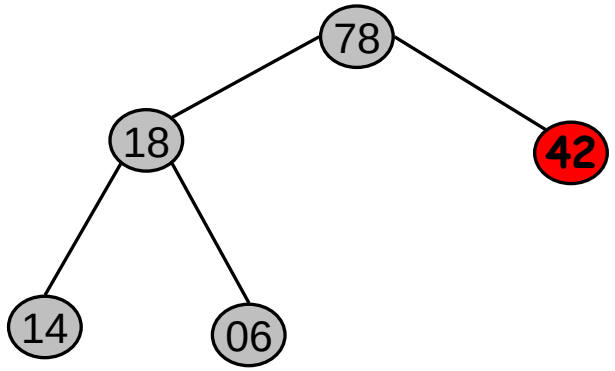
Apply *down()* to node with 14



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

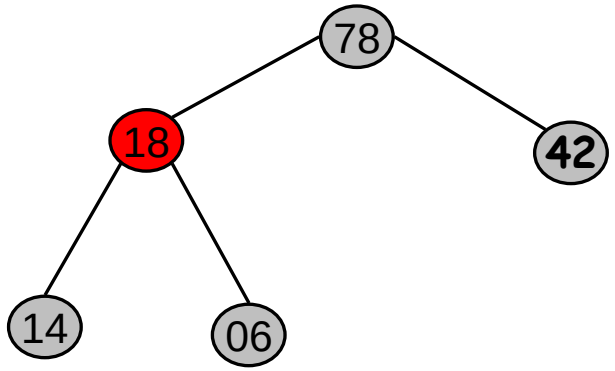
Apply *down()* to node with 42



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

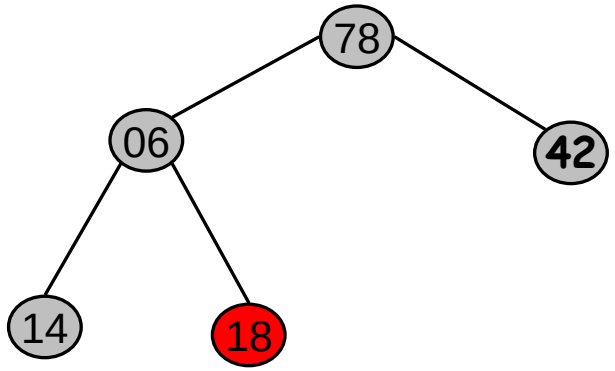
Apply *down()* to node with 18



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

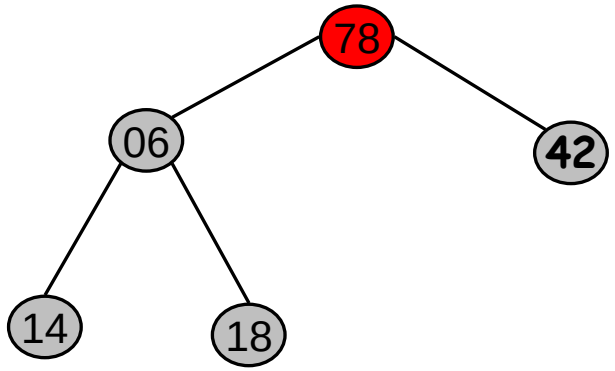
Apply *down()* to node with 18



Building a Heap in O(n)

Example: Numbers 14, 06, 42, 78, 18

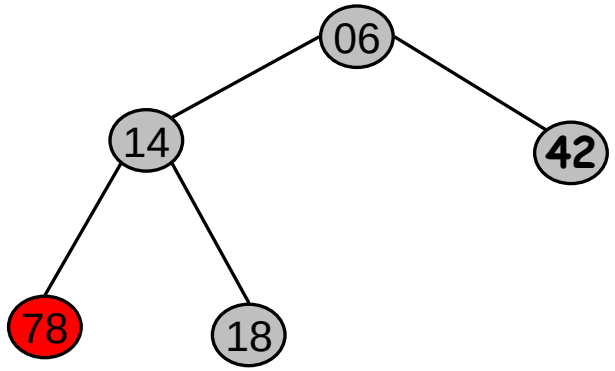
Apply *down()* to node with 78 twice



Building a Heap in O(n)

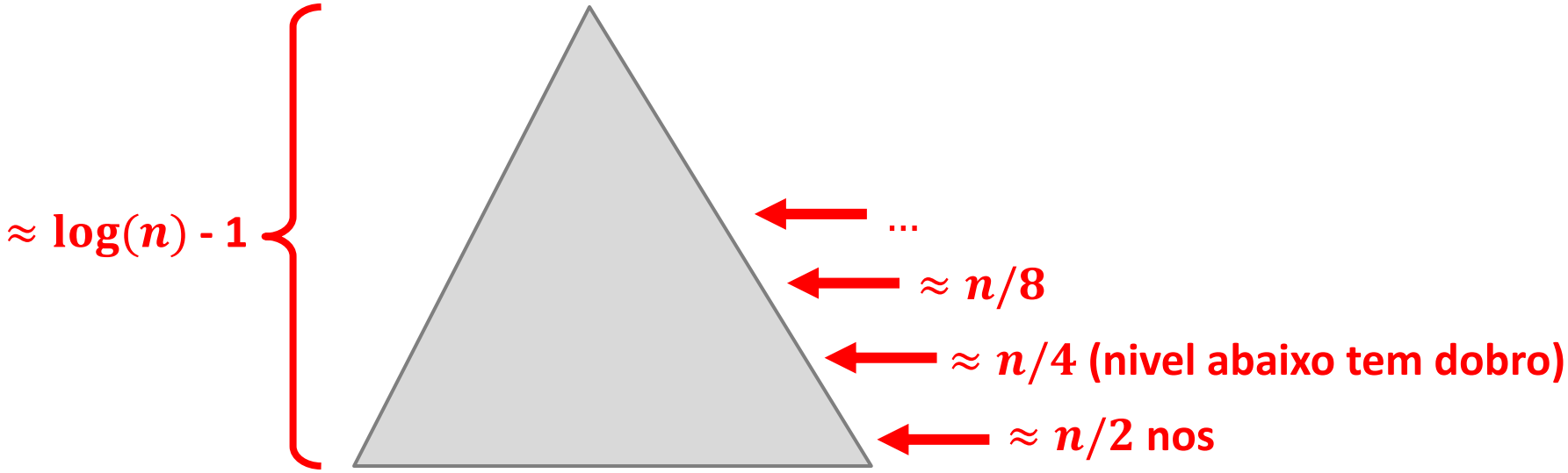
Example: Numbers 14, 06, 42, 78, 18

Apply *down()* to node with 78 twice



Building a Heap in O(n): Analysis

Q: Numa heap com n nós, quantos nós temos no nível i de baixo pra cima?



$\Rightarrow$ nivel i de baixo pra cima tem no máximo $n/2^i$ nós

Building a Heap in O(n): Analysis

Custo do down(node): se node está no nível i de baixo para cima, esse down custa O(i) operações

$$Custo\ \text{total} = \sum_{i=1}^{\log n} i.\frac{n}{2^i} =$$

Building a Heap in O(n): Analysis

Custo do down(node): se node está no nível i de baixo para cima, esse down custa O(i) operações

$$Custo\ \text{total} = \sum_{i=1}^{\log n} i.\frac{n}{2^i} = \Theta(n)$$

Exercise

Exercise 1: Can we use use the procedures we know (Insert, ExtractMin, DeleteMin, DecreaseKey) to implement the IncreaseKey(x,k) that increases the key of an element x to value k?

If so, analyze the complexity of this new operation.

Solution: One possibility is to remove element x and add it with key k.
To remove this value we can:
1) Obtain the minimum MIN using GetMin
2) reduce the key of x to MIN-1 using DecreaseKey (so now x is minimum)
3) Apply RemoveMin to remove it

The total complexity is O(log n), since it uses 4 operations that are at most O(log n).

Exercise

Exercicio 2: Seja S um conjunto de n numeros reais distintos. Explique como seria um algoritmo linear para encontrar os $\sqrt{n}$ menores numeros do conjunto S e analise sua complexidade

Solucao: Crie uma min-heap com os elementos de S e aplique ExtractMin $\sqrt{n}$ vezes; retorne esses elementos removidos

Complexidade: O(n) pra montar o heap, $O(\sqrt{n}\,.\log(n)) = O(n)$ para remover of elementos => total: O(n)

Extra: Hash Tables

Storage and retrieval

Collection of pairs (k1, s1), (k2, s2),…,(kn, sn) where si is an object and ki is the key of si (keys are unique)

How to design a data structure/algorithm to support the following operations?

Insert(k,s): Insert object s with key k to the data structure

Find(k): Checks if there is object with key k in the data structure or not

Delete(k): Remove object with key k from the data structure (if exists)

Storage and retrieval

Assume key is a number for now

Solution 1. Store in a linked list sorted by key

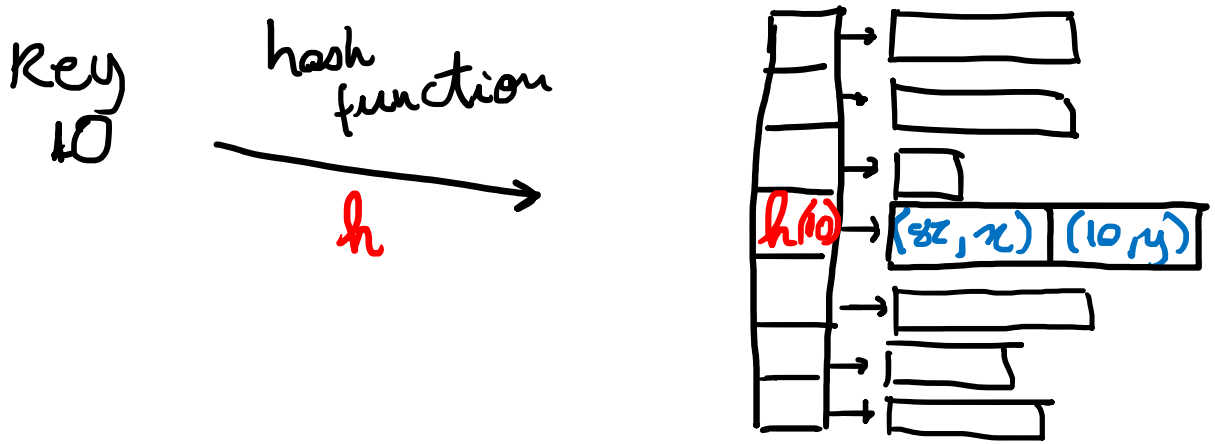
Insert :
Find:
Delete:

Solution 2. Use heap

Insert :
Find:
Delete:

Can we do better? How?

Hash Tables

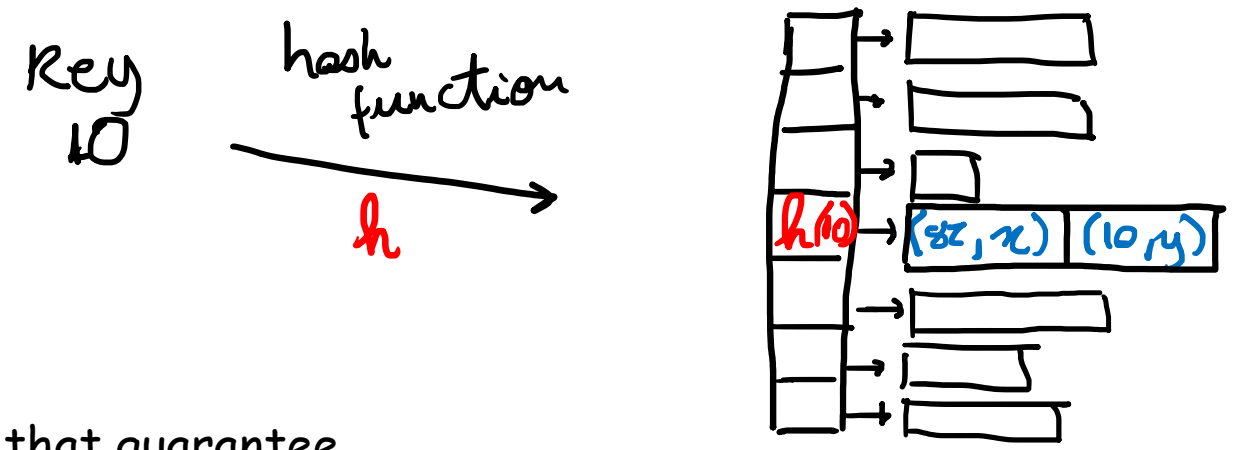


Idea:

- Use multiple unsorted lists *(open hashing)*
- Hash function h(k) tells in which list to store object with key k

Hash Tables: Complexity

Usually computing h(k) is $O(1)$



Insert(k,s): $O(1)$
Find(k): depends on the size of lists...
Delete(k): depends on the size of lists...

There are more complicated hash table structures that guarantee

Insert(k,s): $O(1)$
Find(k): $O(1)$
Delete(k): $O(1)$

but in an "amortized sense", **not worst-case** (per operation)

Importante pra sua vida: código, entrevista de emprego
MAS NAO PODE USAR NA DISCIPLINA!! Não é garantia de pior-caso