Mergeable Heaps

David Kauchak
CS140
Spring 2024

1

Admin

Assignment 2 graded

Assignment 3

Assignment 4

2

Another set data structure

Idea: store data in a collection of arrays

- array *i* has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

Binary array set

https://www.nayuki.io/page/binary-array-set#:~:text=The%20binary%20array%20set%20data,a%20BAS%20is%20practically%20negligible.

3

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are **complete** except the last

Max heap vs. min heap

4

Binary heap - operations

Max - return the largest element in the set

ExtractMax – Return and remove the largest element in the set

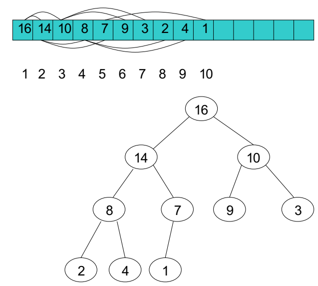
Insert(val) – insert val into the set

IncreaseElement(x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

5

Binary heap representations



6

Heapify

Assume left and right children are heaps, turn current node into a valid heap

HEAPIFY(A, i) aka "sink"

```
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

7

Heapify

Assume left and right children are heaps, turn current set into a valid heap

HEAPIFY(A, i)

```
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

find out which is largest: current, left of right

8

Heapify

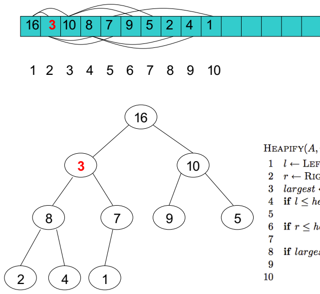
Assume left and right children are heaps, turn current set into a valid heap

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

if a child is larger, swap and recurse

9

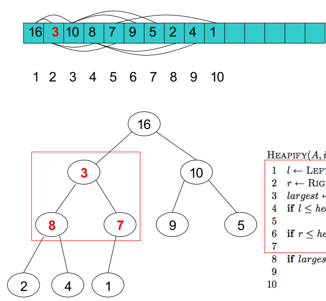
Heapify




```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```


10

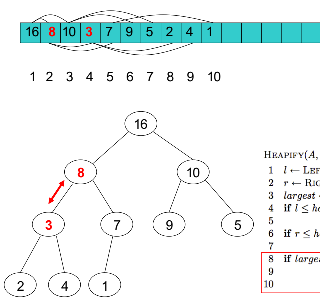
Heapify




```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

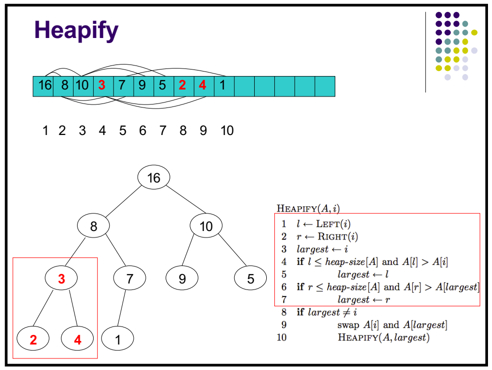

11

Heapify

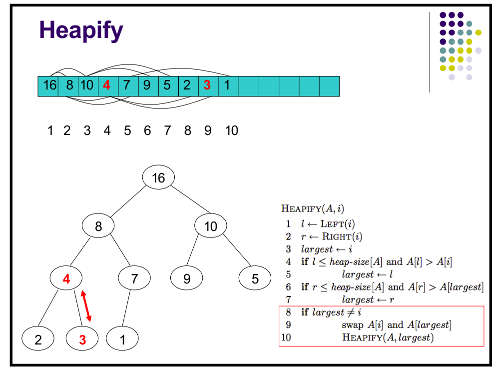



```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

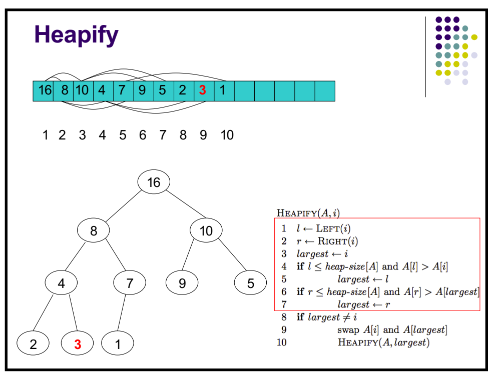

12


Heapify
16 8 10 3 7 9 5 2 4 1
1 2 3 4 5 6 7 8 9 10
16
8 10
3 7 9 5
2 4 1
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 largest ← i
4 if l ≤ heap-size[A] and A[l] > A[i]
5 largest ← l
6 if r ≤ heap-size[A] and A[r] > A[largest]
7 largest ← r
8 if largest ≠ i
9 swap A[i] and A[largest]
10 HEAPIFY(A, largest)

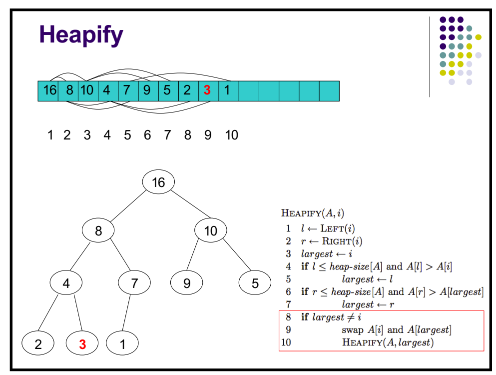
13


Heapify
16 8 10 4 7 9 5 2 3 1
1 2 3 4 5 6 7 8 9 10
16
8 10
4 7 9 5
2 3 1
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 largest ← i
4 if l ≤ heap-size[A] and A[l] > A[i]
5 largest ← l
6 if r ≤ heap-size[A] and A[r] > A[largest]
7 largest ← r
8 if largest ≠ i
9 swap A[i] and A[largest]
10 HEAPIFY(A, largest)

14


Heapify
16 8 10 4 7 9 5 2 3 1
1 2 3 4 5 6 7 8 9 10
16
8 10
4 7 9 5
2 3 1
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 largest ← i
4 if l ≤ heap-size[A] and A[l] > A[i]
5 largest ← l
6 if r ≤ heap-size[A] and A[r] > A[largest]
7 largest ← r
8 if largest ≠ i
9 swap A[i] and A[largest]
10 HEAPIFY(A, largest)

15


Heapify
16 8 10 4 7 9 5 2 3 1
1 2 3 4 5 6 7 8 9 10
16
8 10
4 7 9 5
2 3 1
HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 largest ← i
4 if l ≤ heap-size[A] and A[l] > A[i]
5 largest ← l
6 if r ≤ heap-size[A] and A[r] > A[largest]
7 largest ← r
8 if largest ≠ i
9 swap A[i] and A[largest]
10 HEAPIFY(A, largest)

16

Running time of Heapify

O(height of the tree)

What is the height of the tree?

- Complete binary tree, except for the last level

$$2^h \le n$$

$$h \le \log_2 n$$

O(log n)

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

21

Binary heap - operations

Max - return the largest element in the set

ExtractMax – Return and remove the largest element in the set

Insert(val) – insert val into the set

IncreaseElement(x, val) – increase the value of element x to val

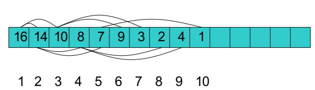
BuildHeap(A) – build a heap from an array of elements

22

Max

What is the largest element in the set?

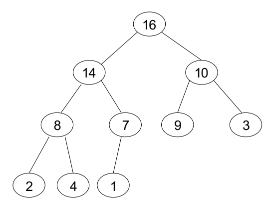
Return A[1]



23

ExtractMax

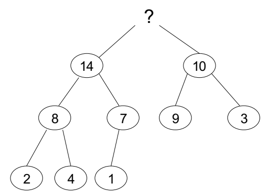
Return and remove the largest element in the set



24

ExtractMax

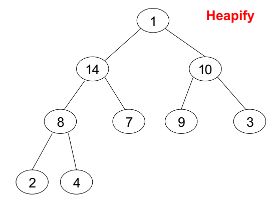
Return and remove the largest element in the set



25

ExtractMax

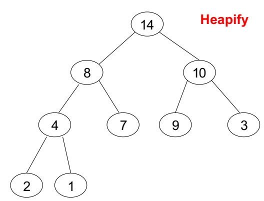
Return and remove the largest element in the set



30

ExtractMax

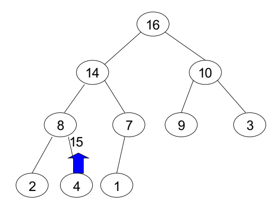
Return and remove the largest element in the set



31

IncreaseElement (aka swim up)

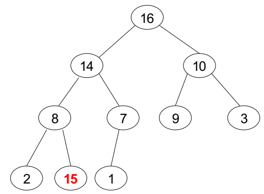
Increase the value of element *x* to *val*



34

IncreaseElement

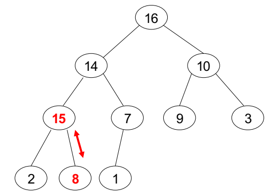
Increase the value of element *x* to *val*



35

IncreaseElement

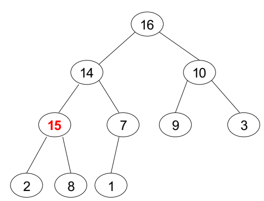
Increase the value of element *x* to *val*



36

IncreaseElement

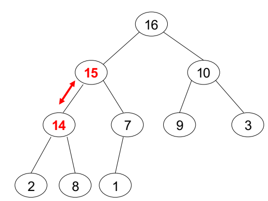
Increase the value of element *x* to *val*



37

IncreaseElement

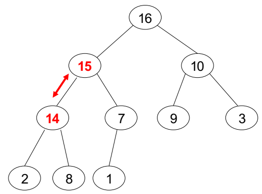
Increase the value of element *x* to *val*



38

IncreaseElement

Increase the value of element *x* to *val*



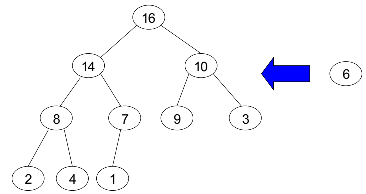
Runtime?

O(height of tree) = O(log n)

39

Insert

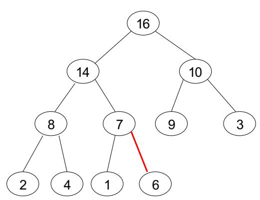
Insert *val* into the set



44

Insert

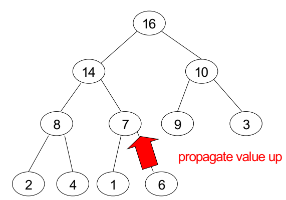
Insert *val* into the set



45

Insert

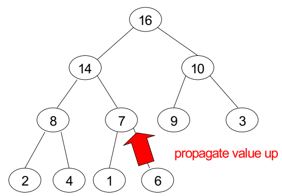
Insert *val* into the set



46

Insert

Insert *val* into the set



O(log n)

47

Building a heap

Can we build a heap using the functions we have so far?

- Max
- ExtractMax
- Insert(val)
- IncreaseElement(x, val)

50

Building a heap

For each element x in array:
    insert(x)

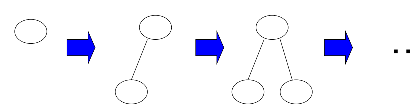
```
BUILD-HEAP1(A)
copy A to B
heap-size[A] ← 0
for i ← 1 to length[B]
        INSERT(A, B[i])
```

51

Running time of BuildHeap1

n calls to Insert – O(n log n)

Can we do better?



52

Building a heap: take 2

```
BUILD-HEAP2(A)
  heap-size[A] ← (length)[A]
  for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

Start with n/2 “one-node” heaps

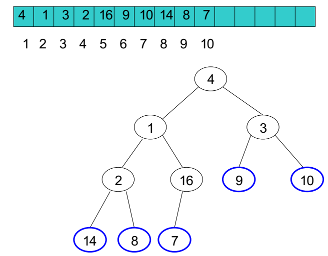
call Heapify on element n/2-1, n/2-2, n/2-3 …

all children have smaller indices

building from the bottom up, makes sure that all the children are heaps

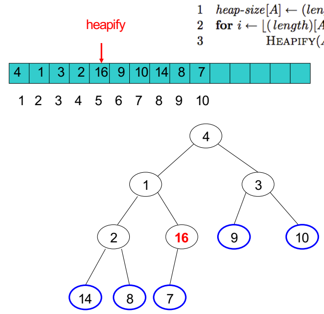
53


```
BUILD-HEAP2(A)
  heap-size[A] ← (length)[A]
  for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

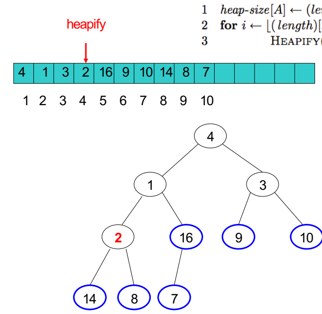
54


```
BUILD-HEAP2(A)
  heap-size[A] ← (length)[A]
  for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

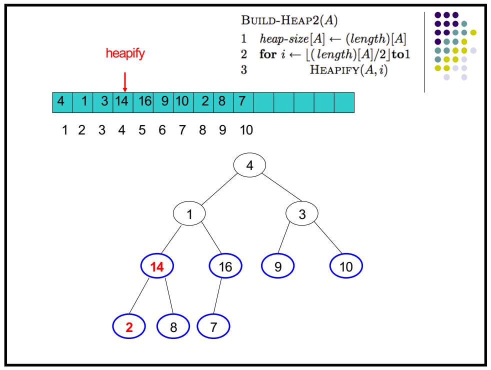



55

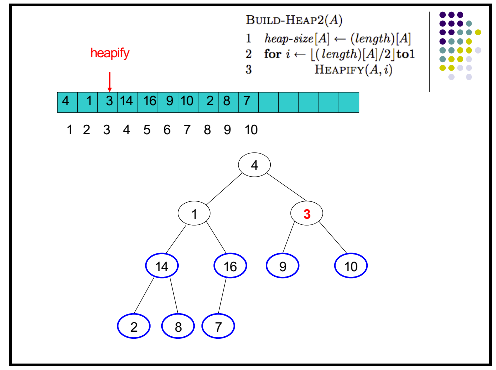

```
BUILD-HEAP2(A)
  heap-size[A] ← (length)[A]
  for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

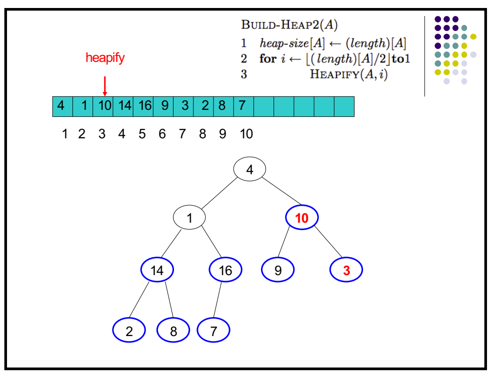
56


BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋to1
HEAPIFY(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10

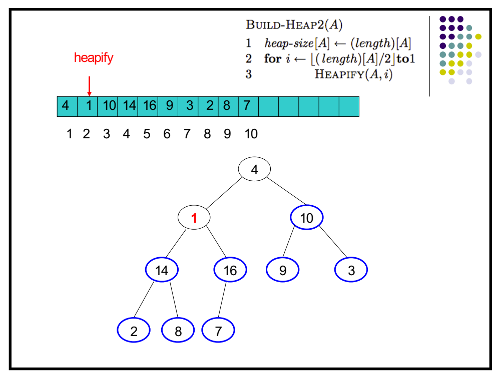
57


BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋to1
HEAPIFY(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10

58


BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋to1
HEAPIFY(A, i)
heapify
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10

59


BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋to1
HEAPIFY(A, i)
heapify
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10

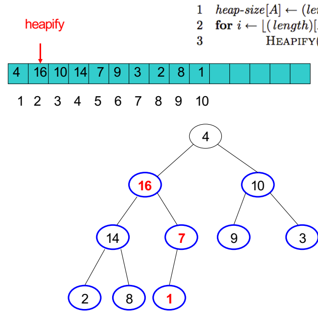
60

BUILD-HEAP2(A)

$heap\text{-}size[A] \leftarrow (length)[A]$

for $i \leftarrow \lfloor (length)[A]/2 \rfloor$ **to** 1

    HEAPIFY(A, i)



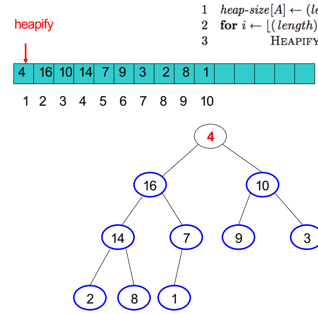
61

BUILD-HEAP2(A)

$heap\text{-}size[A] \leftarrow (length)[A]$

for $i \leftarrow \lfloor (length)[A]/2 \rfloor$ **to** 1

    HEAPIFY(A, i)



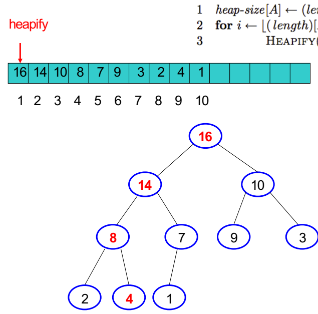
62

BUILD-HEAP2(A)

$heap\text{-}size[A] \leftarrow (length)[A]$

for $i \leftarrow \lfloor (length)[A]/2 \rfloor$ **to** 1

    HEAPIFY(A, i)



63

Running time of BuildHeap2

n/2 calls to Heapify – O(n log n)

Can we get a tighter bound?

BUILD-HEAP2(A)

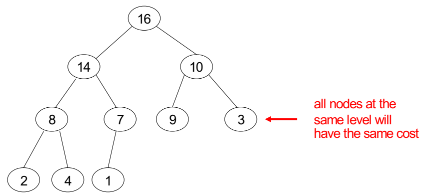
$heap\text{-}size[A] \leftarrow (length)[A]$

for $i \leftarrow \lfloor (length)[A]/2 \rfloor$ **to** 1

    HEAPIFY(A, i)

66

Running time of BuildHeap2



How many nodes are at level *d*? 2^d

67

Running time of BuildHeap2

$$T(n) = \sum_{d=0}^{\log n} 2^d O(d)$$

?

68

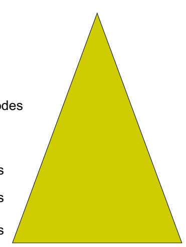
Nodes at height *h*

h < ceil(n/2^{h+1}) nodes

h=2 < ceil(n/8) nodes

h=1 < ceil(n/4) nodes

h=0 < ceil(n/2) nodes



69

Running time of BuildHeap2

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\log n} \left\lceil \frac{1}{2^{h+1}} \right\rceil h \right)$$

$$= O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= \theta(n)$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

70

Binary heaps

Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

74

Mergeable heaps

Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

- Mergeable heaps support the union operation
- Allows us to combine two heaps to get a single heap
- Union runtime for binary heaps?

75

Union for binary heaps

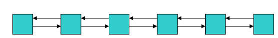
Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	$\Theta(n)$
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

concatenate the arrays and then call Build-Heap

76

Linked-list heap

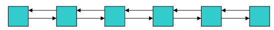


Store the elements in an unordered doubly linked list

- Insert:
- Max:
- Extract-Max:
- Increase:
- Union:

77

Linked-list heap



Store the elements in an unordered doubly linked list

- Insert: add to the end/beginning
- Max: search through the linked list
- Extract-Max: search and delete
- Increase: increase value
- Union: concatenate linked lists

78

Linked-list heap

Procedure	Binary heap (worst-case)	Linked-list
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$\Theta(n)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(n)$
UNION	$\Theta(n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(1)$

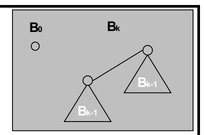
(adapted from Figure 19.1, pg. 456 [1])

Faster Union, Increase, Insert and Delete… but slower Max operations

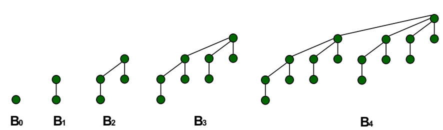
79

Adapted from:
Kevin Wayne

Binomial Tree



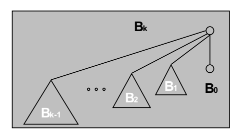
B_k: a binomial tree B_{k-1} with the addition of a left child with another binomial tree B_{k-1}



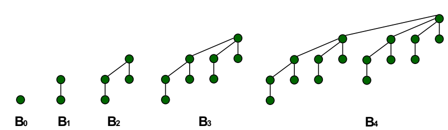
80

Binomial Tree

Number of nodes with respect to k?



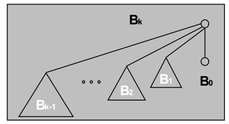
$N(B_o) = 1$
$N(B_k) = 2\ N(B_{k-1}) = 2^k$



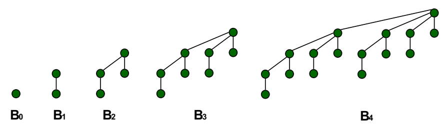
81

Binomial Tree

Height?



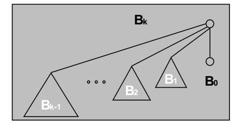
$H(B_0) = 1$
$H(B_k) = 1 + H(B_{k-1}) = k$



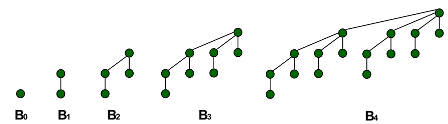
82

Binomial Tree

Degree of root node?



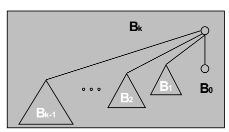
k, each time we add another binomial tree



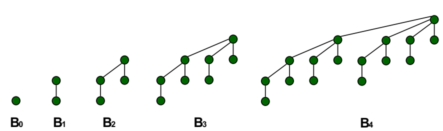
83

Binomial Tree

What are the children of the root?



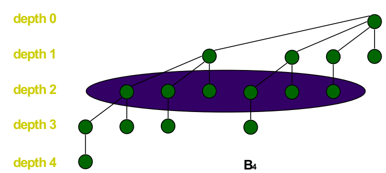
k binomial trees:
$B_{k-1}, B_{k-2}, \ldots, B_0$



84

Binomial Tree

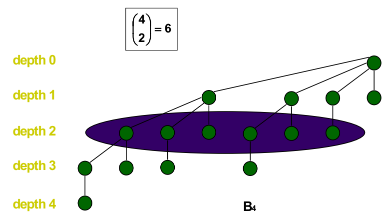
Why is it called a binomial tree?



85

Binomial Tree

B_k has $\binom{k}{i}$ nodes at depth i.



86

Another set data structure: recap

Idea: store data in a collection of arrays

- array *i* has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

87

Another set data structure: recap

Which arrays are full and empty are based on the number of elements

- specifically, binary representation of the number of elements
- 4 items = 100 = A2-full, A1-empty, A_0-empty
- 11 items = 1011 = A_3-full, A_2-empty, A_1-full, A_0-full

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

- Worst case runtime?

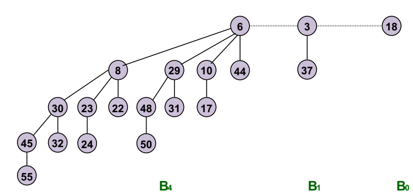
88

Binomial Heap

Binomial heap Vuillemin, 1978.

Sequence of binomial trees that satisfy binomial heap property:

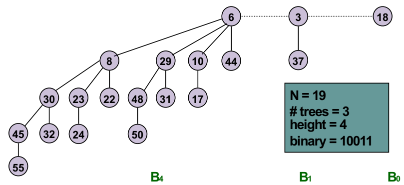
- each tree is min-heap ordered
- top level: full or empty binomial tree of order k
- which are empty or full is based on the number of elements



89

Binomial Heap

A_0: [18]
A_1: [3, 7]
A_2: empty
A_3: empty
A_4: [6, 8, 29, 10, 44, 30, 23, 22, 48, 31, 17, 45, 32, 24, 55]

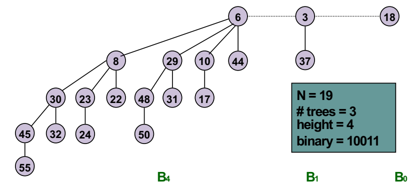


90

Binomial Heap: Properties

How many heaps?

O(log n) – binary number representation

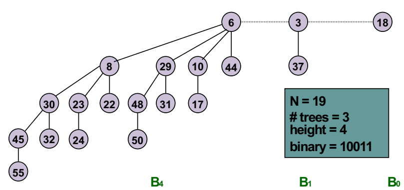


91

Binomial Heap: Properties

Where is the max/min?

Must be one of the
roots of the heaps

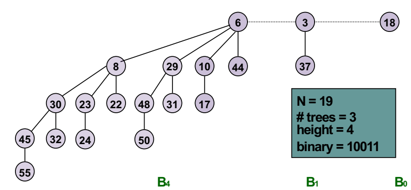


92

Binomial Heap: Properties

Runtime of max/min?

O(log n)



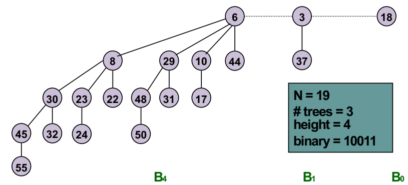
93

Binomial Heap: Properties

Height?

$\log_2 n$

- largest tree = $B_{\log n}$
- height of that tree is log n



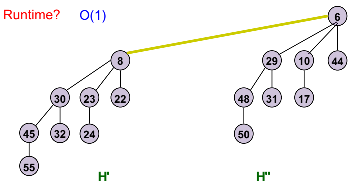
94

Binomial Heap: Union

How can we merge two binomial tree heaps of the same size (2^k)?

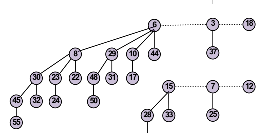
- connect roots of H' and H''
- choose smaller key to be root of H

Runtime? O(1)



95

Binomial Heap: Union

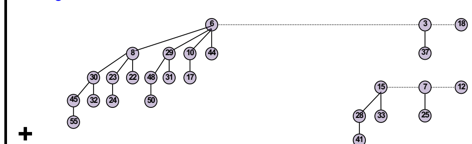


How can we combine/merge binomial heaps (i.e. a combination of binomial tree heaps)?

96

Binomial Heap: Union

Go through each tree size starting at 0 and merge as we go

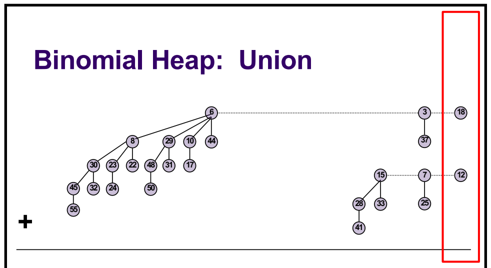


19 + 7 = 26

		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

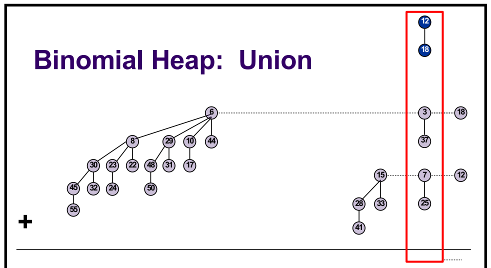
97

Binomial Heap: Union

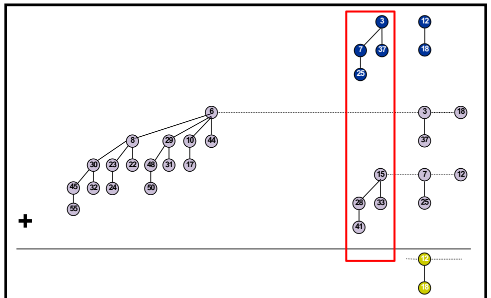


98

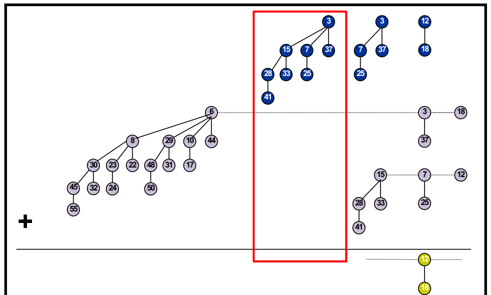
Binomial Heap: Union



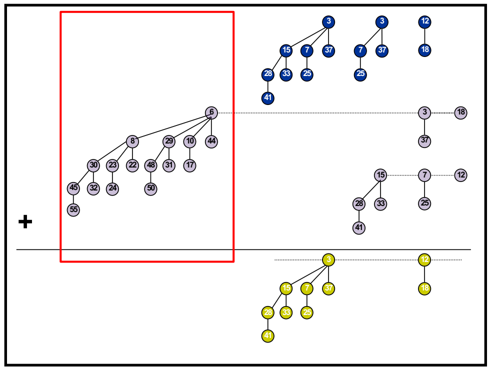
99



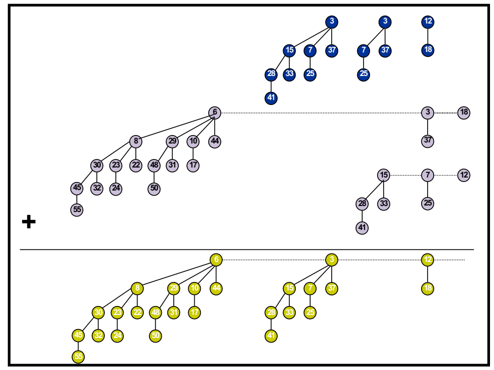
100



101



102



103

Binomial Heap: Union

Analogous to binary addition

Running time?

- Proportional to number of trees in root lists 2 O($\log_2$ N)
- O(log N)

19 + 7 = 26

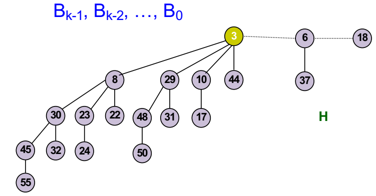
		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

104

Binomial Heap: Delete Min/Max

We can find the min/max in O(log n).
How can we extract it?

Hint: B_k consists of binomial trees:
B_{k-1}, B_{k-2}, …, B_0

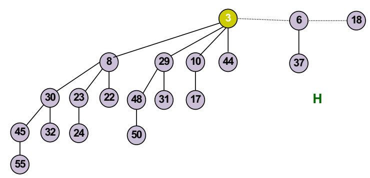


105

Binomial Heap: Delete Min

Delete node with minimum key in binomial heap H.

- Find root x with min key in root list of H, and delete
- H' ← broken binomial trees
- H ← Union(H', H)



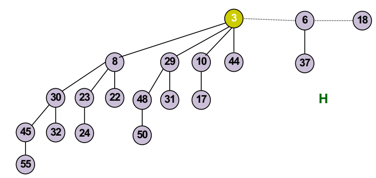
106

Binomial Heap: Delete Min

Delete node with minimum key in binomial heap H.

- Find root x with min key in root list of H, and delete
- H' ← broken binomial trees
- H ← Union(H', H)

Running time? O(log N)



107

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$
MAXIMUM	$\Theta(1)$	$O(\log n)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

114

Fibonacci Heaps

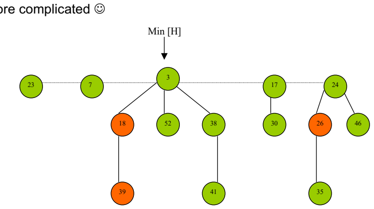
Similar to binomial heap

- A Fibonacci heap consists of a sequence of heaps

More flexible

- Heaps do not have to be binomial trees

More complicated ☺



115

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

Should you always use a Fibonacci heap?

116

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

- Extract-Max and Delete are O(n) worst case
- Constants can be large on some of the operations
- Complicated to implement

117

Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

Can we do better?

118