Week 5
Priority Queues
Maps and Sets

Algorithms and Data Structures

COMP3506/7505

Week 5 – Priority Queues & Maps

1. Priority queues
2. Heaps
3. Adaptable priority queues
4. Maps
5. Sets

Priority Queue ADT

- Each entry is a pair (key, value)
- Main methods
  - insert(k, v)
  - removeMin()
- Additional methods
  - min()
  - size(), isEmpty()

Queue vs. Priority Queue

- Queue
  - First in, first out (FIFO)
  - Elements are added at end of queue
- Priority Queue
  - Elements are stored as Entries
  - Entry with highest priority is removed first
    - lowest key

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }

Example

- Sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5, A)	–	{ (5,A) }
insert(9, C)	–	{ (5,A), (9,C) }
insert(3, B)	–	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7, D)	–	{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Entry ADT: "Maps" key to value

- An entry in a priority queue is simply a key-value pair
- Methods
  - getKey
  - getValue

```
public class Entry implements Comparable<Entry> {
    private int key;
    private String value;

    public Entry(int k, String v) {
        this.key = k;
        this.value = v;
    }

    @Override
    public int compareTo(Entry other) {
        return Integer.compare(this.key, other.key);
    }

    public int getKey() { return key; }
    public String getValue() { return value; }
}
```

Entry ADT: "Maps" key to value

- An entry in a priority queue is simply a key-value pair
- Methods
  - getKey
  - getValue

```
public class Entry implements Comparable<Entry> {
    private int key;
    private String value;

    public Entry(int k, String v) {
        this.key = k;
        this.value = v;
    }
```

```
try other) {
are(this.key, other.key);
```

```
eturn key; }
public String getValue() { return value; }
}
```

```
Entry e1 = new Entry(5, "Five");
Entry e2 = new Entry(10, "Ten");

if (e1.compareTo(e2) < 0) {
    System.out.println("e1 key smaller than e2 key");
}
```

Total Order Relations

- Mathematical concept of total order relation $\leq$
  - Comparability property
    - either $x \leq y$ or $y \leq x$
  - Antisymmetric property
    - $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property
    - $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Note: Imposing a total order on your keys means that trees are representing a kind of *ordered map*

Note, however, that true "maps" require key uniqueness – we **do not** enforce that here for trees

Comparator ADT

- Generic priority queue uses an auxiliary comparator
- compare(a, b) – returns an integer i such that
  - $i < 0$ if a < b,
  - $i = 0$ if a = b
  - $i > 0$ if a > b
  - error occurs if a and b cannot be compared

Example Comparator

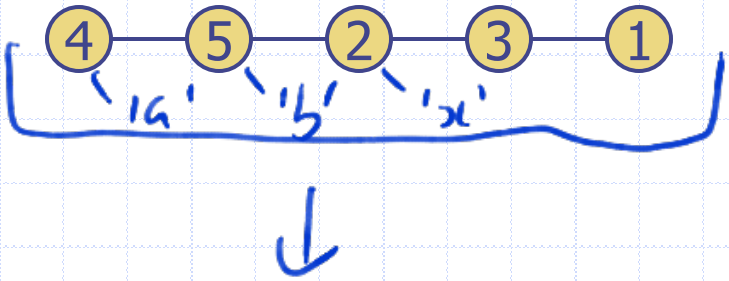
```
public class Point implements Comparable<Point> {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public int compareTo(Point other) {
        if (this.x == other.x) {
            return Integer.compare(this.y, other.y);
        } else {
            return Integer.compare(this.x, other.x);
        }
    }
}
```

Sequence-Based Priority Queue

- Implementation with an unsorted list



Sequence-Based Priority Queue

- Implementation with an unsorted list

- Performance
  - insert takes $O(1)$ time
  - removeMin and min take $O(n)$ time

Sequence-Based Priority Queue

- Implementation with an unsorted list

- Performance
  - insert takes $O(1)$ time
  - removeMin and min take $O(n)$ time

- Implementation with a sorted list

Sequence-Based Priority Queue

- Implementation with an unsorted list

- Performance
  - insert takes $O(1)$ time
  - removeMin and min take $O(n)$ time

- Implementation with a sorted list

- Performance
  - insert takes $O(n)$ time
  - removeMin and min take $O(1)$ time

Sequence-Based Priority Queue

- Implementation with an unsorted list

- Performance
  - insert takes $O(1)$ time
  - removeMin and min take $O(n)$ time

- Implementation with a sorted list

- Performance
  - insert takes $O(n)$ time
  - removeMin and min take $O(1)$ time

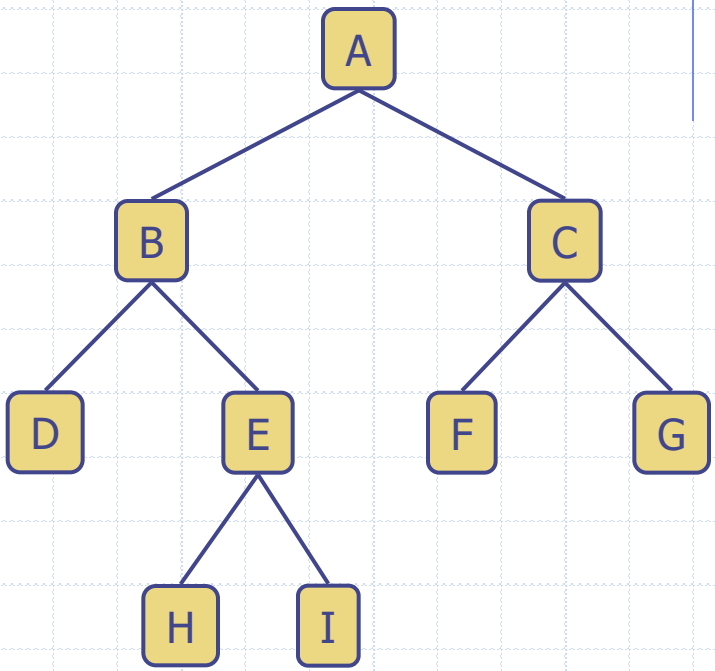
Think: Do these performance characteristics change if we use a linked list vs a (dynamic) array?

Week 5 – Priority Queues and Maps

1. Priority queues
2. Heaps
3. Adaptable priority queues
4. Maps
5. Sets

Recall Binary Trees

- Each internal node has at most two children
  - exactly two for **proper** binary trees
- Node's children are an ordered pair
- Internal nodes have a left and right child



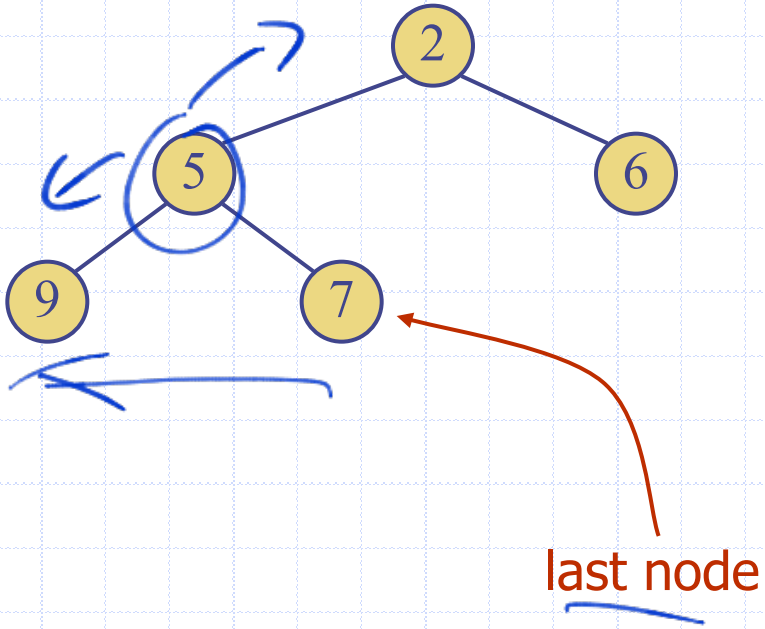
Heaps

Min

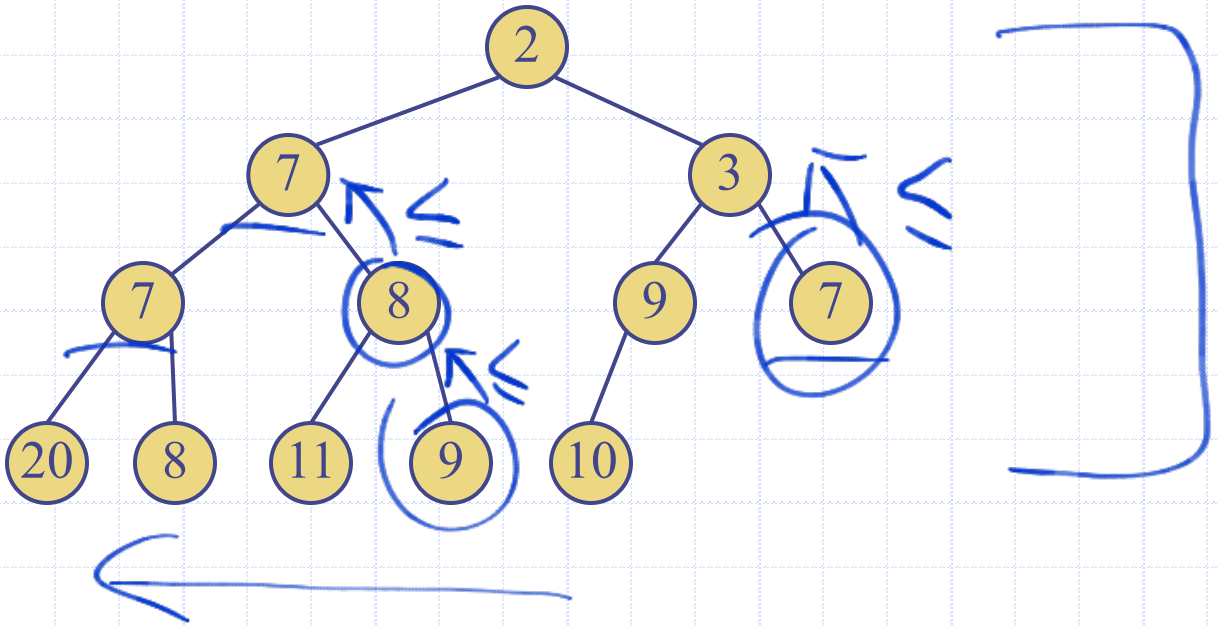
Max

- Binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every node v other than the root
  - $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
  - at height h, the internal nodes are to the left of the external nodes

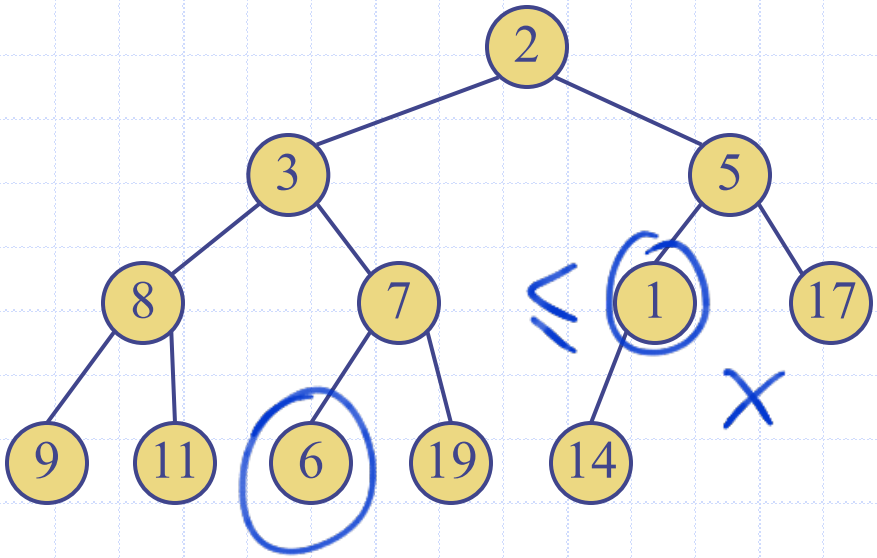
- Last node of a heap is the rightmost node of maximum depth



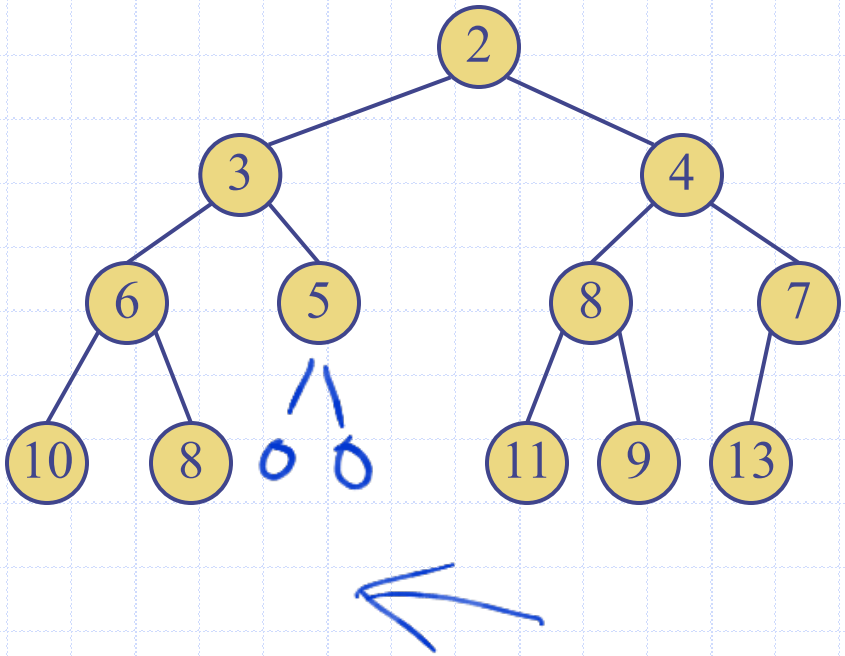
Is this a (min) heap?



Is this a min heap?



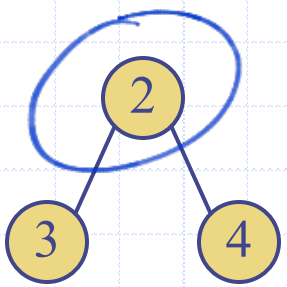
Is this a min heap?

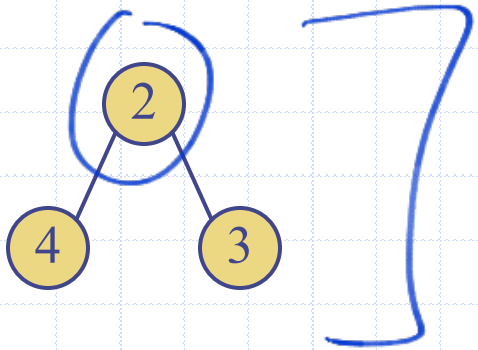


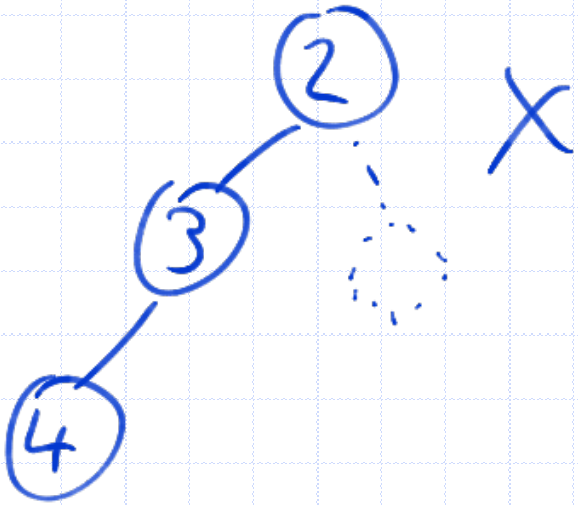
Complete BT

Heap order

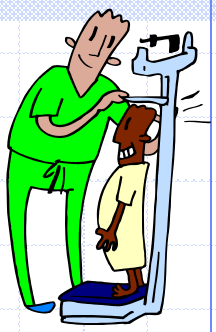
How many heaps are there containing the keys 2, 3, 4?



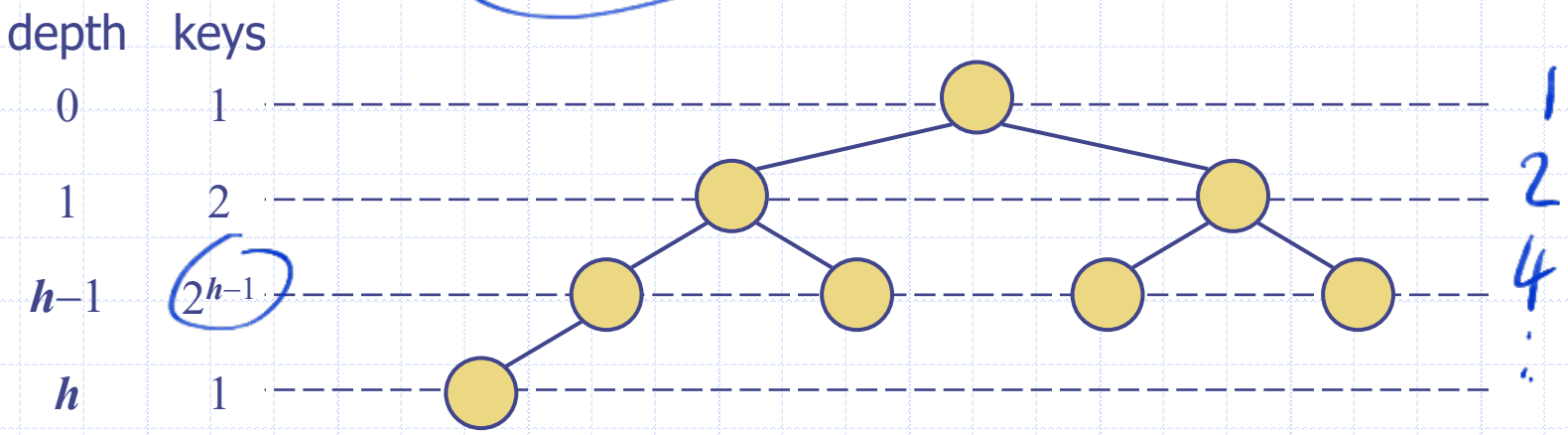




Height of a Heap

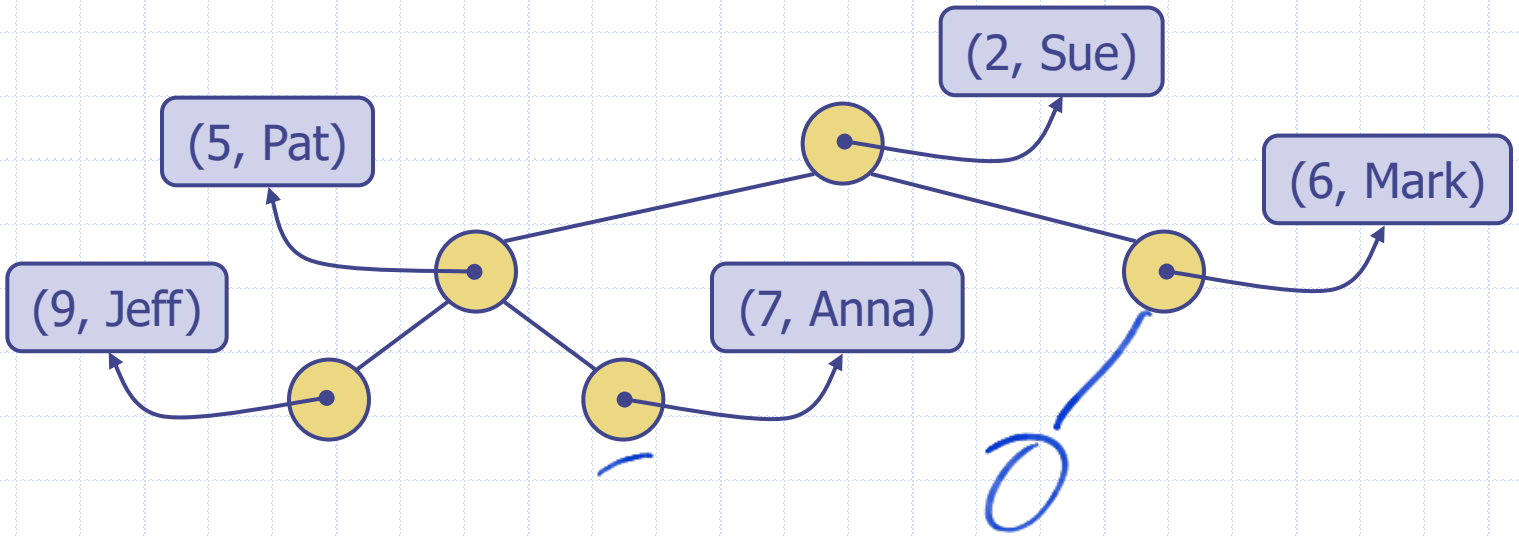


- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1$
  - Thus, $n \geq 2^h$, i.e. $h \leq \log n$



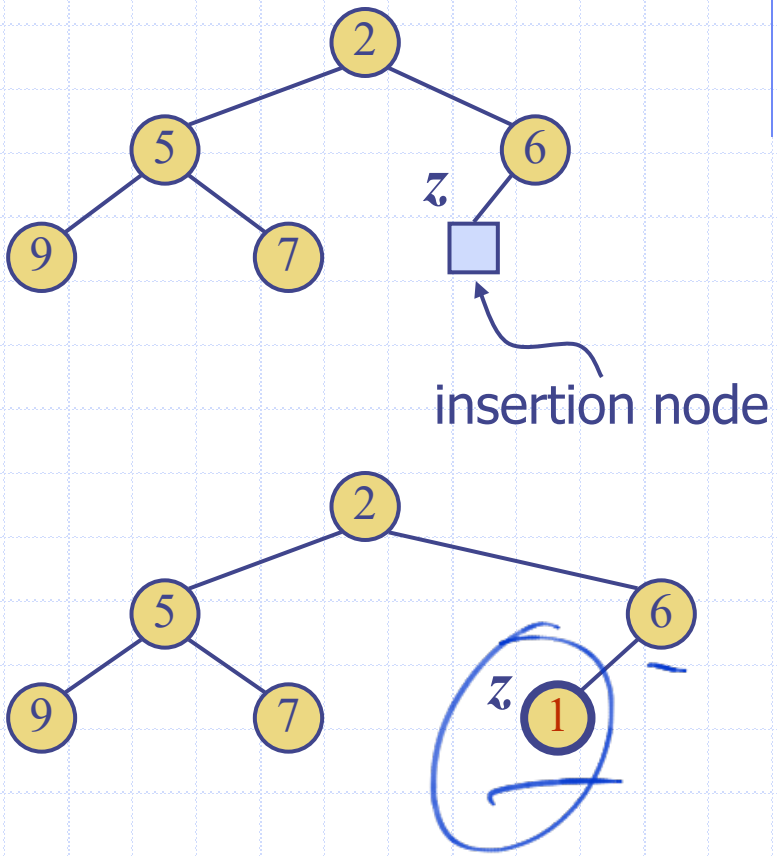
Heaps and Priority Queues

- A heap can implement a priority queue
- Store a (key, element) entry at each node
- Keep track of the position of the last node



Insertion into a Heap

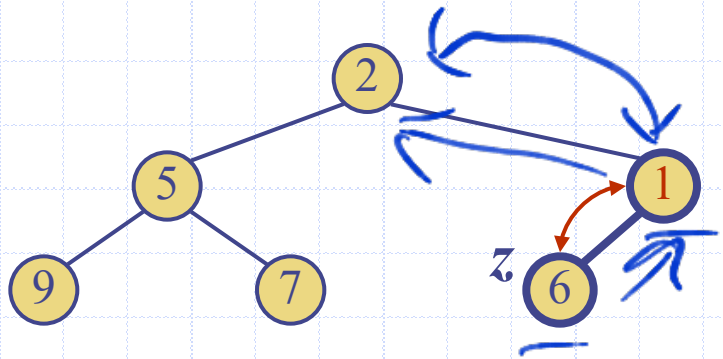
- insert in the PQ ADT corresponds to inserting key k into the heap
- Insertion algorithm
  1. Find the insertion node z
     - new last node
  2. Store k at z
  3. Restore the heap-order property
     - discussed next

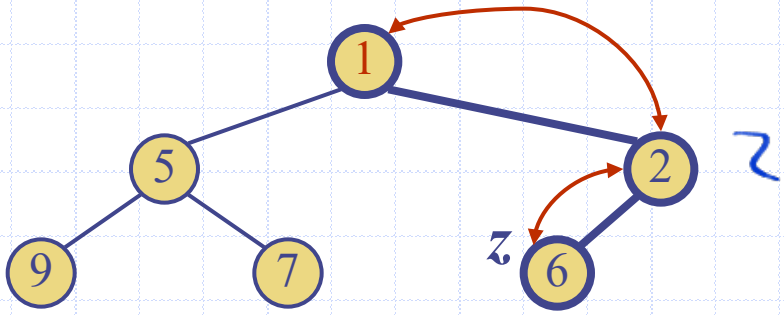


Upheap

AKA Bubbling Up

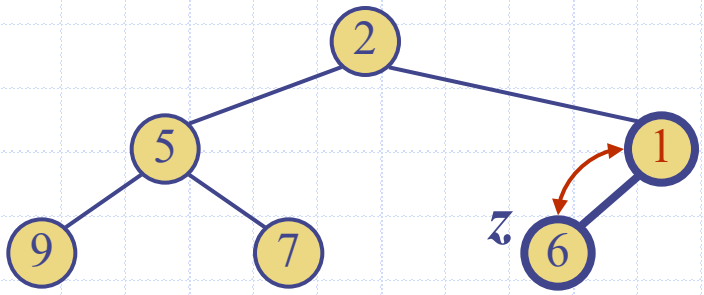
- After inserting a new key k, the heap-order property may be violated
- Think: What would the runtime of upheap be?
  - Hint: What do we know about the tree?

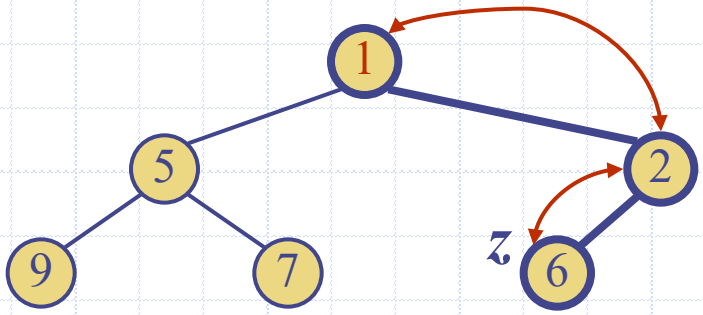




Upheap

- After inserting a new key k, the heap-order property may be violated
- Think: What would the runtime of upheap be?
  - Hint: What do we know about the tree?
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time (worst case)

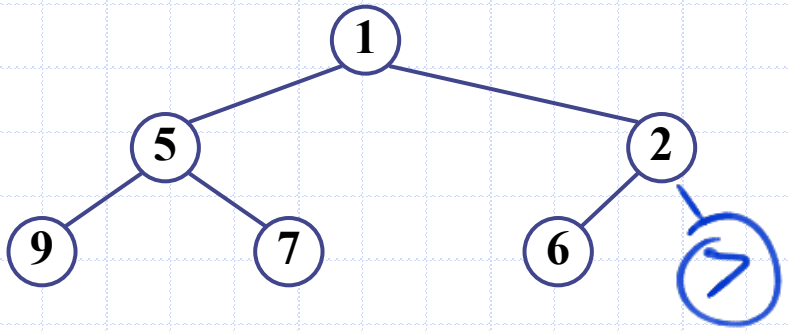




Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

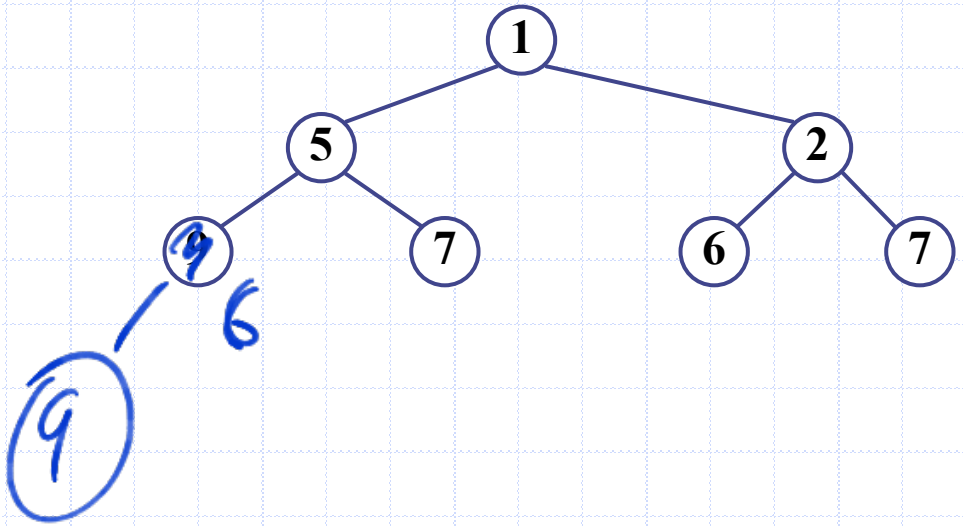
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

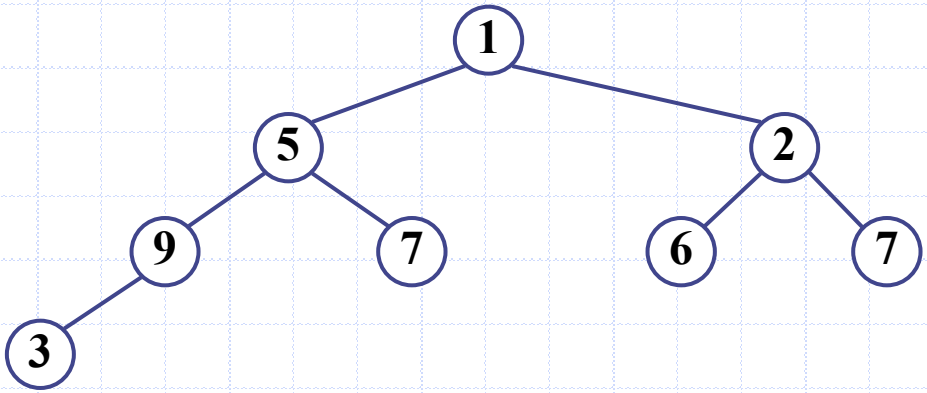
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

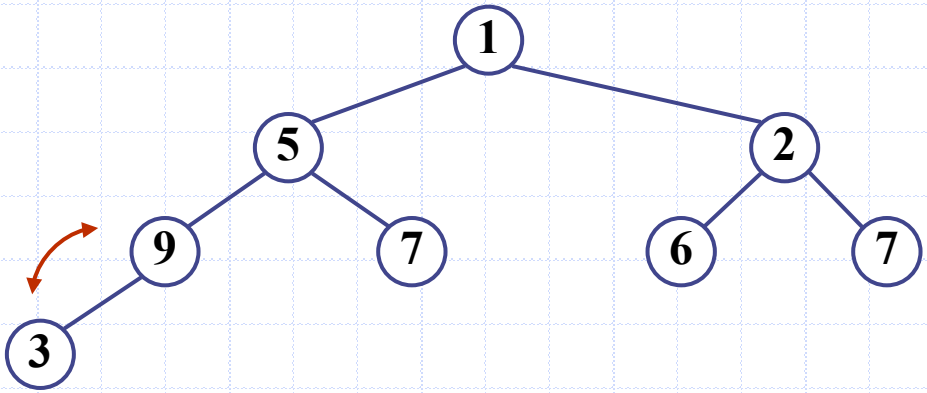
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

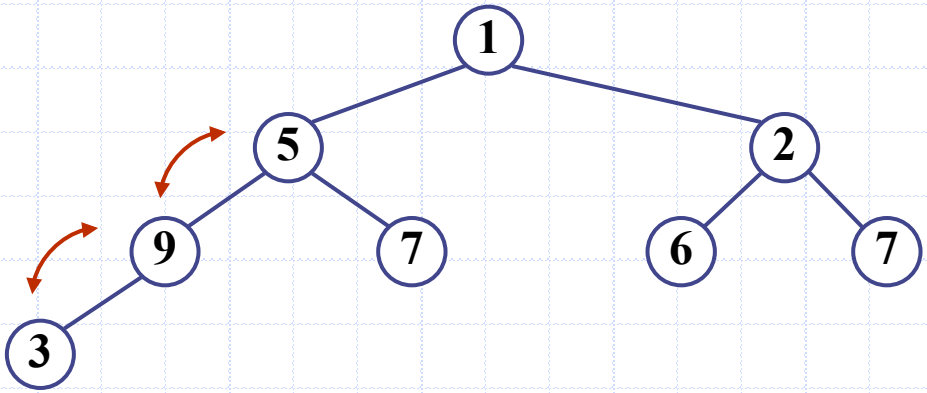
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

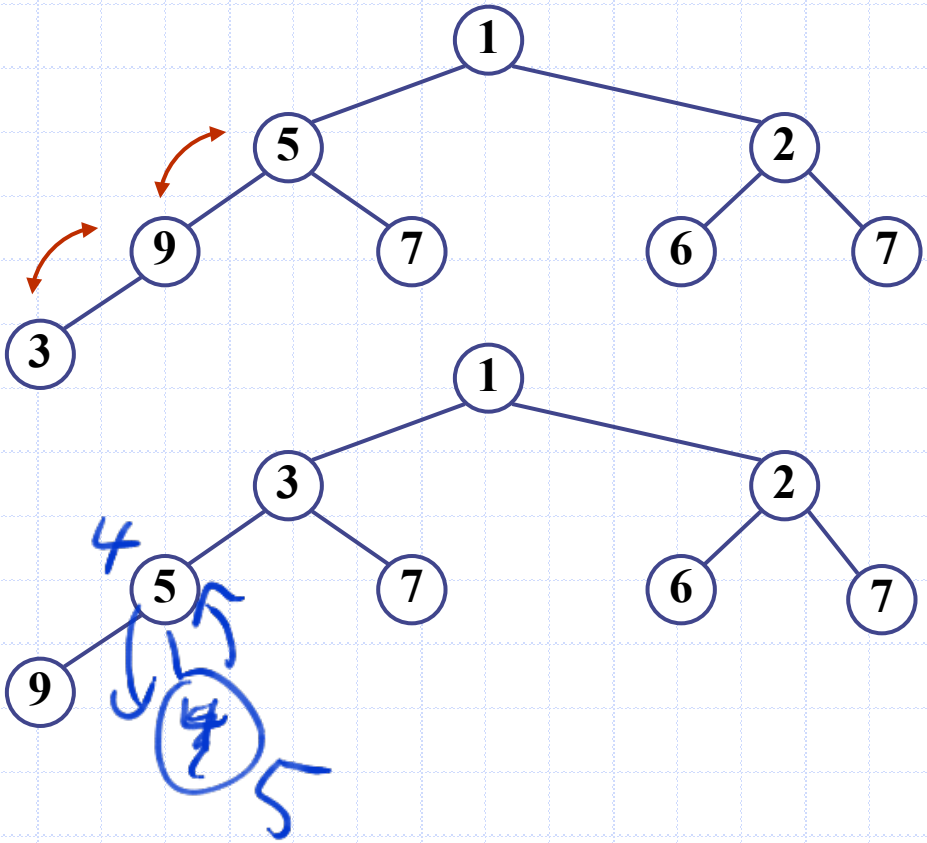
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

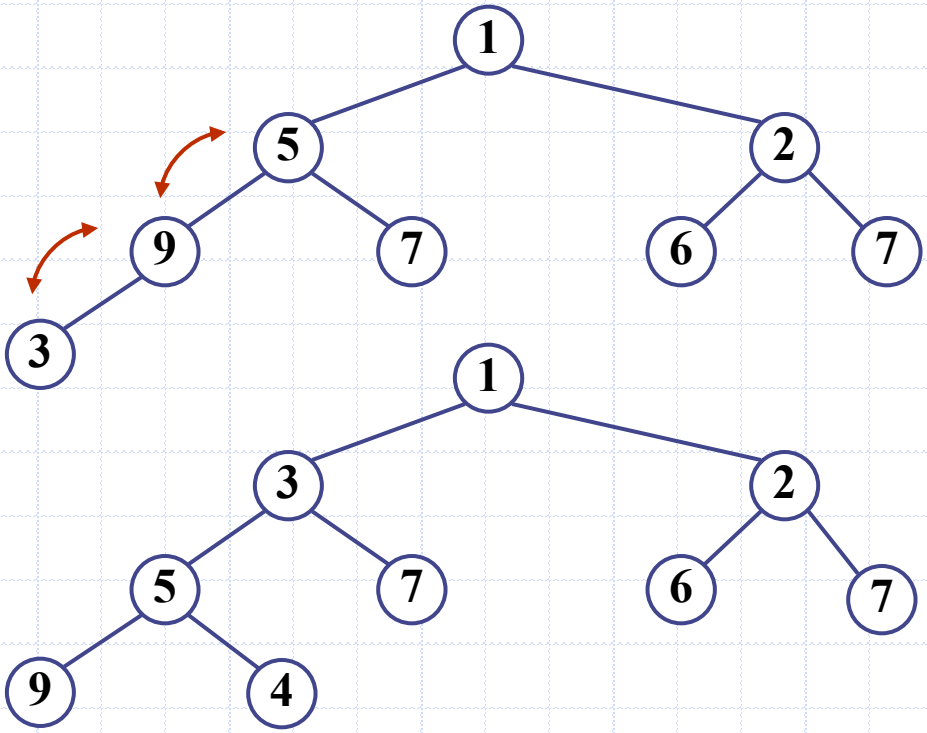
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

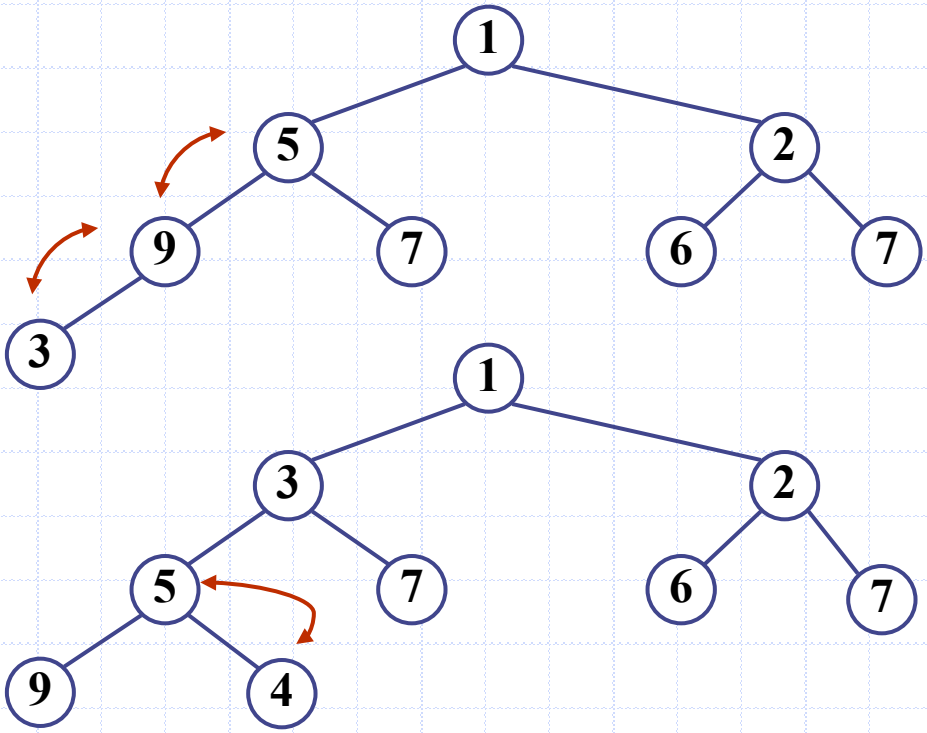
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

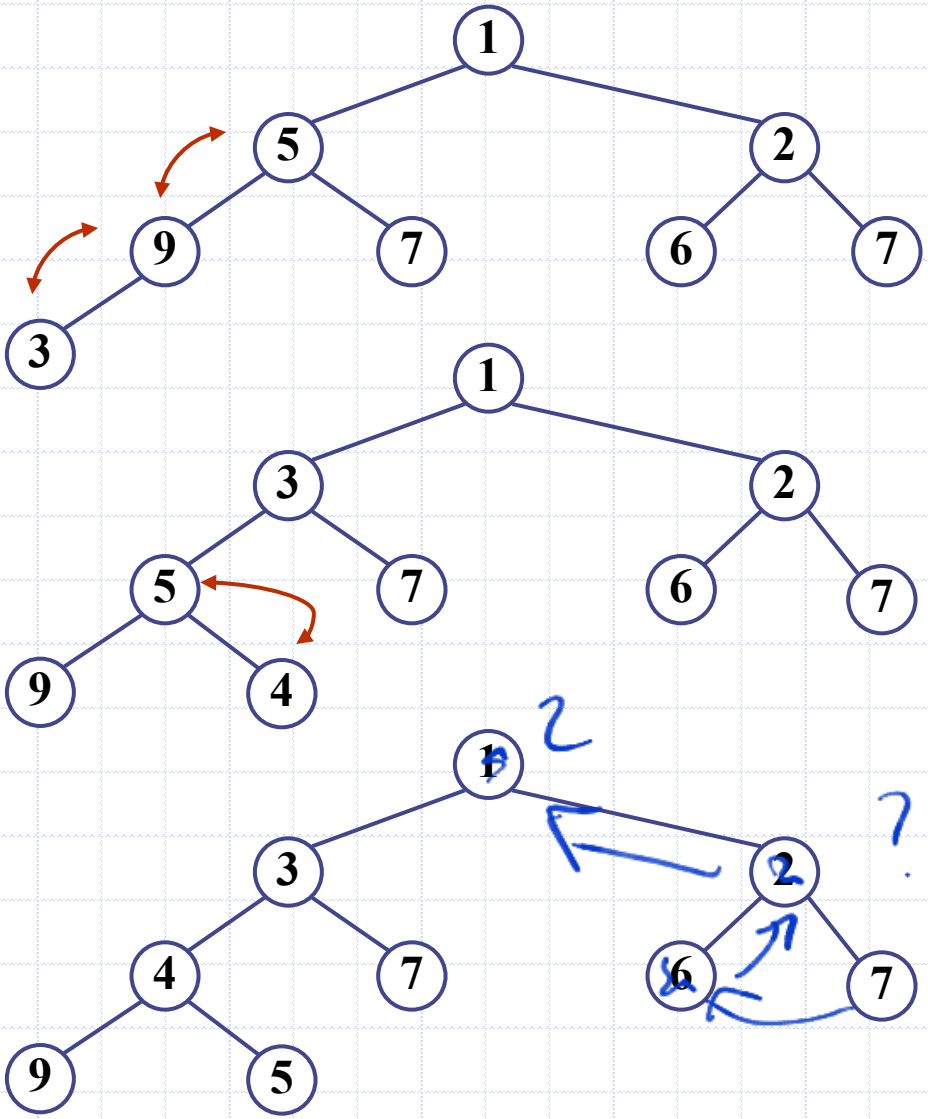
Add entries with the keys 7, 3, and 4



Example: Insertion and Upheap Bubbling

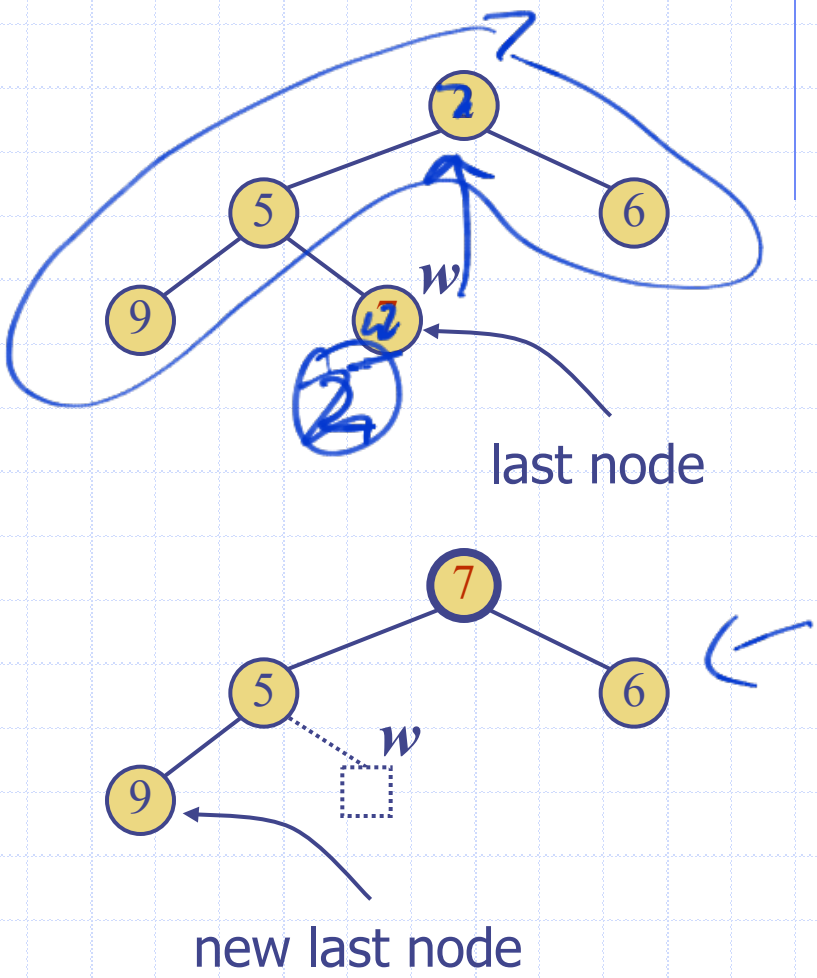
Upheap swaps k along an upward path from the insertion node, until k reaches the root or a node whose parent has a key less than or equal to k

Add entries with the keys 7, 3, and 4



Removal from a Heap

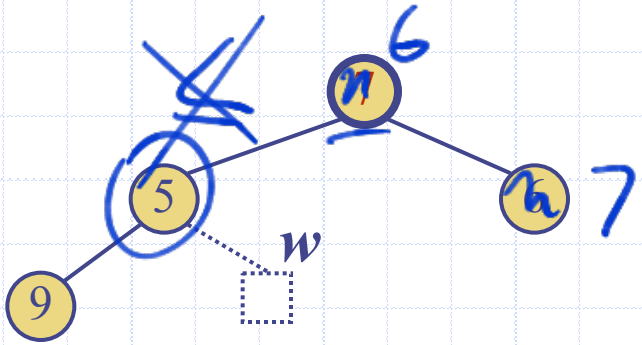
- removeMin in the PQ ADT corresponds to removing the root key from the heap
- Removal algorithm
  1. Replace the root key with the key of the last node w
  2. Remove w
  3. Restore the heap-order property
     - discussed next

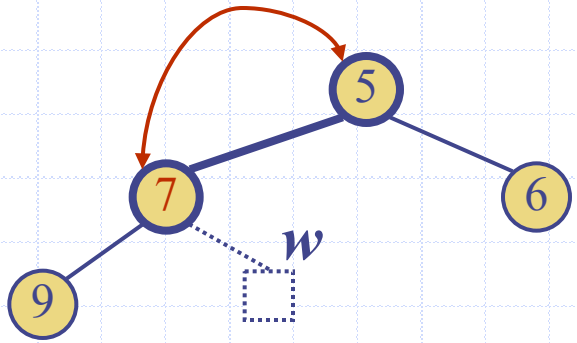


Downheap

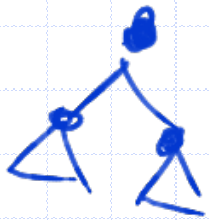
Bubbling down

- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



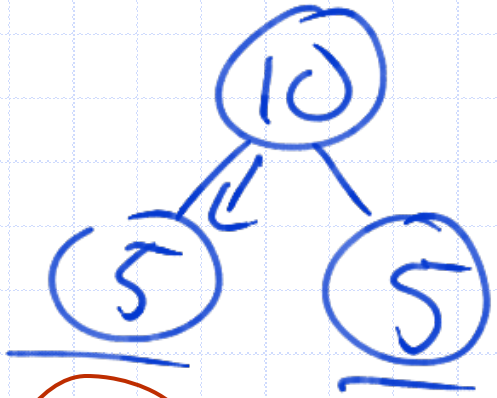


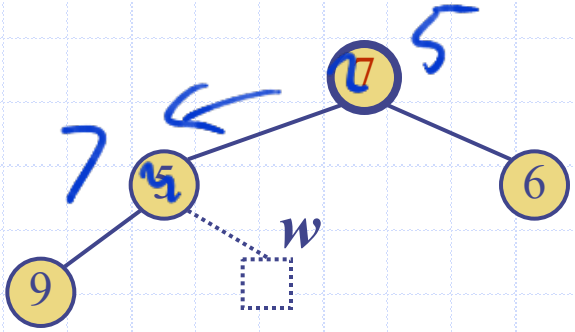
Downheap – Which Child?

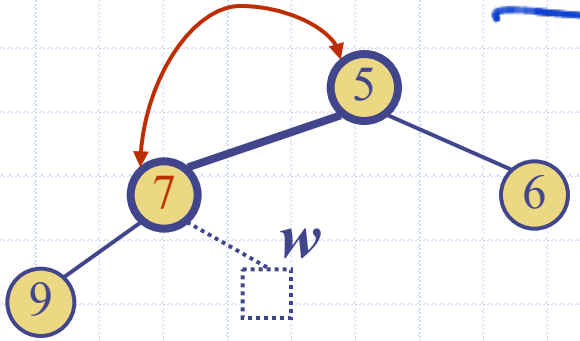


- If a node has no right child, choose the left child
- If a node has both children, choose the one with the smallest key

> What if the keys are equal?

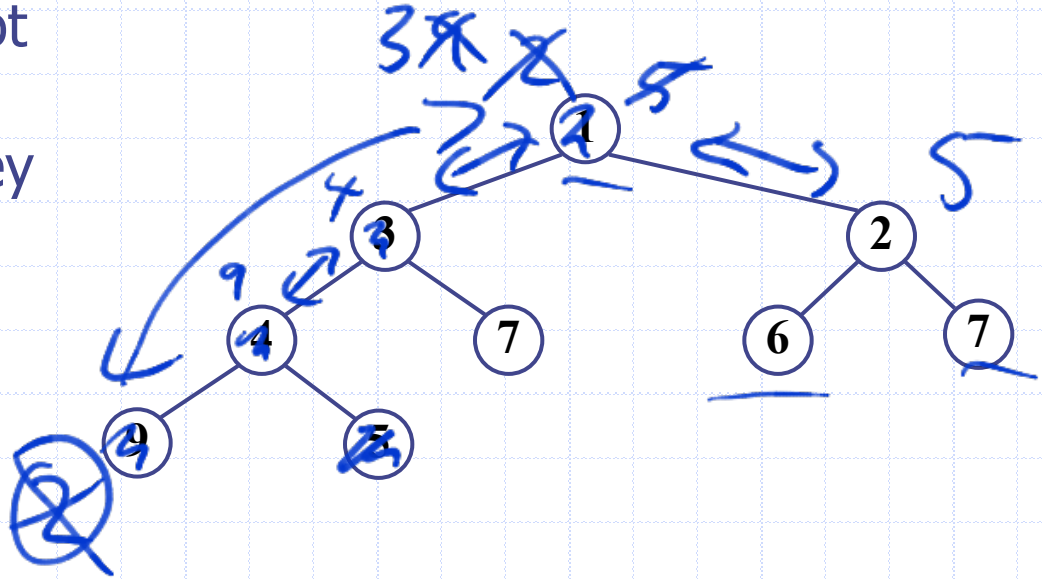






Example: Deletion and Downheap Bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

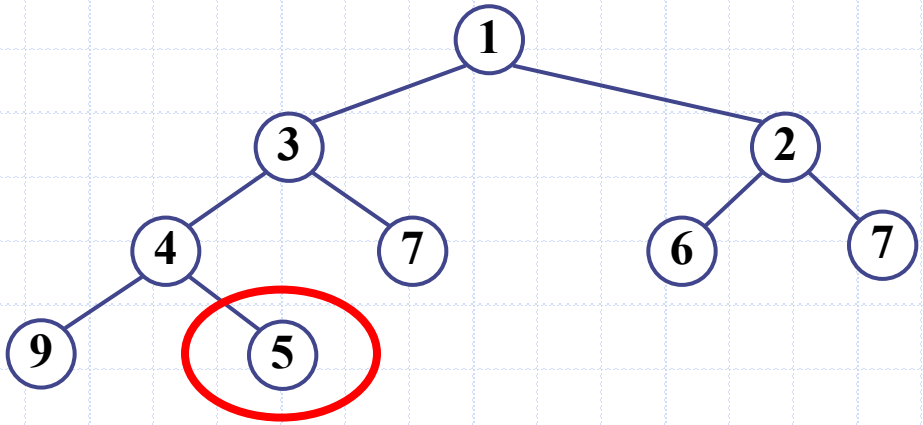


Delete two items from the heap:

1, 2

Example: Deletion and Downheap Bubbling

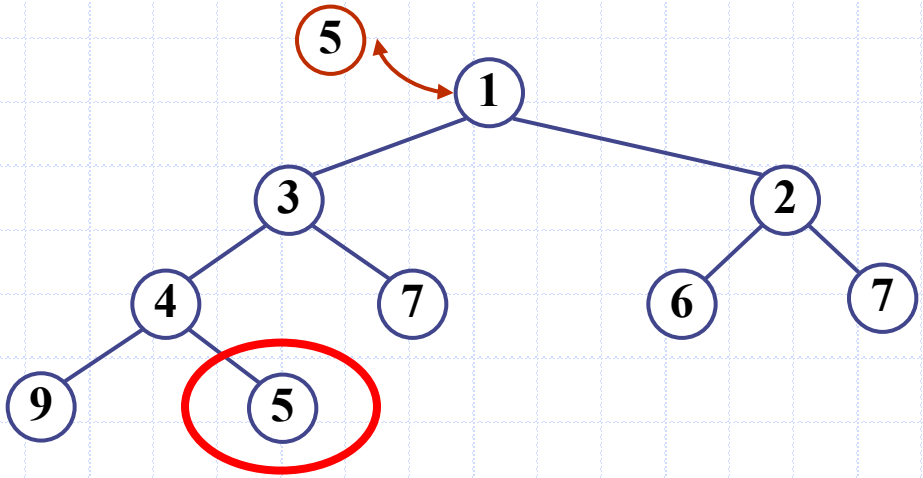
Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root



Delete two items from the heap:
1

Example: Deletion and Downheap Bubbling

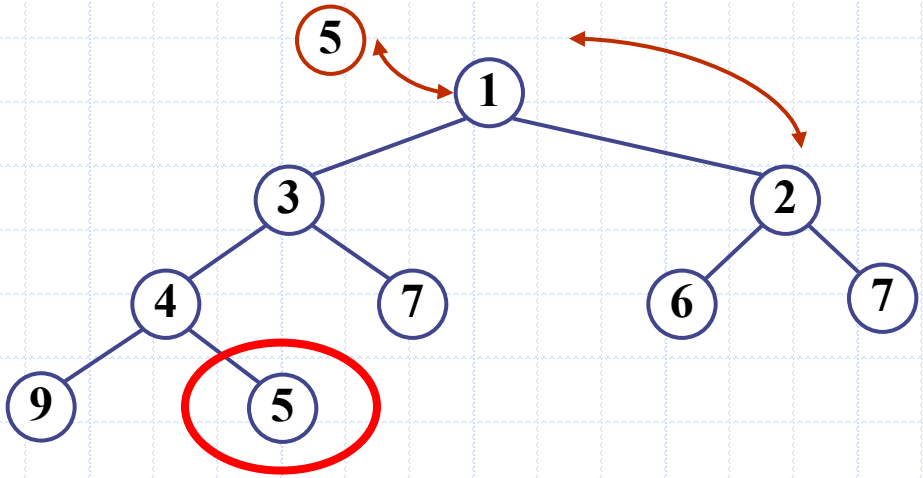
Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root



Delete two items from the heap:
1

Example: Deletion and Downheap Bubbling

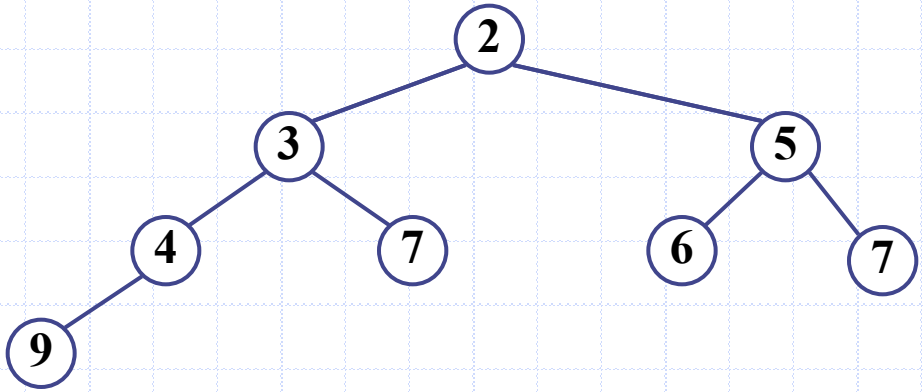
Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root



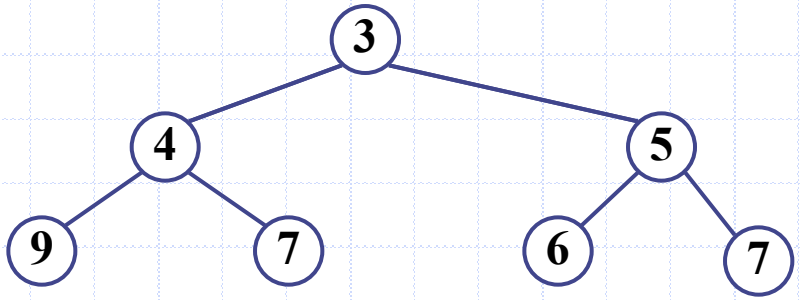
Delete two items from the heap:
1

Example: Deletion and downheap bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

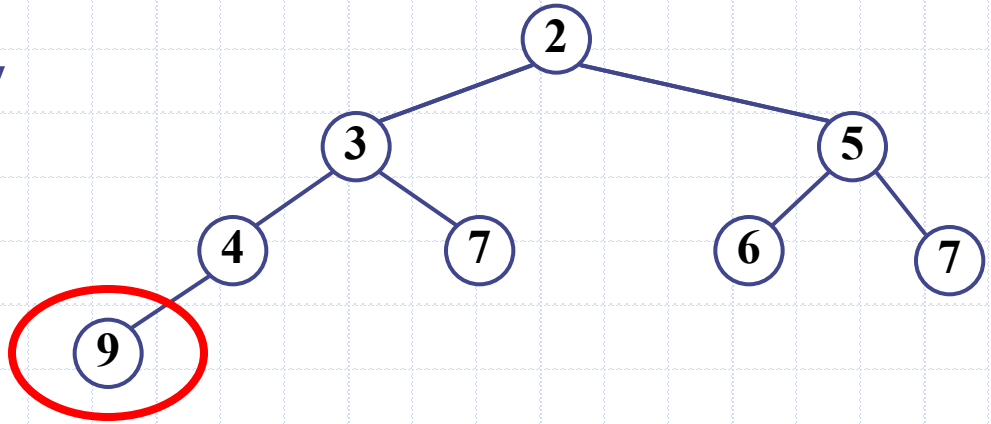


Delete two items from the heap: 1, 2

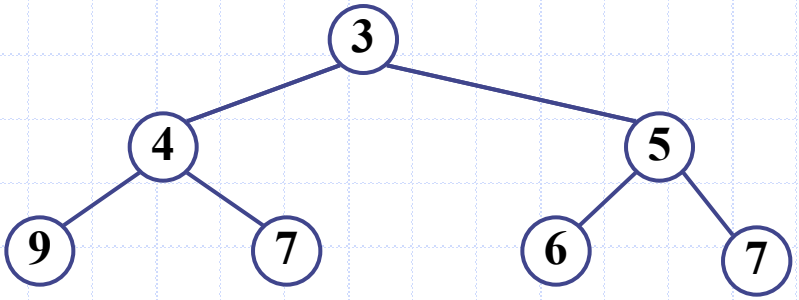


Example: Deletion and downheap bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

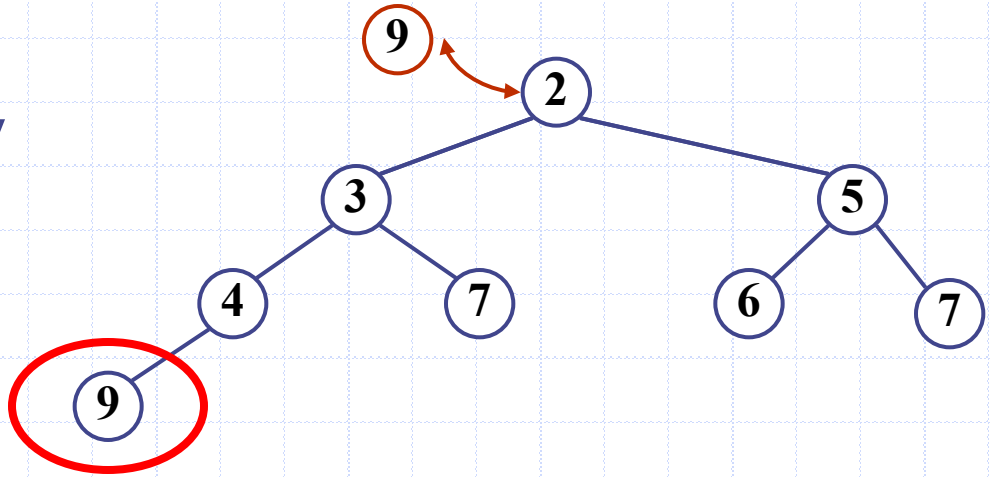


Delete two items from the heap: 1, 2

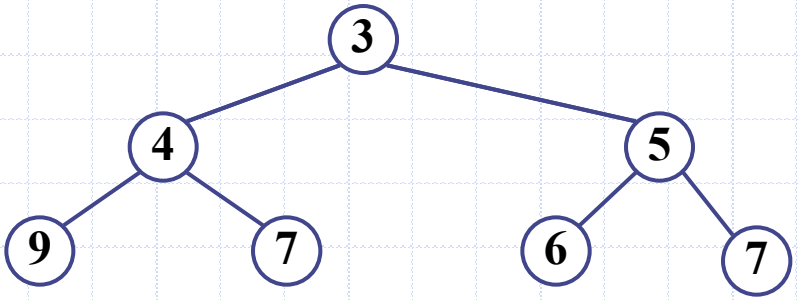


Example: Deletion and downheap bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

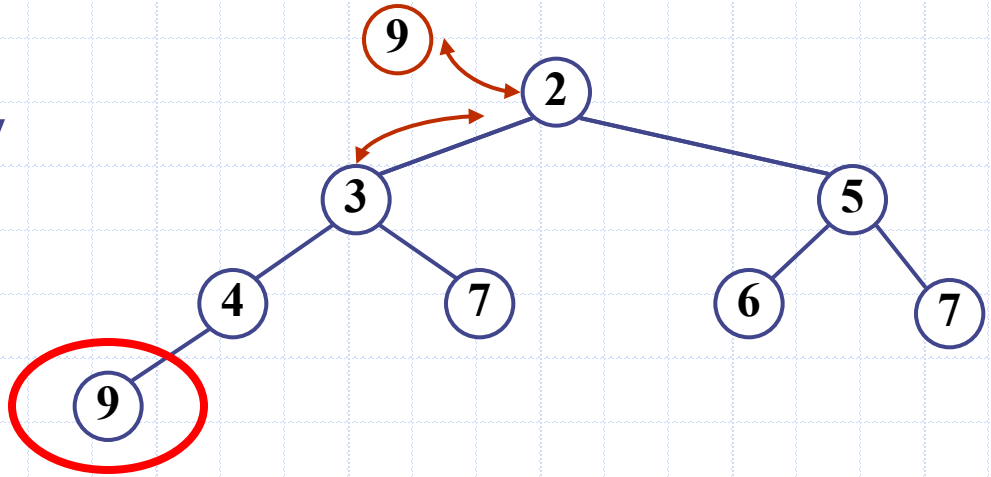


Delete two items from the heap: 1, 2

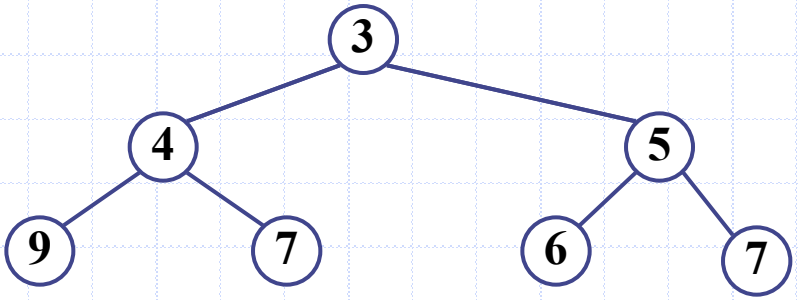


Example: Deletion and downheap bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

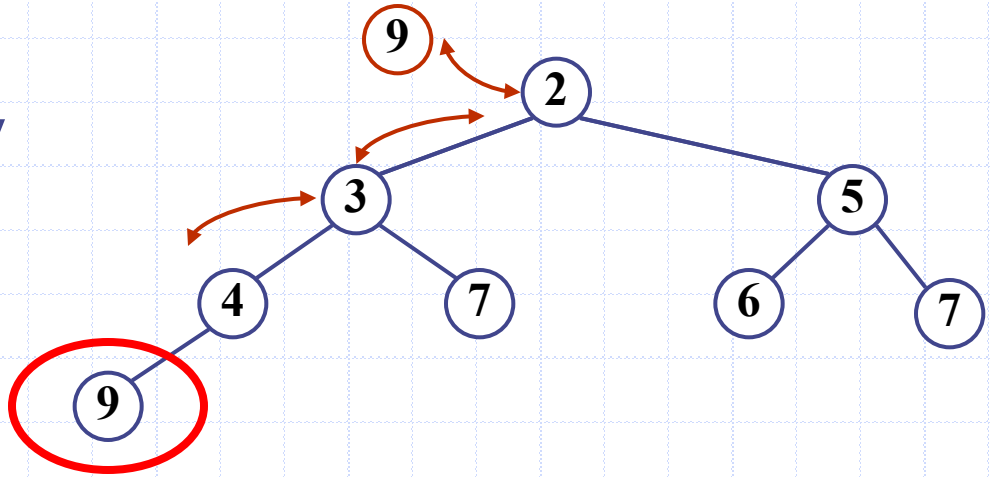


Delete two items from the heap: 1, 2

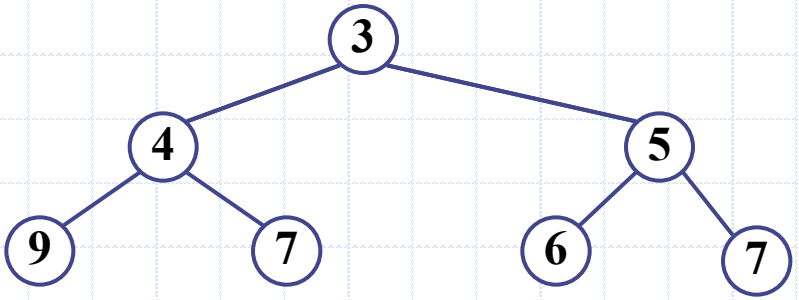


Example: Deletion and downheap bubbling

Downheap replaces root key with key k of the last node and swaps key k along a downward path from the root

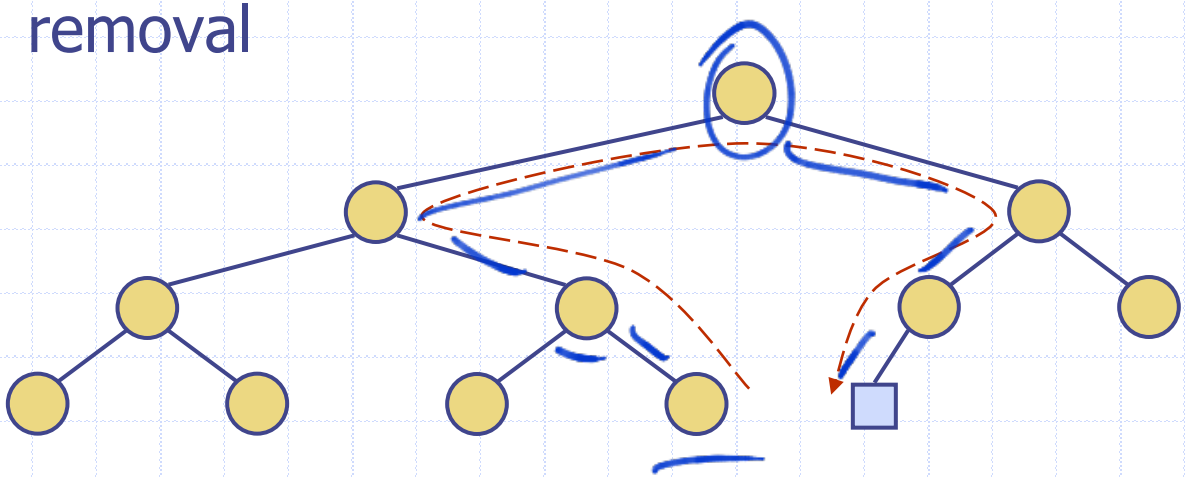


Delete two items from the heap: 1, 2

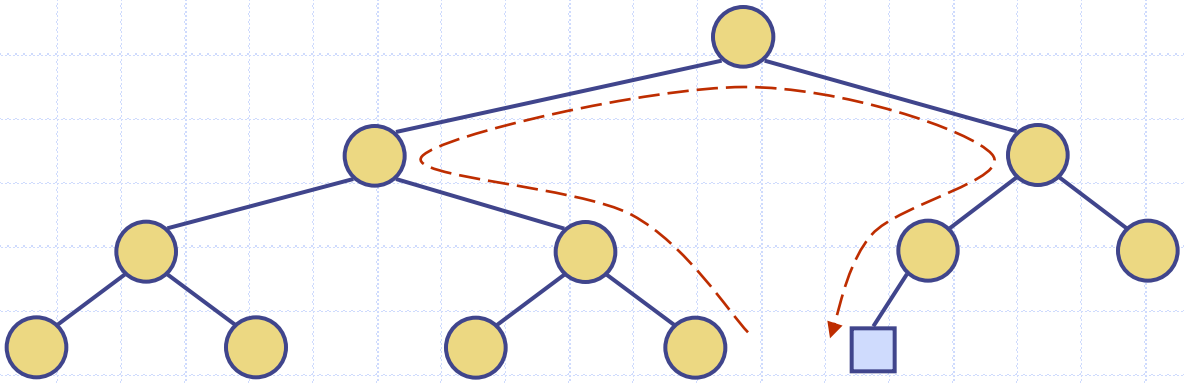


Updating the Last Node

- Insertion node can be found by traversing a path of $O(\log n)$ nodes
- Similar algorithm for updating the last node after a removal

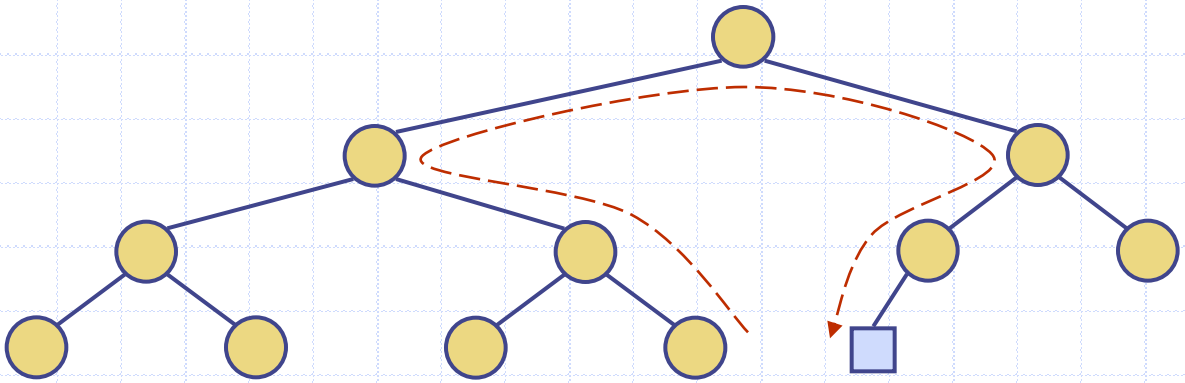


Can we find the last node faster than O(log *n*)?



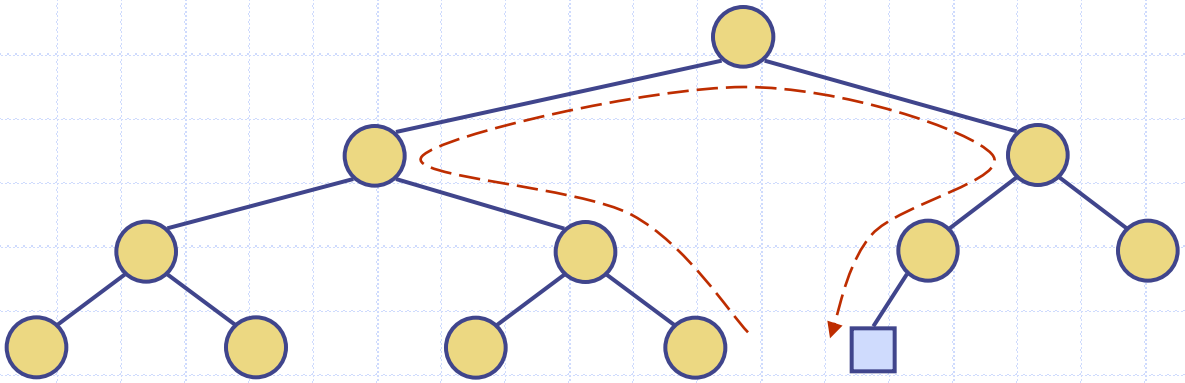
Can we find the last node faster than O(log *n*)?

- It depends on how the complete binary tree is implemented
  - Tree with augmentation?



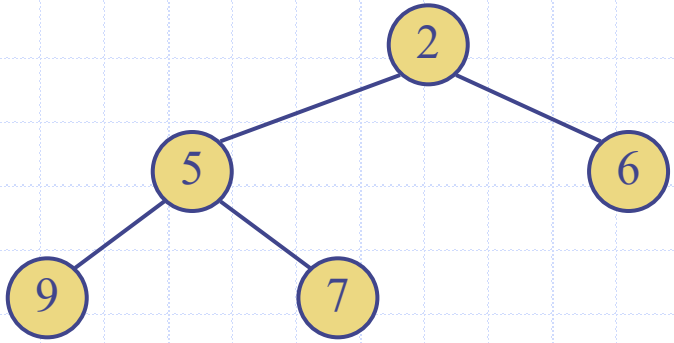
Can we find the last node faster than O(log *n*)?

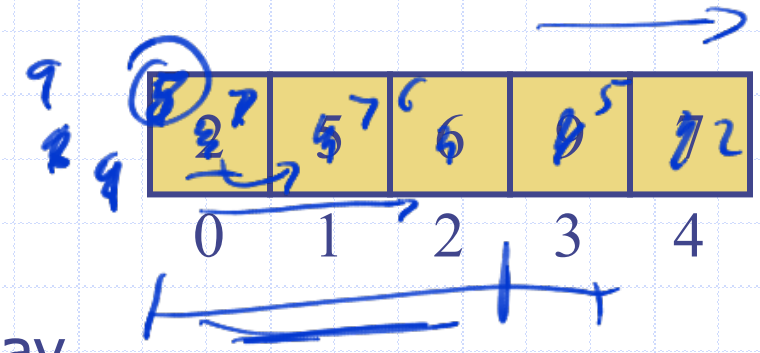
- It depends on how the complete binary tree is implemented
  - Tree with augmentation?
  - **Array!**



Array-Based Heap Implementation

- Can represent a heap with n keys by an array of length n
- For node at index i
  - left child is at index $2i + 1$
  - right child is at index $2i + 2$
- insert corresponds to inserting at index n
- removeMin corresponds to swapping/removing at $n - 1$
- Upheap and downheap implemented by swapping array elements

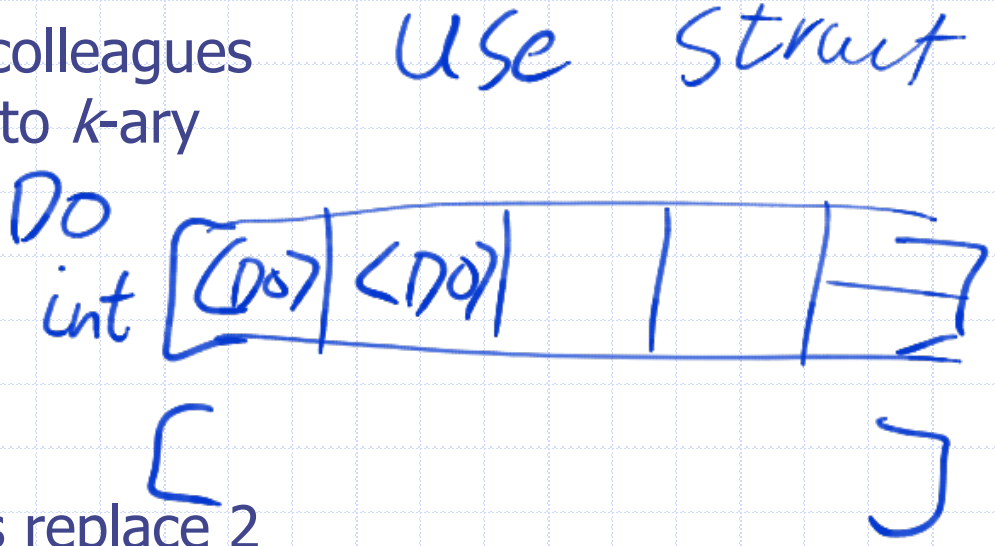




Last Week – Does this generalize to *k*-ary heaps?

Last week, one of our colleagues asked if this generalises to *k*-ary heaps.

Yes, I believe it does!



And, all we need to do is replace 2 with *k* in our code!

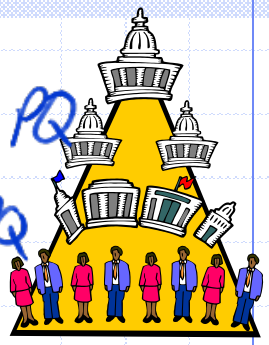
That is: For node at index *i*

- first child is at $ki + 1$
- Last is at index $ki + k$

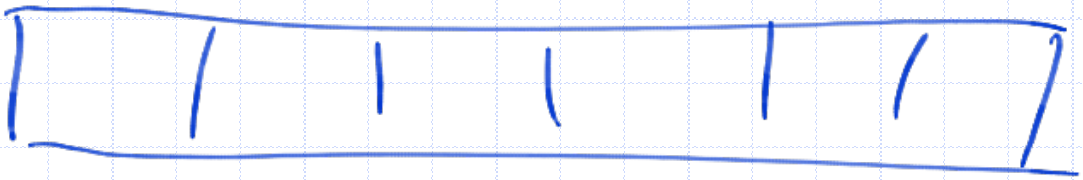
Priority Q

Heap-Sort

1. insert everything in PQ
2. remove everything from PQ



- Consider a priority queue with n items implemented by means of a heap
  - space used is $O(n)$
  - insert and removeMin take $O(\log n)$ time
  - size, isEmpty and min take $O(1)$ time
- Heap-based priority queue can sort a sequence of n elements in $O(n \log n)$ time
  - heap-sort



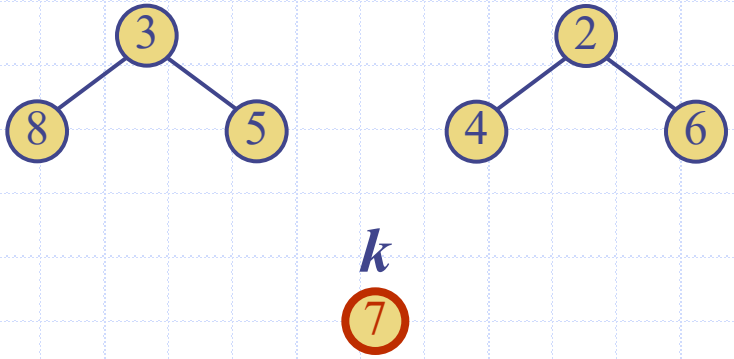
Heap Construction

- Create a heap containing n items
- Using insert
  - n calls gives $O(n \log n)$ time
- Special case construction by merging
  - Bottom-up heap construction
  - Better than $O(n \log n)$ or I wouldn't be telling you about it!

Merging Two Heaps

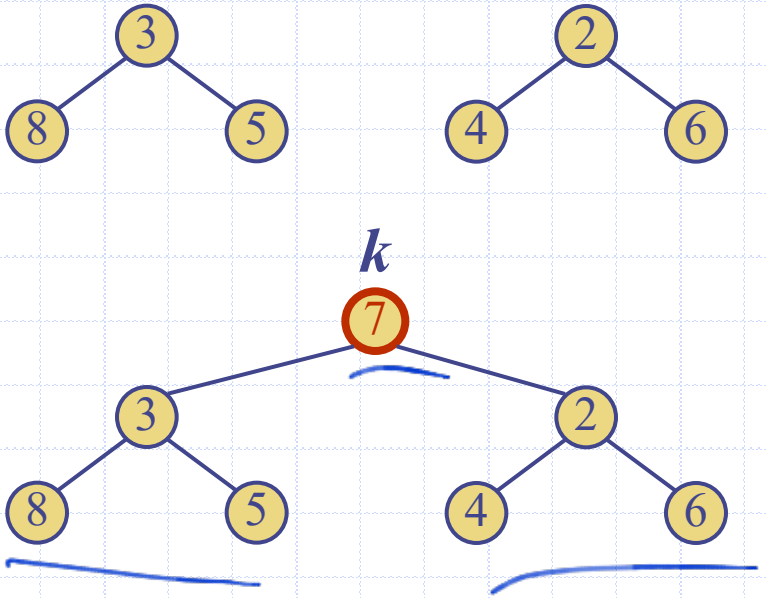
Merging Two Heaps

- Given two heaps and a key k



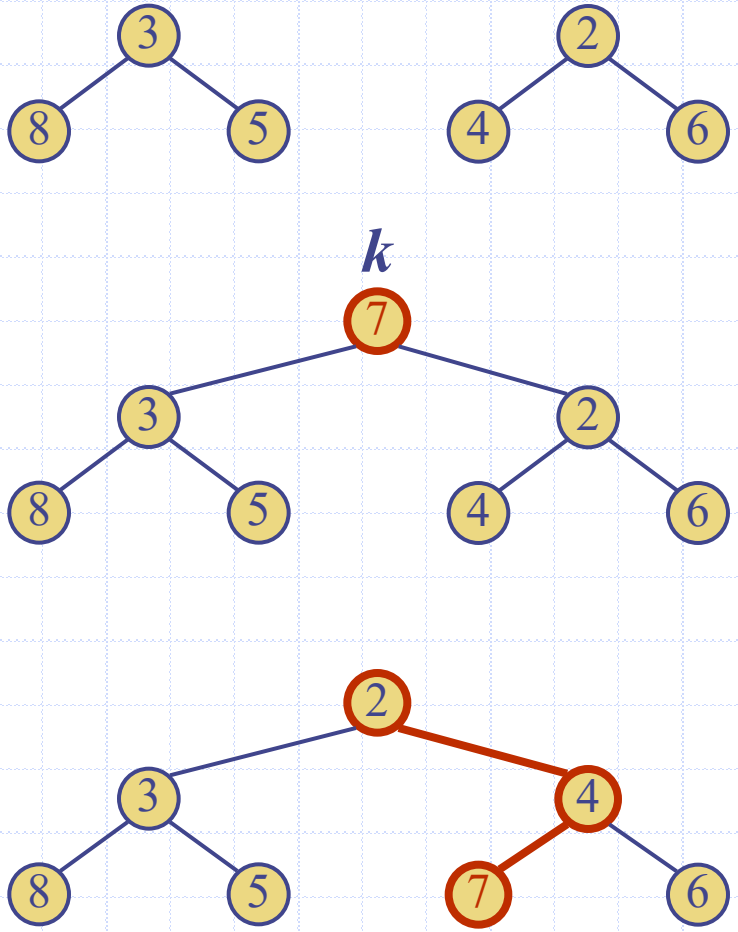
Merging Two Heaps

- Given two heaps and a key k
- Create a new heap with the root node storing k and with the two heaps as subtrees



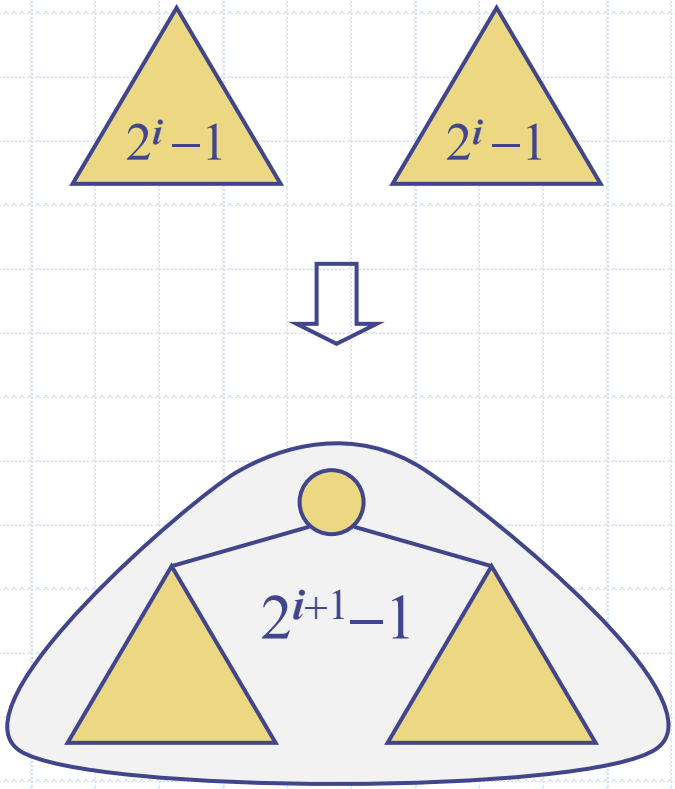
Merging Two Heaps

- Given two heaps and a key k
- Create a new heap with the root node storing k and with the two heaps as subtrees
- Perform downheap to restore the heap-order property



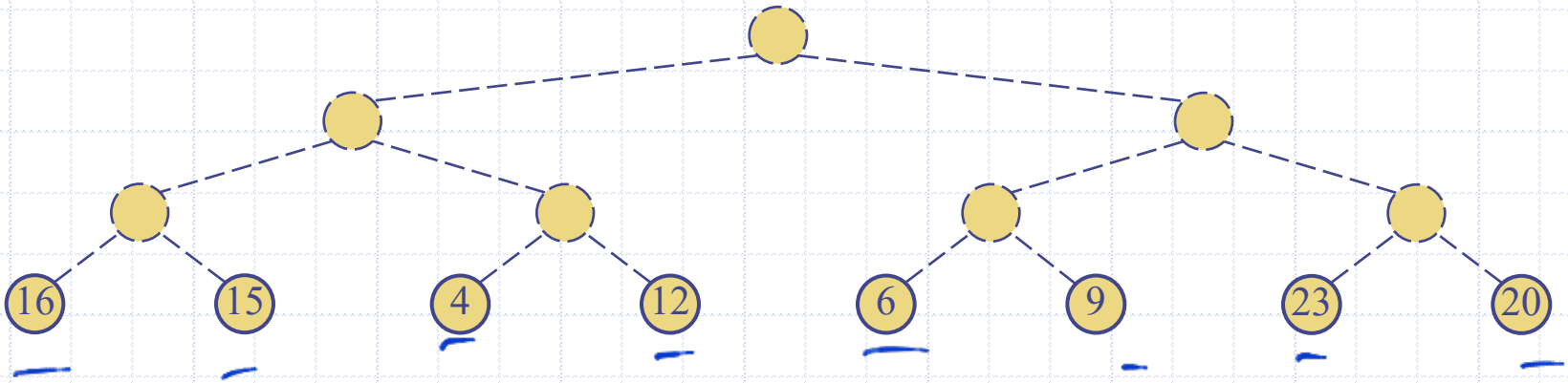
Bottom-Up Heap Construction

- Can construct a heap storing n given keys using bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
- $2^i - 1 + 2^i - 1 + 1 = 2^{i+1} - 1$



Example: Bottom-Up Heap Construction

- Create a heap containing
  - 16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10
  - 1+2+4+8 nodes

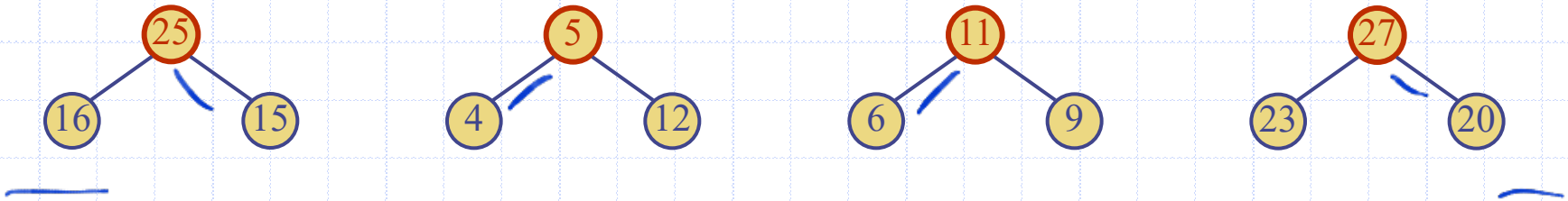


Example (contd.)

16, 15, 4, 12, 6, 9, 23, 20,
25, 5, 11, 27, 7, 8, 10

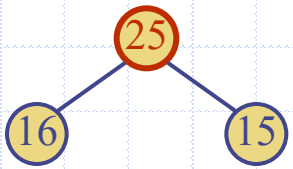
Example (contd.)

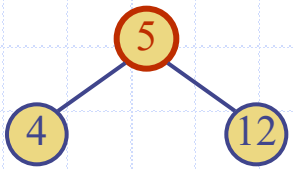
16, 15, 4, 12, 6, 9, 23, 20,
25, 5, 11, 27, 7, 8, 10

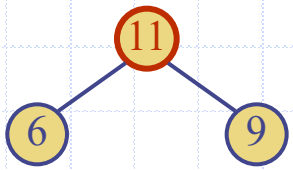


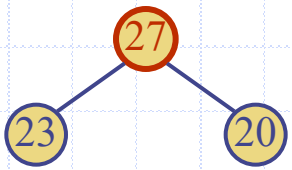
Example (contd.)

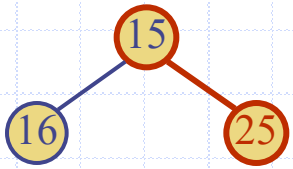
16, 15, 4, 12, 6, 9, 23, 20,
25, 5, 11, 27, 7, 8, 10

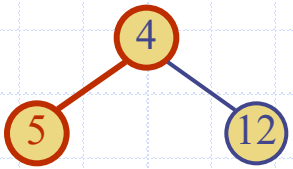


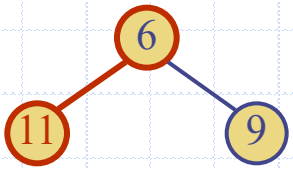


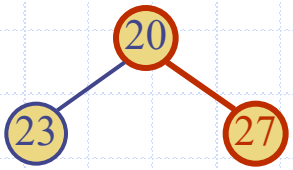






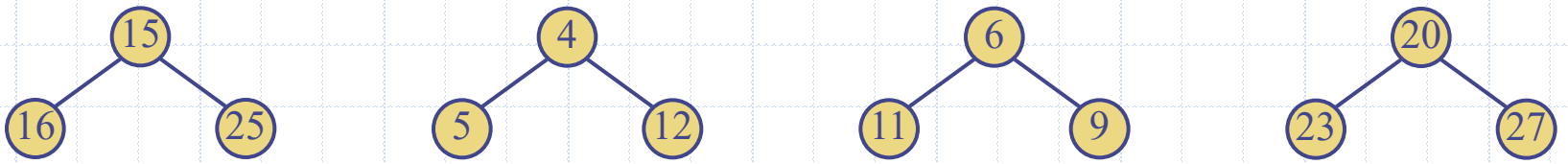






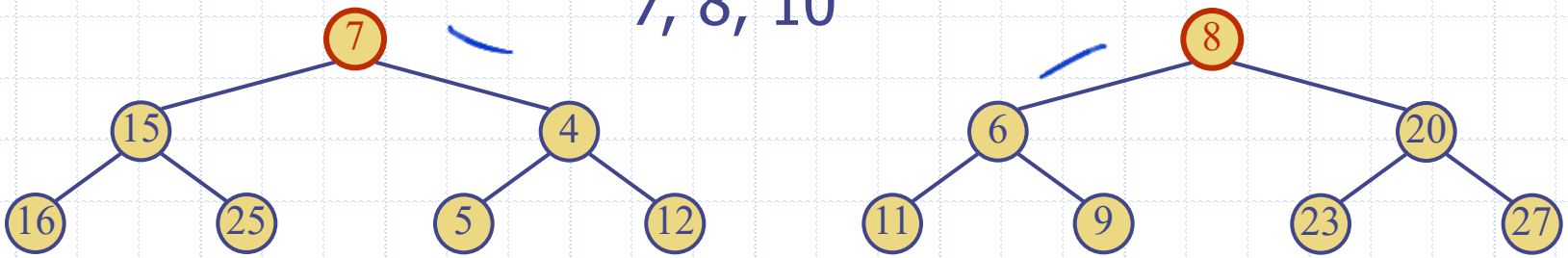
Example (contd.)

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27,
7, 8, 10



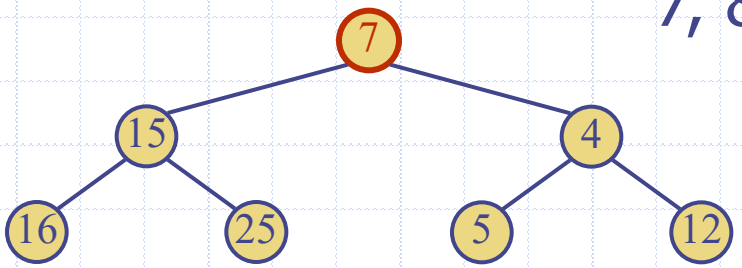
Example (contd.)

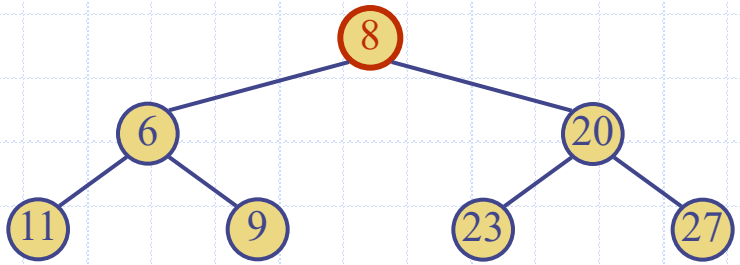
16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10

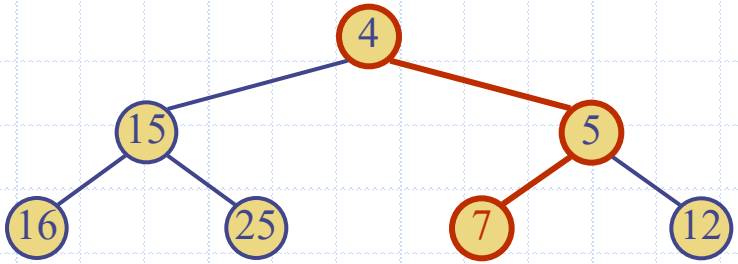


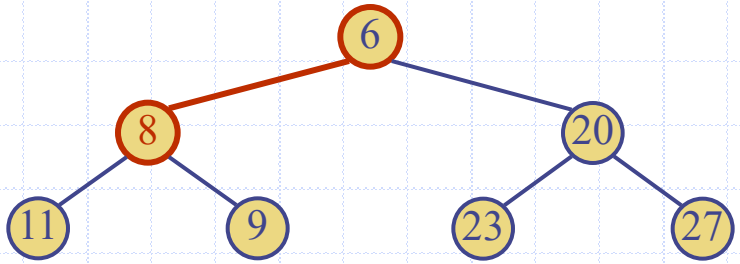
Example (contd.)

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



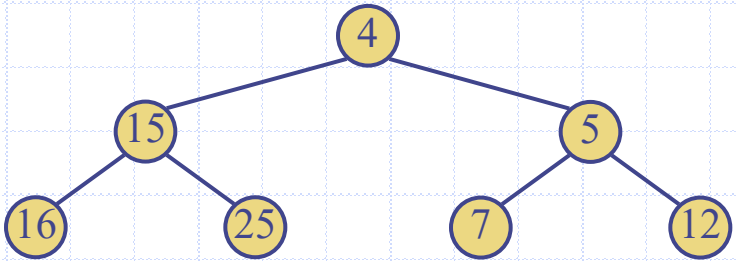


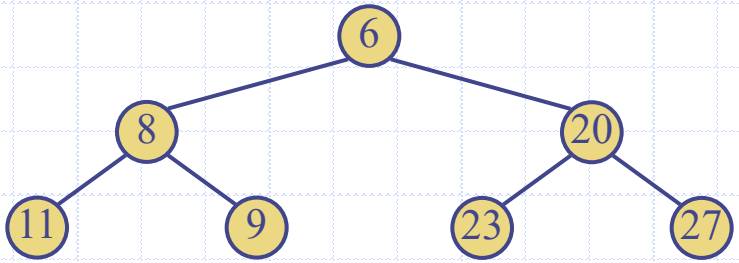




Example (end)

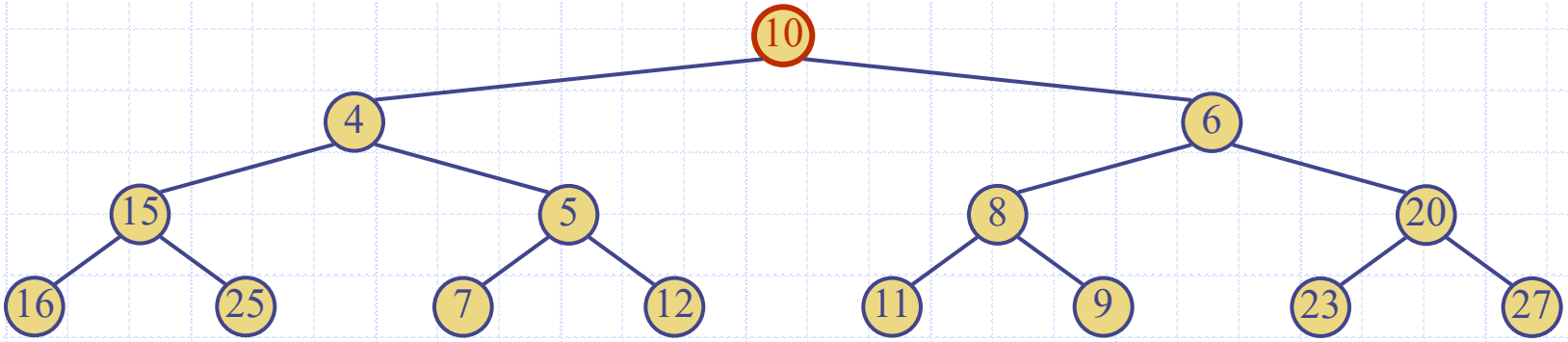
16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10





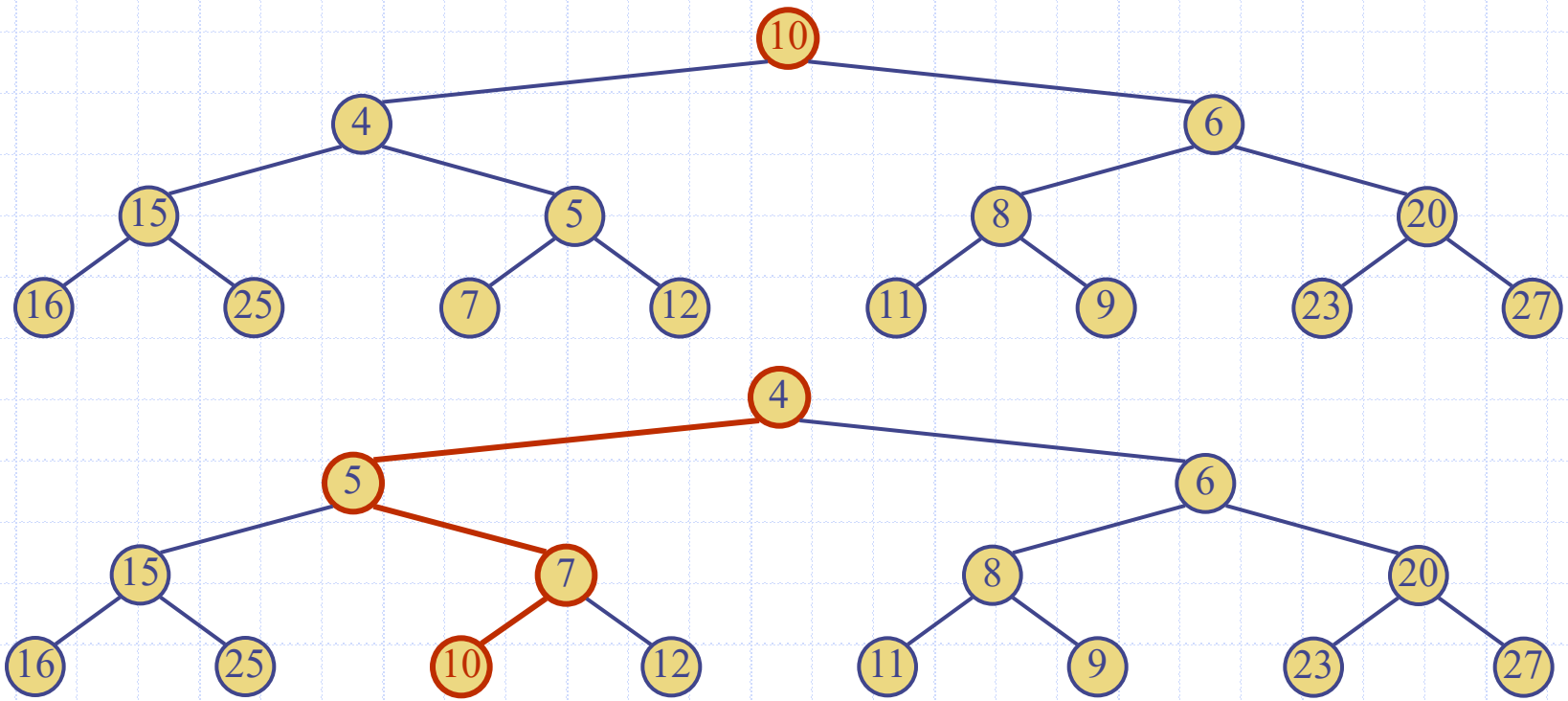
Example (end)

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



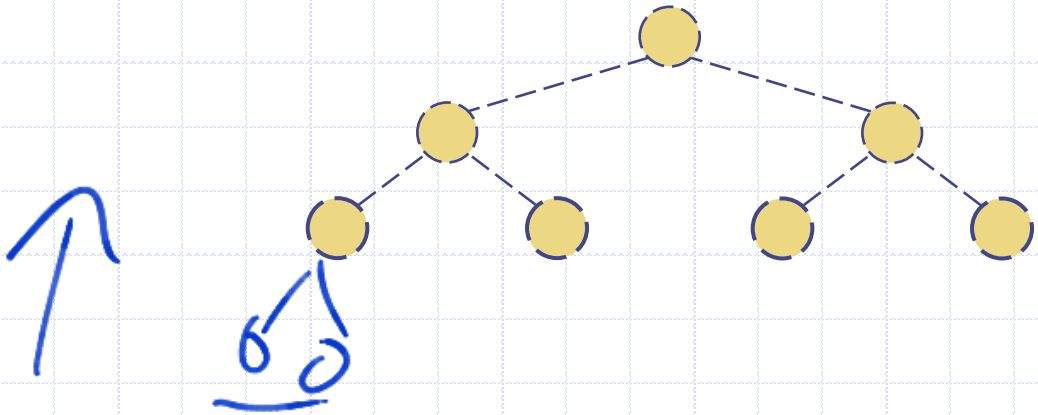
Example (end)

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



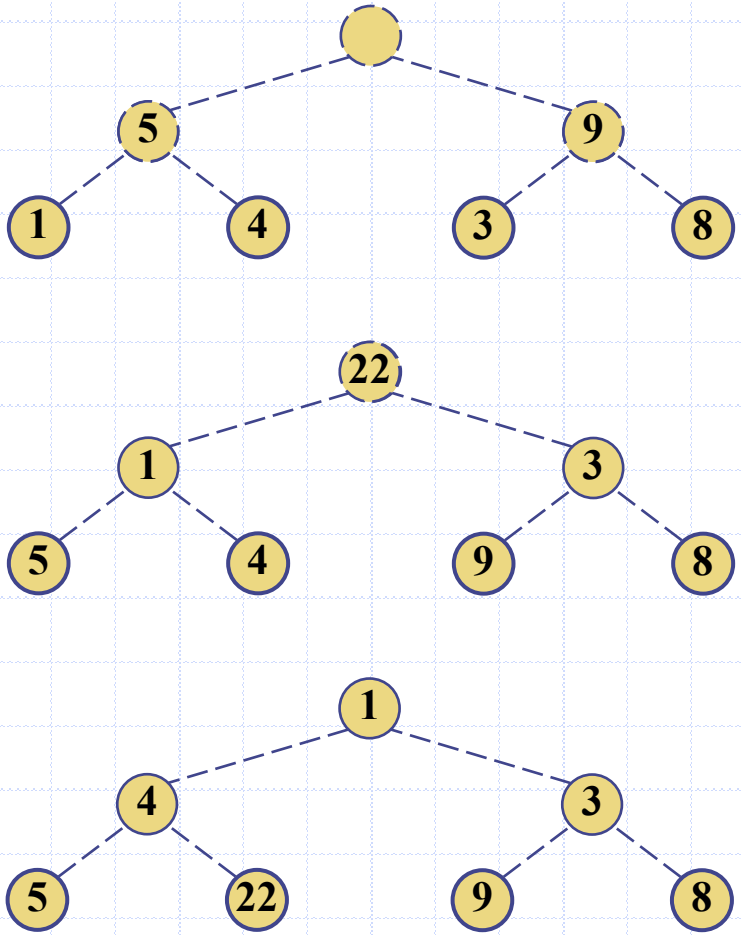
Exercise:
Insert 1, 4, 3, 8, 5, 9, 22 into a Heap

- How many elements should we place
- into leaves?



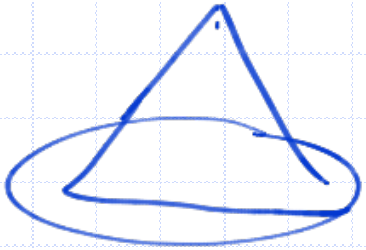
No. leaves = $n - 2^h + 1$
where h = floor of $\log_2 n$
So leaf count = $7 - 2^2 + 1 = 4$

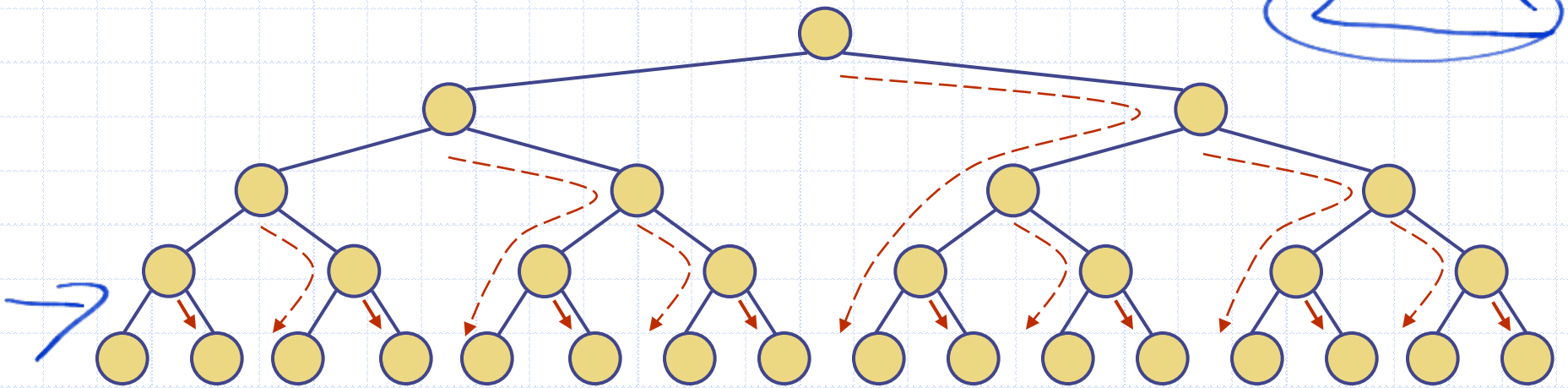
Exercise Solution: Insert 1, 4, 3, 8, 5, 9, 22 into a Heap



Analysis

- Visualise worst-case time of a downheap with a proxy path
- The total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction speeds up the first phase of heap-sort





Exercise

- Think about what building a max-heap (in place) looks like using an array-based heap implementation…
- That is: Given an unsorted array, turn it into a max heap.

Week 5 – Priority Queues and Maps

1. Priority queues
2. Heaps
3. Adaptable priority queues
4. Maps
5. Sets

Adaptable Priority Queue ADT

- remove(e) – remove an arbitrary element (not necessarily the minimum/maximum)
- replaceKey(e, k)
  - Error condition occurs if k is invalid
    - k cannot be compared with other keys
  - Essentially a "priority update"
- replaceValue(e, v)
  - Changing the value of an item (not the key)

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }
key(e_2)	3	{ (3,B), (5,A), (7,C) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }
key(e_2)	3	{ (3,B), (5,A), (7,C) }
remove(e_1)	e_1	{ (3,B), (7,C) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }
key(e_2)	3	{ (3,B), (5,A), (7,C) }
remove(e_1)	e_1	{ (3,B), (7,C) }
replaceKey(e_2, 9)	3	{ (7,C), (9,B) }

Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }
key(e_2)	3	{ (3,B), (5,A), (7,C) }
remove(e_1)	e_1	{ (3,B), (7,C) }
replaceKey(e_2, 9)	3	{ (7,C), (9,B) }
replaceValue(e_3, D)	C	{ (7,D), (9,B) }

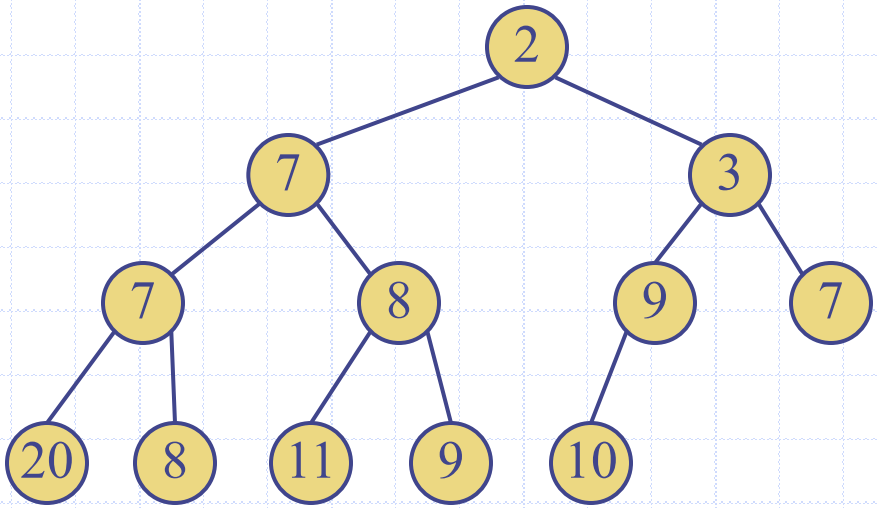
Example

Operation	*Output*	*PQ*
insert(5, A)	e_1	{ (5,A) }
insert(3, B)	e_2	{ (3,B), (5,A) }
insert(7, C)	e_3	{ (3,B), (5,A), (7,C) }
min()	e_2	{ (3,B), (5,A), (7,C) }
key(e_2)	3	{ (3,B), (5,A), (7,C) }
remove(e_1)	e_1	{ (3,B), (7,C) }
replaceKey(e_2, 9)	3	{ (7,C), (9,B) }
replaceValue(e_3, D)	C	{ (7,D), (9,B) }
remove(e_2)	e_2	{ (7,D) }

Locating Entries

- remove(e), replaceKey(e, k), and replaceValue(e, v), need fast ways of locating an entry e in a priority queue
- Can search the entire (queue) data structure to find an entry e
  - but there are better ways for locating entries
- Position in the data structure implementing the PQ is called the location of the entry

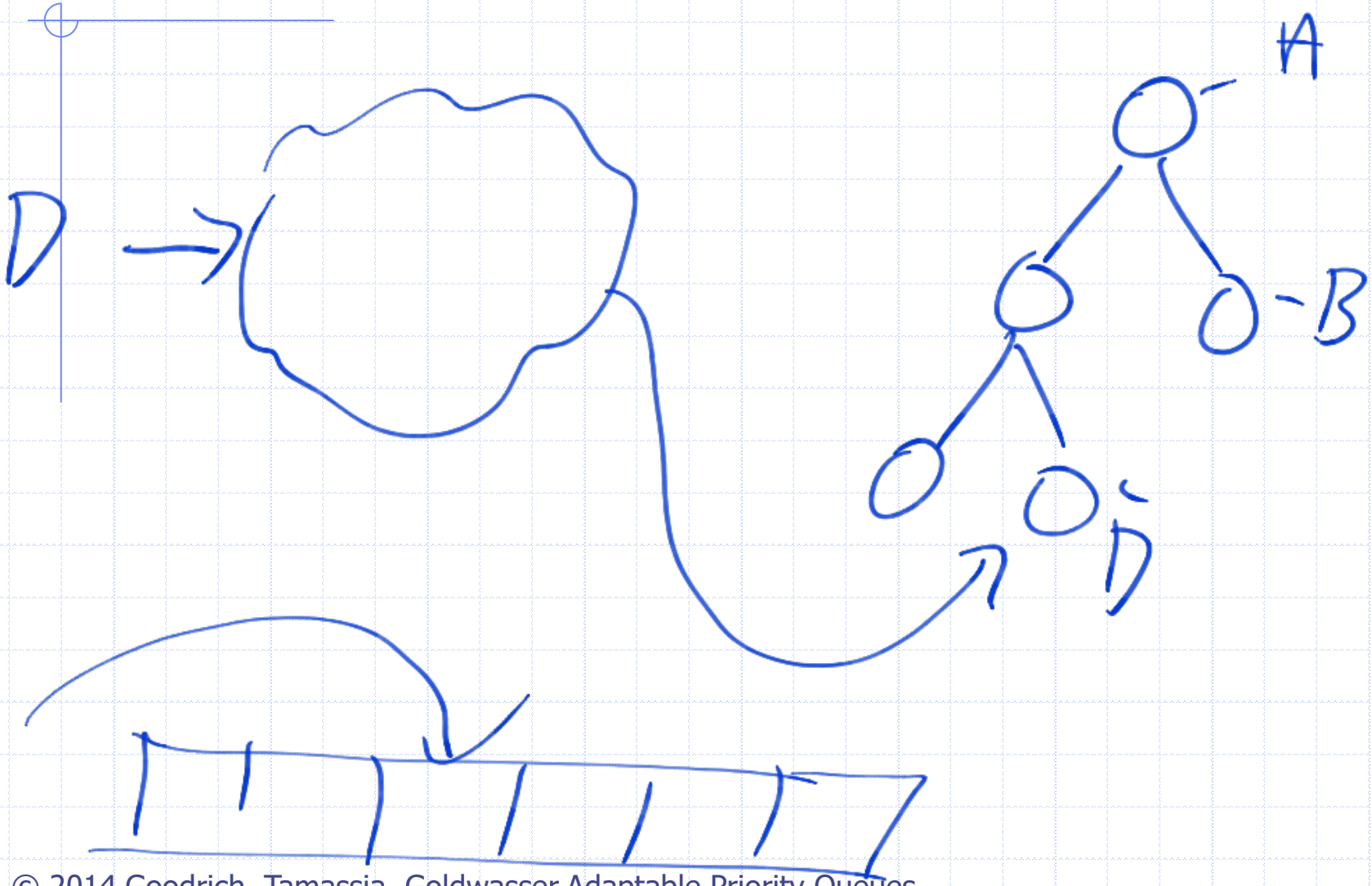
How could you locate the entry with key 11?



Location-Aware Entries

- Locators can be used to keep track of elements as they are moved around inside a container
- That is, keep "pointers" to keys inside the tree that follow each (key, value) node as they move around
  - In practice, assuming array-based storage, this just looks like each item storing a member variable containing the index of its current array position

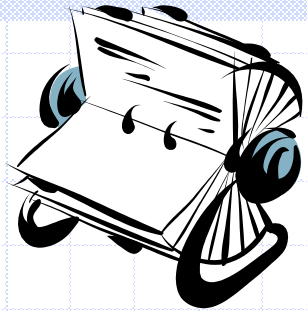
Sketch of this idea



Week 5 – Priority Queues & Maps

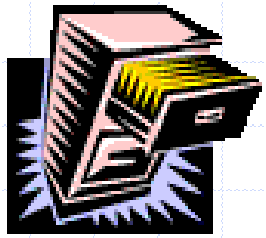
1. Priority queues
2. Heaps
3. Adaptable priority queues
4. Maps
5. Sets

Maps



- Searchable collection of key-value entries
- Main operations
  - searching, inserting and deleting items
- Keys are unique
  - cannot have multiple entries with same key
- Applications
  - address book
  - student-record database

Map ADT



- get(k)
- put(k, v)
- remove(k)
- size(), isEmpty()
- entrySet() – yield a collection of key/value pairs
- keySet() – yield an collection of keys
- values() – yield a collection of values

- Note: The last three could return a container, or may just return an iterable.

Use of null as a Sentinel

- get, put and remove return null (None) if a requested entry is not present
  - None is a sentinel value
- Thus, we cannot store None as a key!
  - Alternative: throw an exception for a key that is not in the map

Example

Operation	Return	Map Contents

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	$\emptyset$
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	$\emptyset$
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(2)	*E*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(2)	*E*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
size()	4	(5,*A*), (7,*B*), (2,*E*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(2)	*E*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
size()	4	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
remove(5)	*A*	(7,*B*), (2,*E*), (8,*D*)
remove(2)	*E*	(7,*B*), (8,*D*)

Example

Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(2)	*E*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
size()	4	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
remove(5)	*A*	(7,*B*), (2,*E*), (8,*D*)
remove(2)	*E*	(7,*B*), (8,*D*)
get(2)	**null**	(7,*B*), (8,*D*)

Example

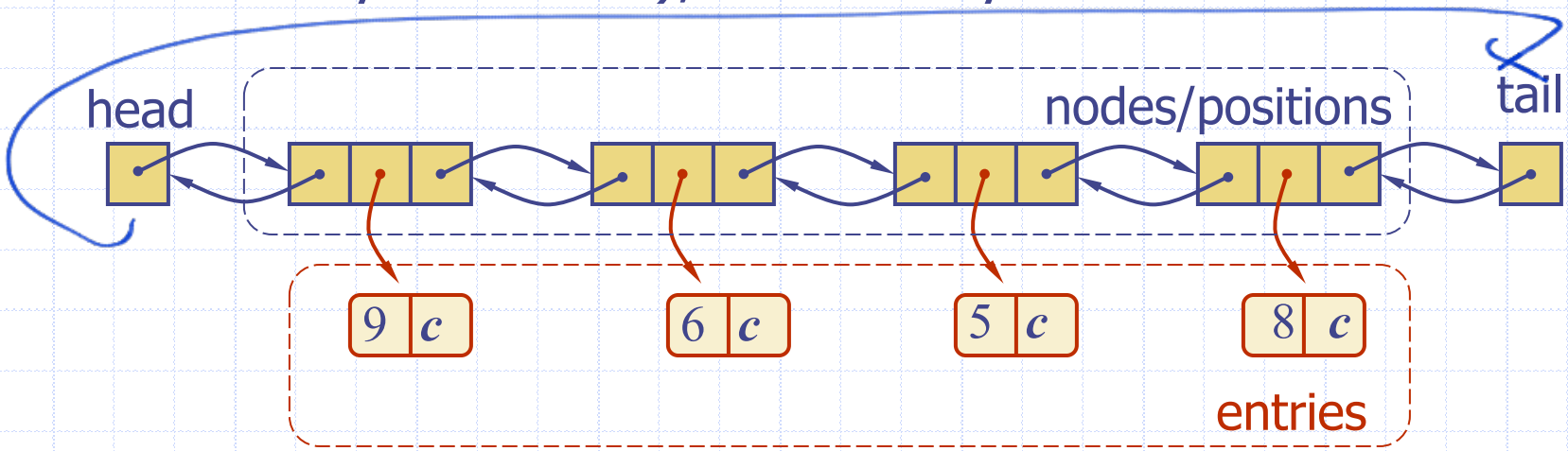
Operation	Return	Map Contents
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*), (7,*B*)
put(2,*C*)	**null**	(5,*A*), (7,*B*), (2,*C*)
put(8,*D*)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
put(2,*E*)	*C*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(7)	*B*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(4)	**null**	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
get(2)	*E*	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
size()	4	(5,*A*), (7,*B*), (2,*E*), (8,*D*)
remove(5)	*A*	(7,*B*), (2,*E*), (8,*D*)
remove(2)	*E*	(7,*B*), (8,*D*)
get(2)	**null**	(7,*B*), (8,*D*)
isEmpty()	**false**	(7,*B*), (8,*D*)

Comparison with Earlier Data Structures

- Priority queues and maps both store key-value pairs
  - We call these key-value pairs entries
- Keys
  - Map keys **must be unique**
  - Priority queue allows multiple entries to have same key
    - Two or more elements may share the same priority of course!
- Total order relation on keys
  - Required for priority queues
  - Optional for maps (may or may not be ordered)
- Accessing elements
  - Maps allow access to any entry by key
  - Priority queues allow access to highest priority element

Simple List-Based Map

- Implemented using an unsorted list
  - store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Grand Idea: Use a linked list to store what you have in your collection! Right??

get(k) Algorithm

```
Algorithm get(k):
    cur = head
    while cur is not empty do
        if  cur.getKey() = k  then
            return  cur.get_value()
    return  null
```

Complexity: $O(n)$

We can surely do better??

put(k, v) Algorithm

Algorithm put(k, v):

```
cur = tail
new = Node(k, v)
cur.set_next = new
new.set_prev = cur
self._tail = new
```

Complexity: $O(1)$ – What if we hadn't augmented the tail pointer?

put(k, v) Algorithm

Algorithm put(k, v):
    cur = tail
    new = Node(k, v)
    cur.set_next = new
    new.set_prev = cur
    self._tail = new

HANG ON!!! Something is wrong!!! What is it?

~~***Complexity: O(1)***~~ ~~What if we hadn't augmented the tail pointer~~?
Complexity is $O(n)$ – We need to check for duplicates of course…
Ouch!

put(k, v) Algorithm

```
Algorithm put(k, v):
    cur = head
    while cur is not empty do
        if  cur.getKey() = k  then
            cur.set_value(v)
            return
    cur = tail
    new = Node(k, v)
    cur.set_next = new
    new.set_prev = cur
    self._tail = new
```

remove(k) Algorithm

```
Algorithm remove(k):
    cur = head
    while cur is not empty do
        if  cur.getKey() = k  then
            # Typical linked list deletion
```

O(n)

O(1)

Complexity: O(n)

Can we do better?

- Yes, of course!
- This is our focus for the next **2 weeks**
  - Hashing + Hash Tables
  - Tree-Based Maps

A few final words on sets and multi-sets/maps

- Ultimately, a very similar set of CDTs to Maps with a few minor implementation differences; so we don’t stress too hard about these.

Multimaps

get(k)

put(k, v)

remove(k, v)

removeAll(k)

size( )

entries( )

keys( )

keySet( )

values( )

Similar to a map but **can** store multiple entries with the **same** key.

Returns collections of values associated with the given key.

Week 5 – Priority Queues & Maps

1. Priority queues
2. Heaps
3. Adaptable priority queues
4. Maps
5. Sets

Definitions

- **Set** – unordered collection of elements, without duplicates
  - elements of a set are like keys of a map, but without any auxiliary values
- **Multiset** (alias **Bag**) – set-like container that allows duplicates

Set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e:\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Intuition: Union on sorted lists

- Generalised merge of two sorted lists A and B
- Auxiliary methods
  - remove_first
- Runs in $O(n_A + n_B)$ time assuming the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge(A, B):
    S ← empty sequence
    while not A.isEmpty() ∧ not B.isEmpty()
        a ← A.first().value(); b ← B.first().value()
        if a < b
            append a to S; A.remove_first()
        else if b < a
            append b to S; B.remove_first()
        else { b = a }
            append a to S
            A.remove_first()
            B.remove_first()
    while ¬A.isEmpty()
        append a to S; A.remove_first()
    while ¬B.isEmpty()
        append b to S; B.remove_first()
    return S
```

Further Reading

- Data Structures and Algorithms in Python
  - Chapter 9
  - Chapter 10.1, 10.5
- Introduction to Algorithms
  - Chapter 6