CS121 Data Structures A, C
Heaps

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

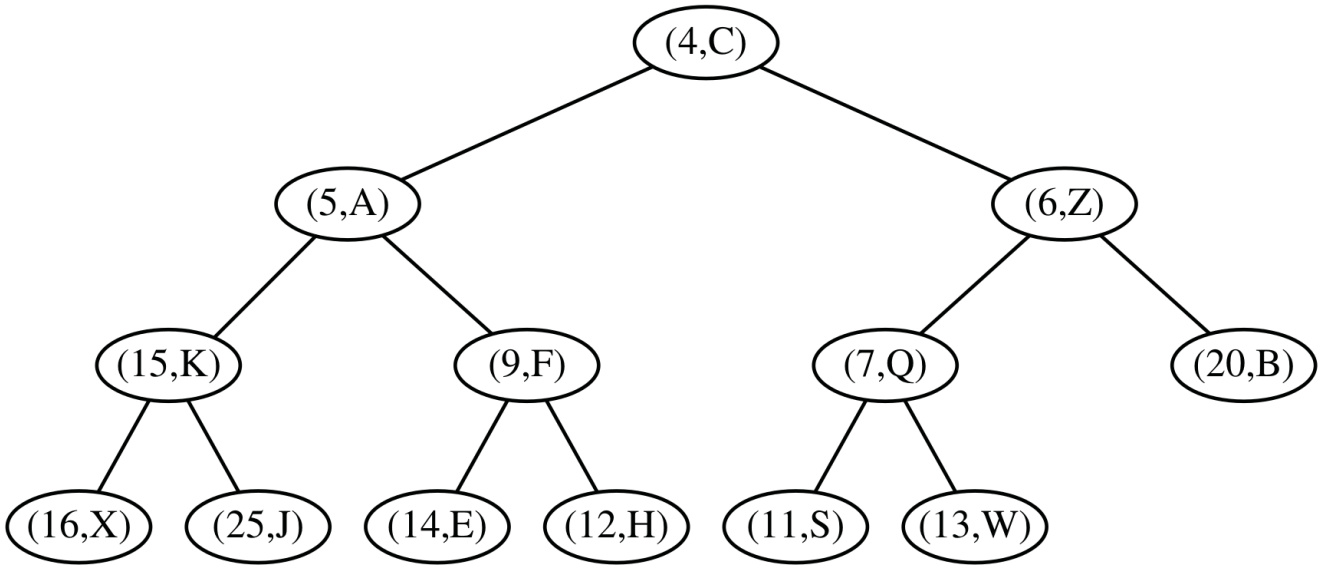
Fall 2021

Heaps

In addition to the implementations of a priority queue with an unsorted and a sorted lists, we can implement it using a data structure called a **binary heap**

Allows to perform both insertions and removals in logarithmic time

Uses the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted



The Heap Data Structure: Heap-Order Property

A **heap** is a binary tree T that stores entries at its positions and that satisfies a relational property and a structural property

- **Heap-Order Property**: for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent

Hence, the keys encountered on a path from the root to a leaf of T are in nondecreasing order

A minimal key is always stored at the root of T, i.e. easy to locate such an entry "at the top of the heap"

The Heap Data Structure: Complete Binary Tree Property

For the sake of efficiency, we want the heap T to have as small a height as possible

- **Complete Binary Tree Property**: A heap T with height h is a **complete** binary tree if levels 0, 1, 2, ..., $h-1$ of T have the maximal number of nodes possible (level i has 2^i nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level

A complete binary tree with n elements is one that has positions with level numbering 0 through $n-1$

The Height of a Heap

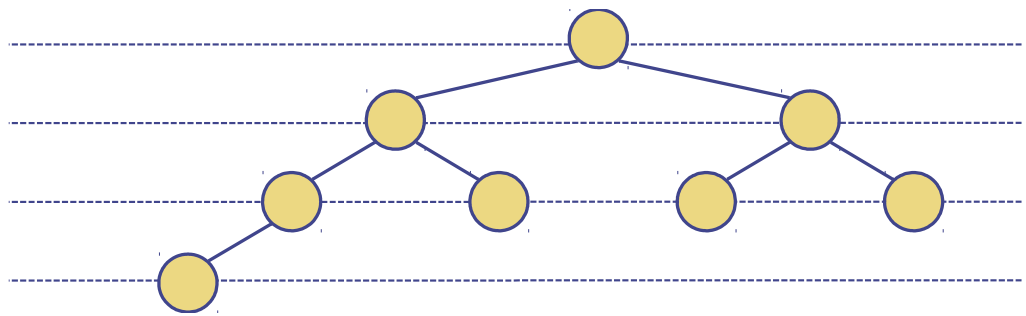
Proposition: A heap T storing n entries has height $h = \lfloor \log n \rfloor$

Justification: Since T is complete, the number of nodes in levels 0 through $h-1$ is precisely $1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$; in level h there are between 1 and 2^h nodes. Therefore

$$2^h = 2^h - 1 + 1 \quad \leq \quad n \quad \leq \quad 2^h - 1 + 2^h = 2^{h+1} - 1$$

Taking the logarithm, we get $h \leq \log n$ and $h \geq \log(n+1) - 1$, thus $h = \lfloor \log n \rfloor$

depth	entries
0	1
1	2
$h-1$	2^{h-1}
h	1



Implementing a Priority Queue with a Heap

If we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time

We use the composition pattern to store key-value pairs as entries in the heap

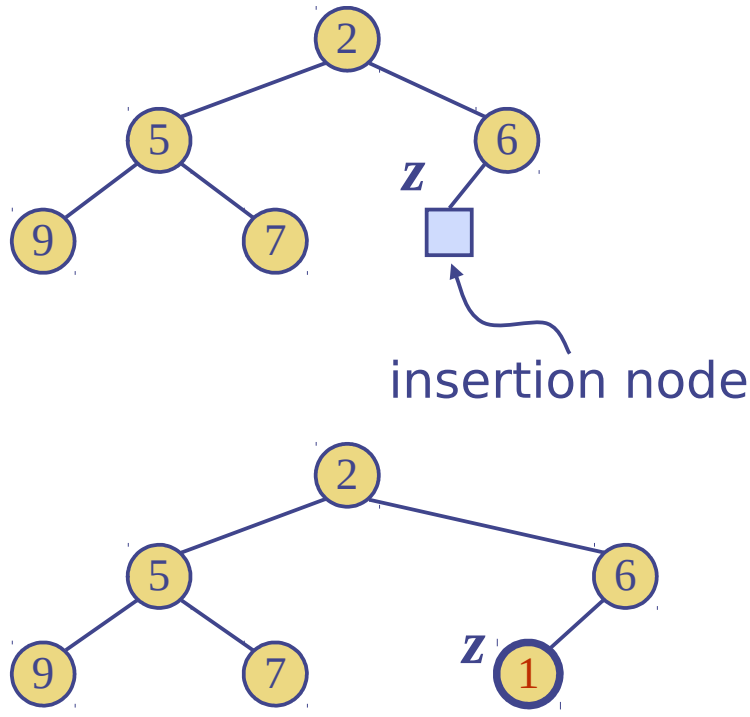
The `size` and `isEmpty` methods can be implemented based on examination of the tree

The `min` operation is equally trivial because the heap property assures that the element at the root of the tree has a minimal key

Adding an Entry to the Heap

Method `insert` of the priority queue ADT corresponds to the insertion of an entry (k, v) to the heap
The insertion algorithm consists of three steps:

1. Maintaining the **complete binary tree property**, find the insertion node z (the new last node)
2. Store (k, v) at z
3. Restore the **heap-order property**



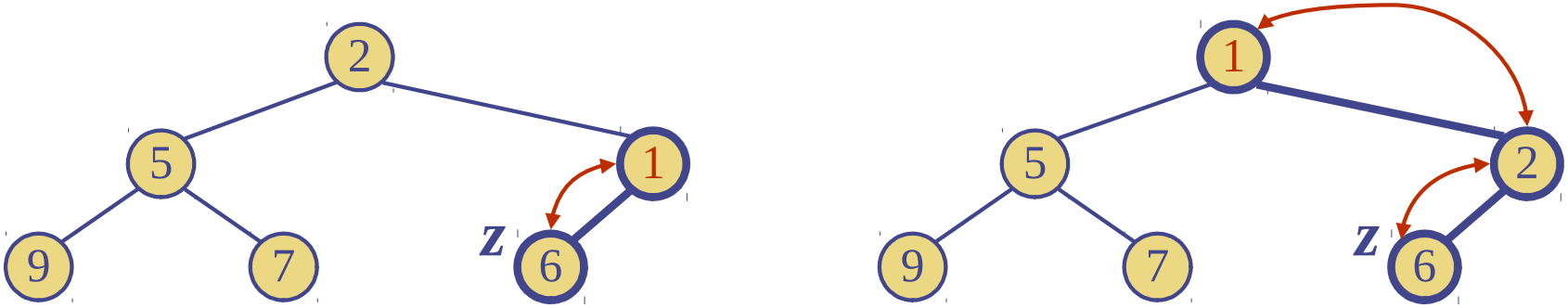
Up-Heap Bubbling

After the insertion of a new entry (k, v), the heap-order property may be violated

Algorithm **up-heap bubbling** restores the heap-order property by swapping (k, v) along an upward path from the insertion node

Up-heap terminates when the entry (k, v) reaches the root or a node whose parent has a key smaller than or equal to k

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

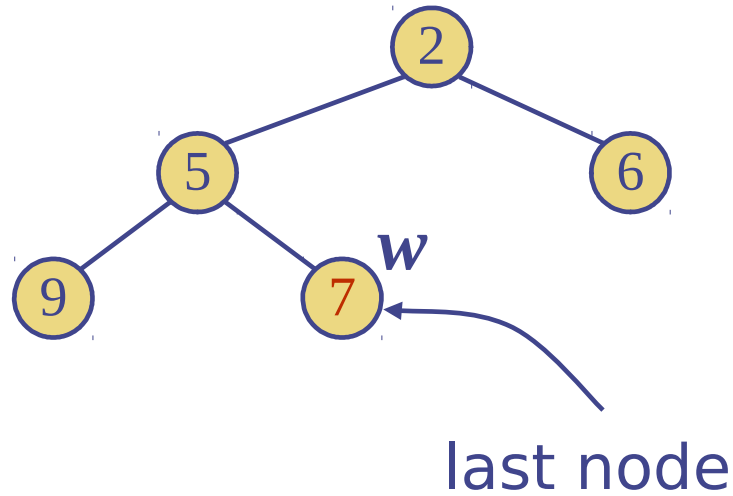


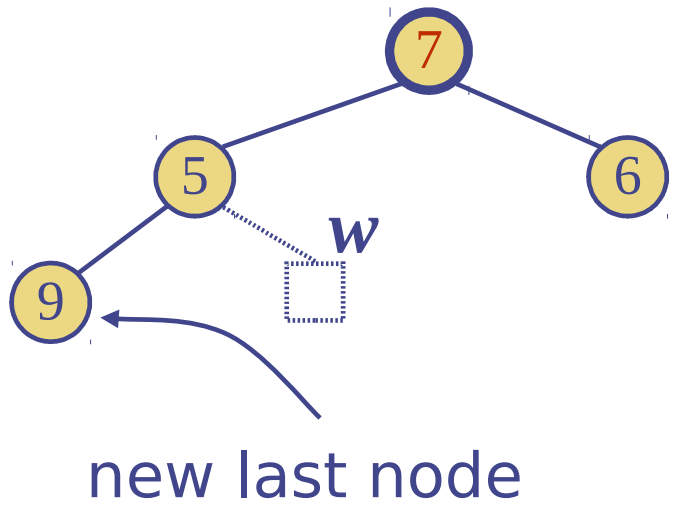
Removing the Entry with Minimal Key

Method `removeMin` of the priority queue ADT corresponds to the removal of the root entry from the heap

The removal algorithm consists of three steps:

1. Replace the root entry with the entry of the last node w
2. Remove w, thus maintaining the **complete binary tree property**
3. Restore the **heap-order property**





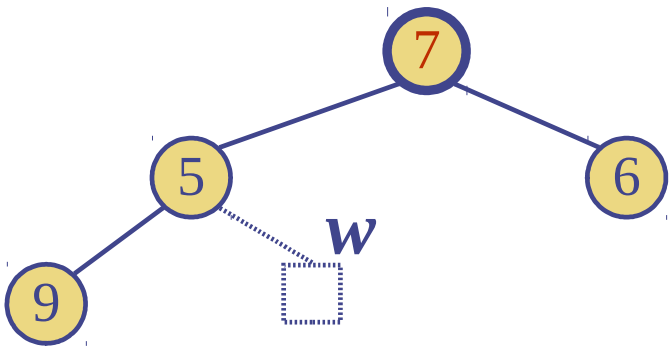
Down-Heap Bubbling

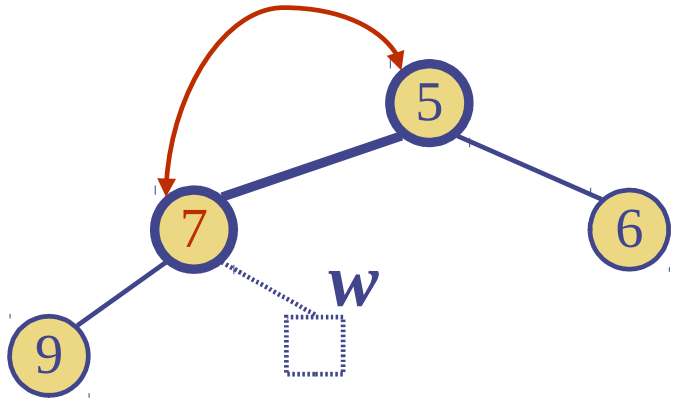
After replacing the root entry with the entry (k, v) of the last node, the heap-order property may be violated

Algorithm down-heap restores the heap-order property by swapping entry (k, v) along a downward path from the root

Down-heap terminates when entry (k, v) reaches a leaf or a node whose children have keys greater than or equal to k

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



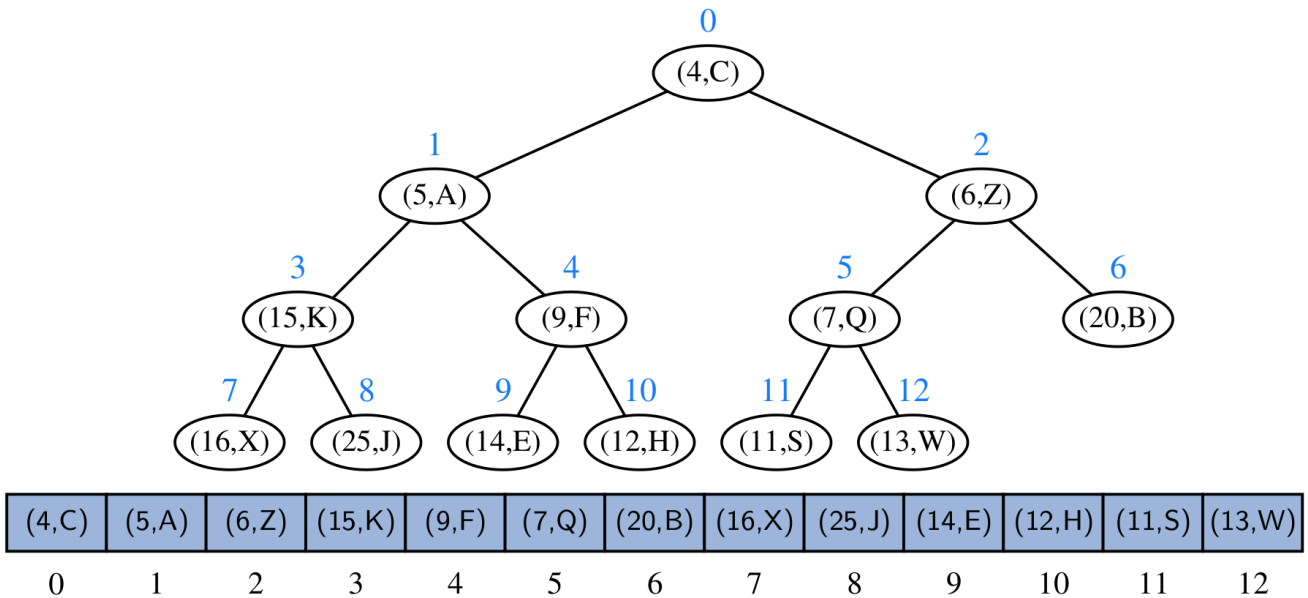


Array-Based Representation of a Binary Tree

The element at position p is stored at $A[f(p)]$

f is called **level numbering** of positions in T

- If p is the root, then $f(p) = 0$
- If p is the left child of q, then $f(p) = 2f(q) + 1$
- If p is the right child of q, then $f(p) = 2f(q) + 2$



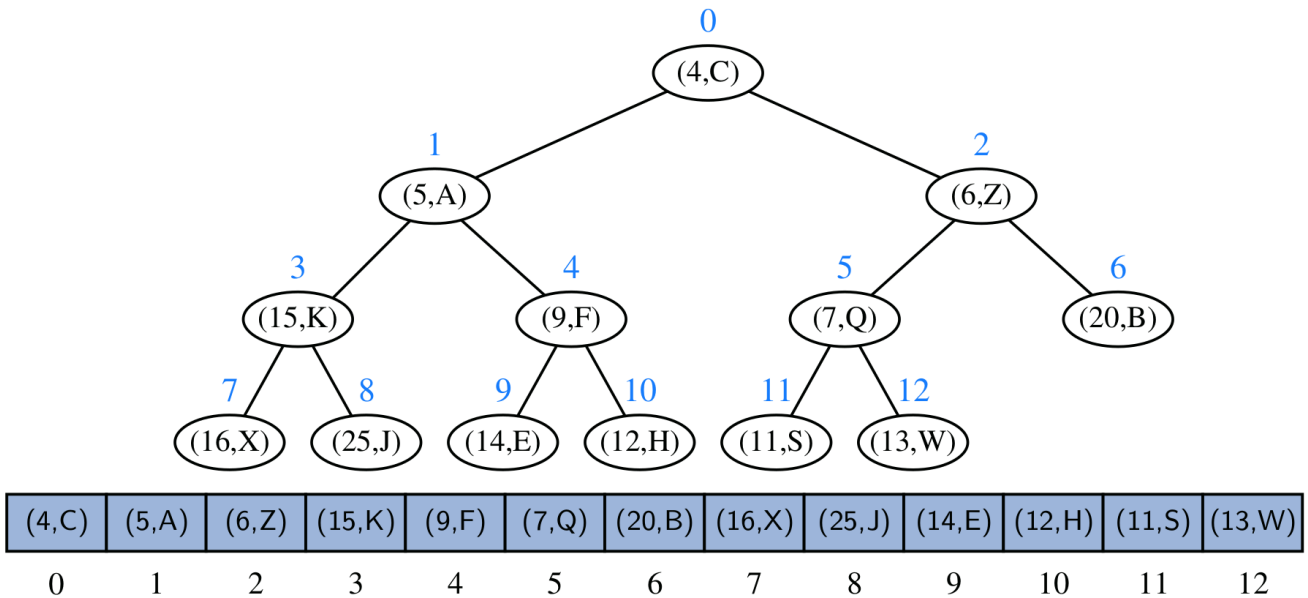
Space requirements for a complete tree?

Array-Based Representation of a Binary Tree

The element at position p is stored at $A[f(p)]$

f is called **level numbering** of positions in T

- If p is the root, then $f(p) = 0$
- If p is the left child of q, then $f(p) = 2f(q) + 1$
- If p is the right child of q, then $f(p) = 2f(q) + 2$



Space requirements for a complete tree?

The elements have contiguous indices in the range $[0, n-1]$

Array-Based Heaps

The array-based heap representation avoids some complexities of a linked tree structure

Methods `insert` and `removeMin` depend on locating the last position of a heap

With the array-based representation of a heap of size n, the last position is simply at index $n - 1$

If the size of a priority queue is not known in advance, we need to dynamically resize the array on occasion, as is done with a Java `ArrayList`

The time bounds of methods for adding or removing elements become **amortized**

Heap-Based Priority Queue: Analysis

The running time of the priority queue ADT functions for the heap implementation of a priority queue:

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

Space usage: $O(n)$, where n is the number of entries in the priority queue

We assume that two keys can be compared in $O(1)$ time and that the heap T is implemented with an array-based or linked-based tree representation

* amortized, if using dynamic array

Heap Implementation in Java

```
1  /** An implementation of a priority queue using an array-based heap. */
2  public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3    /** primary collection of priority queue entries */
4    protected ArrayList<Entry<K,V>> heap = new ArrayList<>( );
5    /** Creates an empty priority queue based on the natural ordering of its keys. */
6    public HeapPriorityQueue( ) { super( ); }
7    /** Creates an empty priority queue using the given comparator to order keys. */
8    public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
9    // protected utilities
10   protected int parent(int j) { return (j-1) / 2; } // truncating division
11   protected int left(int j) { return 2*j + 1; }
12   protected int right(int j) { return 2*j + 2; }
13   protected boolean hasLeft(int j) { return left(j) < heap.size( ); }
14   protected boolean hasRight(int j) { return right(j) < heap.size( ); }
15   /** Exchanges the entries at indices i and j of the array list. */
16   protected void swap(int i, int j) {
17     Entry<K,V> temp = heap.get(i);
18     heap.set(i, heap.get(j));
19     heap.set(j, temp);
20   }
```

Heap Implementation in Java (cont'd)

```
21    /** Moves the entry at index j higher, if necessary, to restore the heap property. */
22    protected void upheap(int j) {
23      while (j > 0) {       // continue until reaching root (or break statement)
24        int p = parent(j);
25        if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
26        swap(j, p);
27        j = p;                                    // continue from the parent's location
28      }
29    }
30    /** Moves the entry at index j lower, if necessary, to restore the heap property. */
31    protected void downheap(int j) {
32      while (hasLeft(j)) {                        // continue to bottom (or break statement)
33        int leftIndex = left(j);
34        int smallChildIndex = leftIndex;          // although right may be smaller
35        if (hasRight(j)) {
36          int rightIndex = right(j);
37          if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
38            smallChildIndex = rightIndex;         // right child is smaller
39        }
40        if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
41          break;                                  // heap property has been restored
42        swap(j, smallChildIndex);
43        j = smallChildIndex;                      // continue at position of the child
44      }
45    }
```

Heap Implementation in Java (cont'd)

```
46    /** Returns the number of items in the priority queue. */
47    public int size( ) { return heap.size( ); }
48    /** Returns (but does not remove) an entry with minimal key (if any). */
49    public Entry<K,V> min( ) {
50      if (heap.isEmpty( )) return null;
51      return heap.get(0);
52    }
53    /** Inserts a key−value pair and returns the entry created. */
54    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
55      checkKey(key);  // auxiliary key−checking method (could throw exception)
56      Entry<K,V> newest = new PQEntry<>(key, value);
57      heap.add(newest);                          // add to the end of the list
58      upheap(heap.size( ) − 1);                  // upheap newly added entry
59      return newest;
60    }
61    /** Removes and returns an entry with minimal key (if any). */
62    public Entry<K,V> removeMin( ) {
63      if (heap.isEmpty( )) return null;
64      Entry<K,V> answer = heap.get(0);
65      swap(0, heap.size( ) − 1);                 // put minimum item at the end
66      heap.remove(heap.size( ) − 1);             // and remove it from the list;
67      downheap(0);                               // then fix new root
68      return answer;
69    }
70  }
```

Bottom-Up Heap Construction

Consider the task of constructing a heap of given n entries

- start with an initially empty heap and make n successive calls to the `insert` operation, or
- construct the heap **bottom-up**

If the take the first approach with n calls to `insert`, the worst case running time will be

Bottom-Up Heap Construction

Consider the task of constructing a heap of given n entries

- start with an initially empty heap and make n successive calls to the `insert` operation, or
- construct the heap **bottom-up**

If the take the first approach with n calls to `insert`, the worst case running time will be $O(n \log n)$

Bottom-up heap construction runs in $O(n)$ time

Bottom-Up Heap Construction Procedure

For simplicity assume $n = 2^{h+1} - 1$. Then the heap height is

Bottom-Up Heap Construction Procedure

For simplicity assume $n = 2^{h+1} - 1$. Then the heap height is $h = \log(n+1) - 1$

1. Construct $(n+1)/2$ elementary heaps with one entry each
2. Form $(n+1)/4$ heaps with three entries each: join pairs of elementary heaps, add new entry, and down-heap
3. Form $(n+1)/8$ heaps with seven entries each: join pairs of 3-entry heaps, add new entry, and down-heap

$\vdots$

i. $2 \leq i \leq h$, Form $(n+1)/2^i$ heaps with $2^i - 1$ entries each: join pairs of $(2^{i-1} - 1)$-entry heaps, add new entry, and down-heap

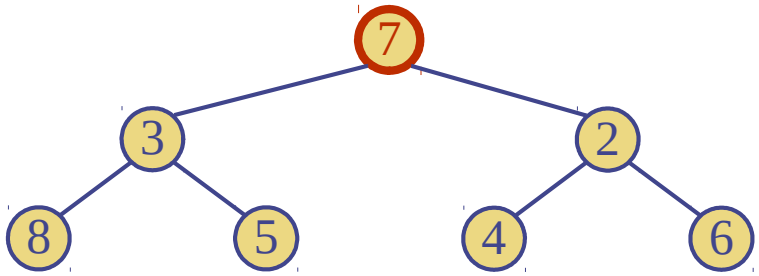
$\vdots$

$h+1$. Form the final heap with n entries: join two heaps with $(n-1)/2$ entries each, add new entry, and down-heap

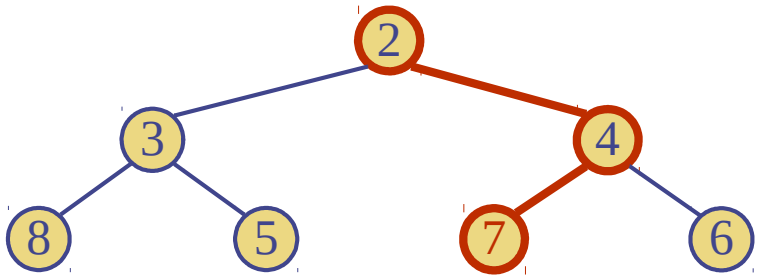
Merging Two Heaps

We are given two heaps and an entry e

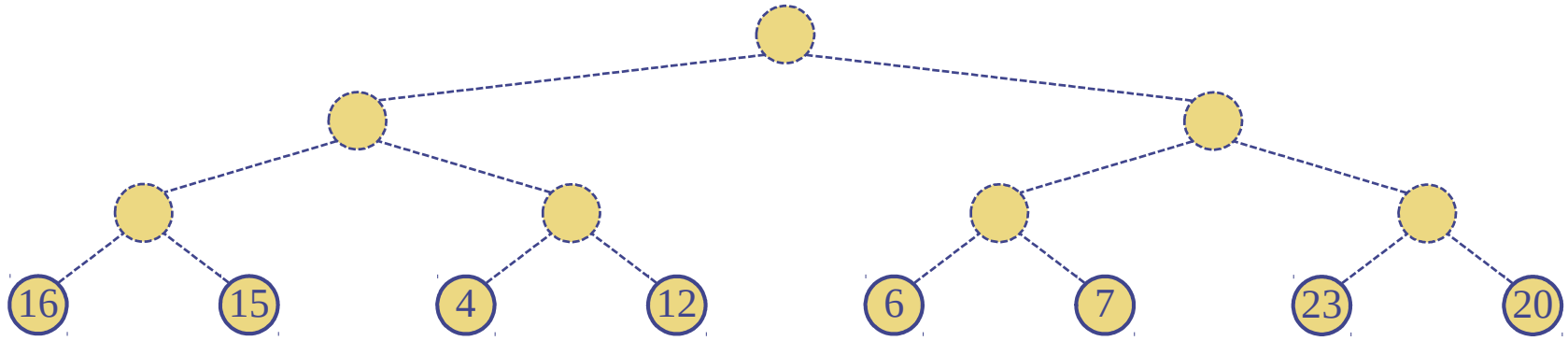
We create a new heap with the root node storing e and with the two heaps as subtrees

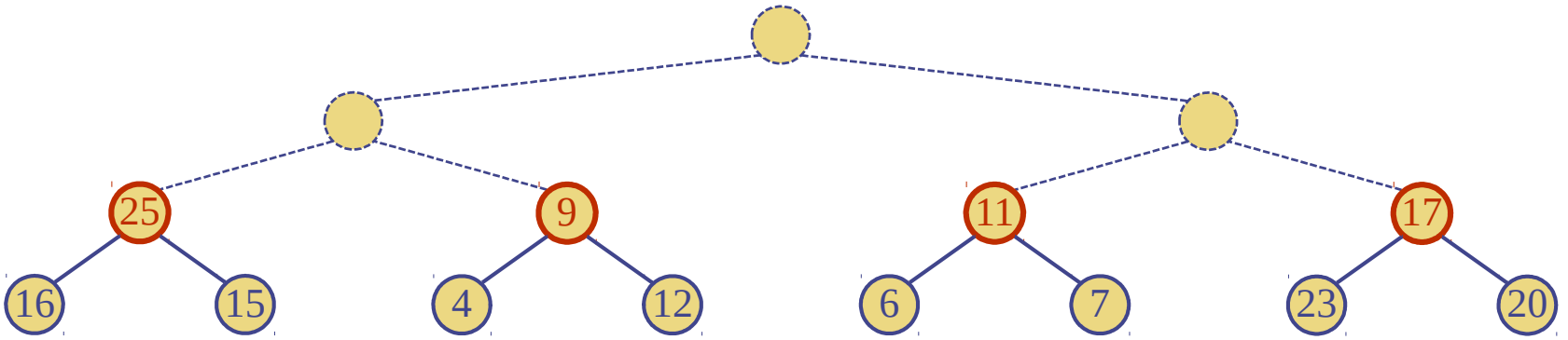


We perform down-heap to restore the heap-order property

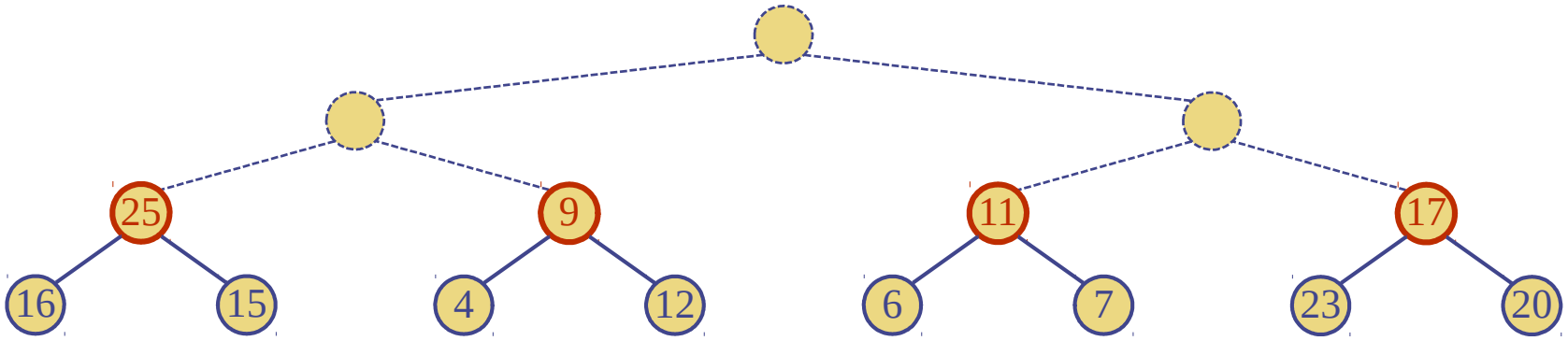


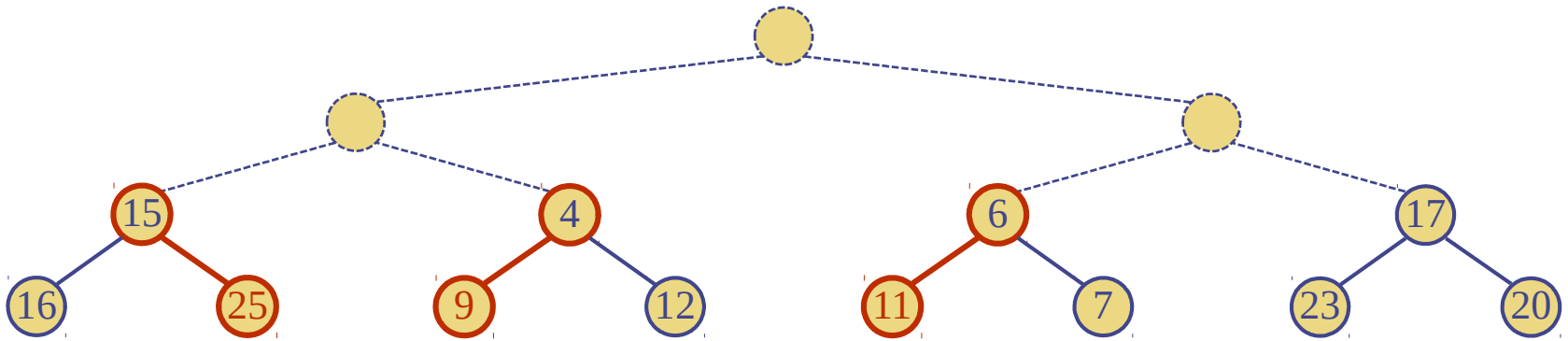
Example (steps 1 and 2.form)



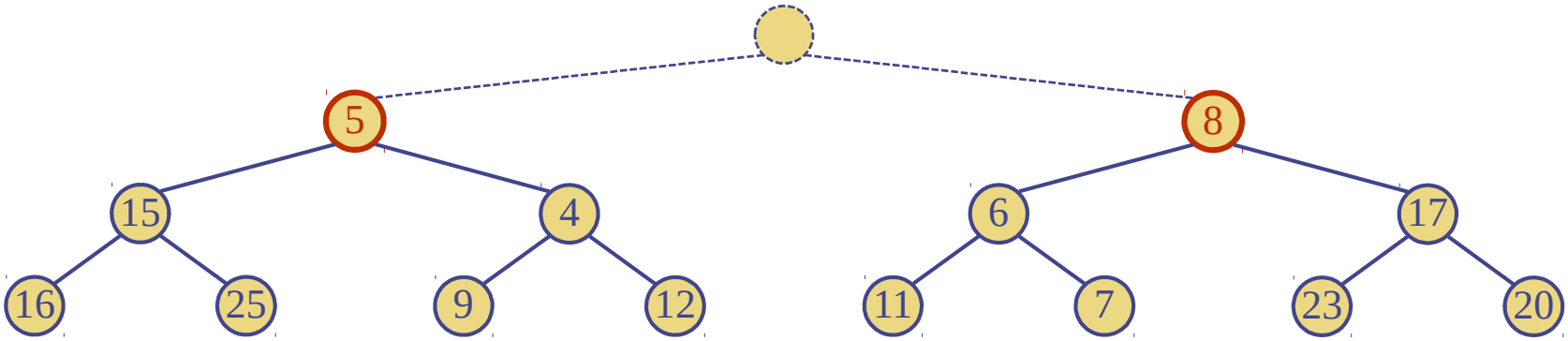


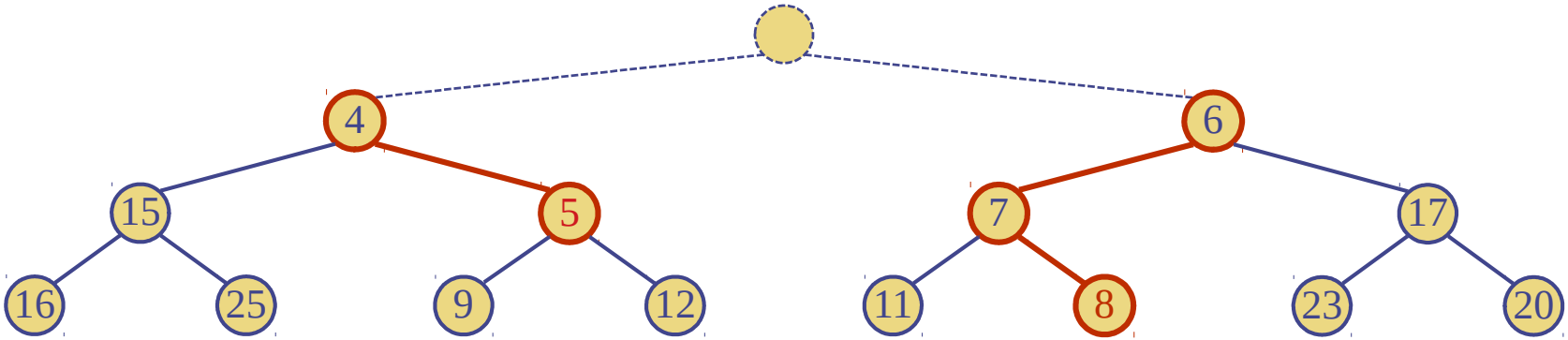
Example (step 2.down-heap)



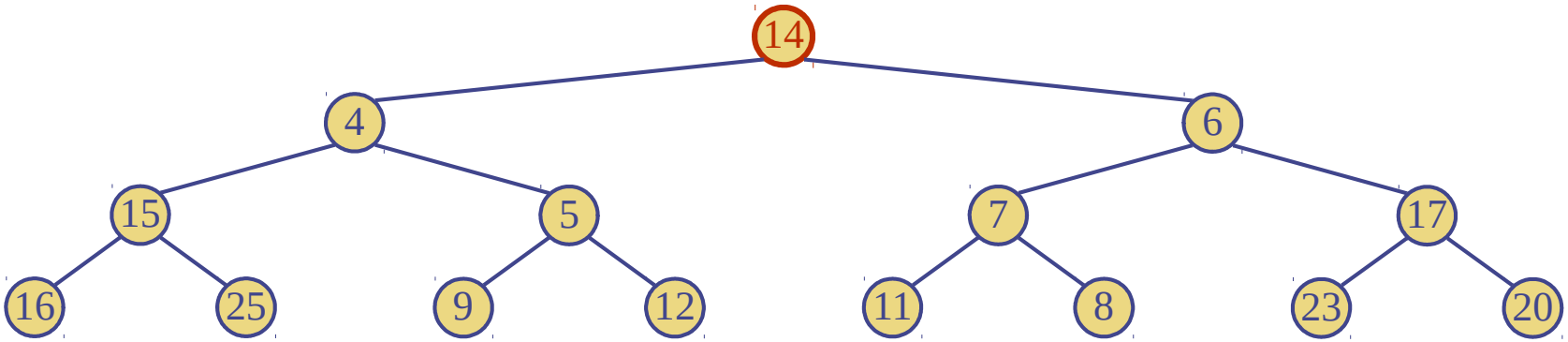


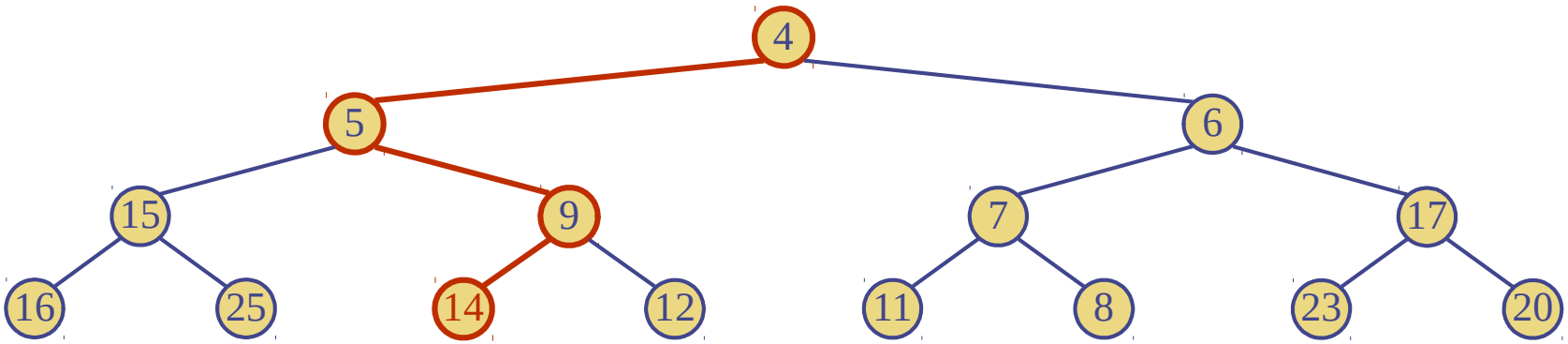
Example (steps 3.form and 3.down-heap)





Example (steps 4.form and 4.down-heap)





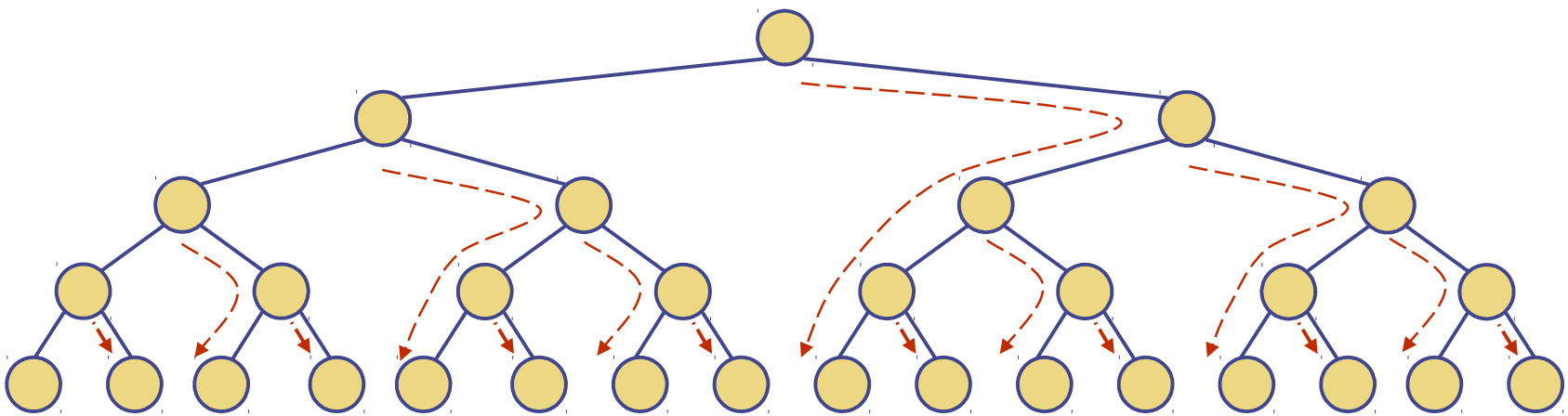
Analysis

We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)

Since each node is traversed by at most two proxy paths, 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 is faster than n successive insertions and speeds up the first phase of heap sort



Summary

Reading

Section 9.3 Heaps

Questions?