

Priority Queues



1

Introduction

- ◆ Priority Queue
 - Data structure for storing a collection of prioritized elements
 - Supporting arbitrary element insertion
 - Supporting removal of elements in order of priority

- ◆ So far, we covered “position-based” data structures
 - Stacks, queues, dequeues, lists, and even lists
 - Store elements at specific positions (linear or hierarchical)
 - Insertion and removal based on “position” (linear or hierarchical)
 - But, priority queue
 - ◆ Insertion and removal: priority-based

- ◆ Question: how to express the priority of an element
 - **Key** (example: your student id)

2

Priority Queue ADT

- ◆ A priority queue stores a collection of entries
- ◆ Typically, an **entry** is a pair (key, value), where the key indicates the priority
- ◆ Main methods of the Priority Queue ADT
 - **insert(e)**
inserts an entry e (with an implicit associated key value)
 - **removeMin()**
removes the entry with smallest key
- ◆ Additional methods
 - **min()**
returns, but does not remove, an entry with smallest key
 - **size(), empty()**
- ◆ Applications:
 - Standby flyers
 - Auctions
 - Stock market

3

Total Order Relations (a topic of Discrete Math)

- ◆ Keys in a priority queue can be arbitrary objects on which an order is defined
- ◆ Two distinct entries in a priority queue can have the same key
- ◆ Total ordering
 - Comparison rule should be defined for every pair of keys
- ◆ Mathematical concept of total order relation \leq
 - Reflexive property:
 $x \leq x$
 - Antisymmetric property:
 $x \leq y \wedge y \leq x \Rightarrow x = y$
 - Transitive property:
 $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- ◆ Satisfying the above three properties ensures:
 - Never leading to a comparison contradiction

4

Example: Total order & Partial order

- ◆ 2D points with (x-coordinate, y-coordinate)
 - Define relation ' \geq ' based on x-first, and y-next
 - $(4,3) \geq (3,4)$, $(3,5) \geq (3,4)$
 - Total ordering

 - What about defining relation ' \geq ' based on both x and y
 - $(4,3) \geq (2,1)$, but $(4,3) ??? (3,4)$
 - Partial ordering
 - ◆ Comparison not defined for some objects

- ◆ We assume that we define a comparison that leads to total ordering.

5

Priority Queue Sorting

- ◆ We can use a priority queue to sort a set of comparable elements
 1. Insert the elements one by one with a series of **insert** operations
 2. Remove the elements in sorted order with a series of **removeMin** operations

- ◆ The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort(S, C)*

Input sequence S , comparator C for the elements of S

Output sequence S sorted in increasing order according to C

$P \leftarrow$ priority queue with comparator C

while $\neg S.empty()$

$e \leftarrow S.front(); S.eraseFront()$

$P.insert(e, \emptyset)$

while $\neg P.empty()$

$e \leftarrow P.removeMin()$

$S.insertBack(e)$

6

Sequence-based Priority Queue

◆ Implementation with an unsorted list



◆ Implementation with a sorted list



◆ Performance:

- **insert** takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
- **removeMin** and **min** take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

◆ Performance:

- **insert** takes $O(n)$ time since we have to find the place where to insert the item
- **removeMin** and **min** take $O(1)$ time, since the smallest key is at the beginning

7

Selection-Sort

◆ Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

◆ Running time of Selection-sort:

1. Inserting the elements into the priority queue with n **insert** operations takes $O(n)$ time
2. Removing the elements in sorted order from the priority queue with n **removeMin** operations takes time proportional to

$$1 + 2 + \dots + n$$

◆ Selection-sort runs in $O(n^2)$ time

8

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

9

Insertion-Sort

- ◆ Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- ◆ Running time of Insertion-sort:
 1. Inserting the elements into the priority queue with n **insert** operations takes time proportional to
$$1 + 2 + \dots + n$$
 2. Removing the elements in sorted order from the priority queue with a series of n **removeMin** operations takes $O(n)$ time
- ◆ Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
⋮	⋮	⋮
(g)	(2,3,4,5,7,8,9)	()

Comparator

Another design method



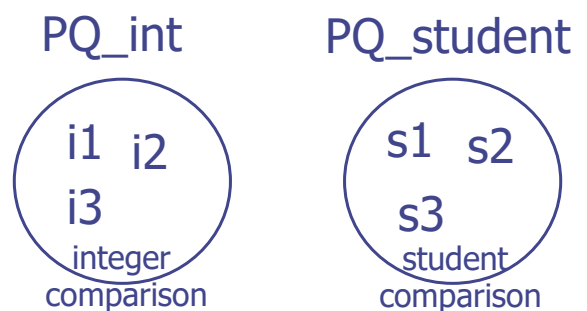
How to define order for any object? (1)

- ◆ Integer, float, double
 - Quite clear on how to define “order”
- ◆ Student: id, sex, department
 - S1 is less than S2? In what sense?
- ◆ Flight Passengers: airplane number, seat number, sex
 - P1 is less than P2? In what sense?
- ◆ How to design “comparison logic” in a programming language?
- ◆ What design is good?

13

Design 1: Separate Design

- ◆ Different Priority Queue based on the element type and the manner of comparing elements
- ◆ PQ_Int, PQ_Student, PQ_XXX
- ◆ Simple, but not general
- ◆ Many copies of the same code



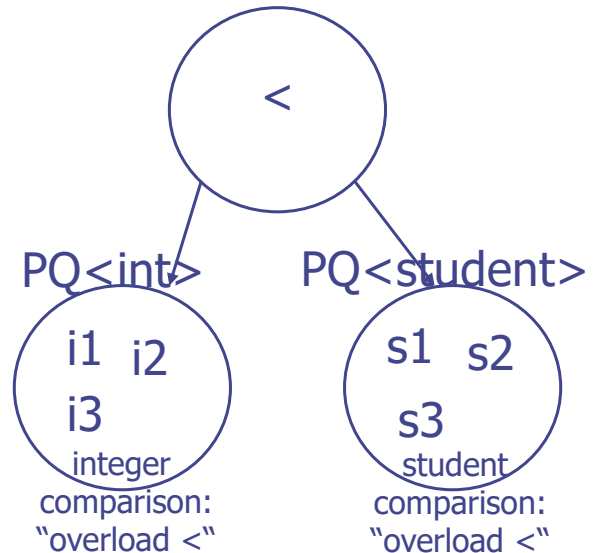
14

Design 2: Template and Overloading (2)

```
bool operator<(const Point2D& p, const Point2D& q) {  
    if (p.getX() == q.getX())    return p.getY() < q.getY();  
    else                          return p.getX() < q.getX();  
}
```

PQ template

- ◆ General enough for many situations
- ◆ But,
 - Cannot have multiple comparison methods for the same type
 - What about comparison based on y-first, and x-next?
- ◆ Even for the same data type, we want to apply different comparison methods A or B, depending on the situations



15

Design 3: Separating Comparator (1)

- ◆ 2D points: Point2D p, Point 2: q
 - Sometimes we want either of X-based comparison, Y-based comparison
- ◆ Idea
 - Define a comparator class, e.g., “LeftRight” (x-based) and “BottomTop” (y-based)
 - Overload “()” operator

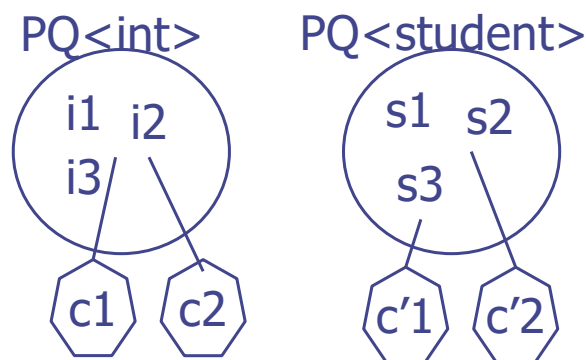
```
class LeftRight { // a left-right comparator  
public:  
    bool operator()(const Point2D& p, const Point2D& q) const  
    { return p.getX() < q.getX(); }  
};  
  
class BottomTop { // a bottom-top comparator  
public:  
    bool operator()(const Point2D& p, const Point2D& q) const  
    { return p.getY() < q.getY(); }  
};
```

16

Design 3: Separating Comparator (2)

```
Point2D p(1.3, 5.7), q(2.5, 0.6);           // two points
LeftRight leftRight;                       // a left-right comparator
BottomTop bottomTop;                       // a bottom-top comparator
printSmaller(p, q, leftRight);             // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);             // outputs: (2.5, 0.6)
```

```
template <typename E, typename C>          // element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
    cout << (isLess(p, q) ? p : q) << endl; // print the smaller of p and q
}
```



17

In C++

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

// sort using a custom function object
struct MyLess{
    bool operator()(int a, int b) const
    {
        return a > b;
    }
};

int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (int i=0 ; i<s.size();i++) {
        std::cout << s[i] << " ";
    }
    std::cout << '\n';

    MyLess myless;

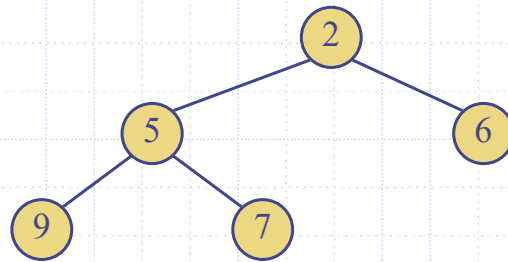
    std::sort(s.begin(), s.end(), myless);

    for (int i=0 ; i<s.size();i++) {
        std::cout << s[i] << " ";
    }
    std::cout << '\n';
}
```

```
[yi@iMacyung ~/tmp]# ./a.out
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

18

Heaps



19

Recall Priority Queue ADT

- ◆ A priority queue stores a collection of entries
- ◆ Typically, an **entry** is a pair (key, value), where the key indicates the priority
- ◆ Main methods of the Priority Queue ADT
 - **insert(e)** inserts an entry e
 - **removeMin()** removes the entry with smallest key
- ◆ Additional methods
 - **min()** returns, but does not remove, an entry with smallest key
 - **size(), empty()**
- ◆ Applications:
 - Standby flyers
 - Auctions
 - Stock market

Recall PQ Sorting



- ◆ We use a priority queue
 - Insert the elements with a series of **insert** operations
 - Remove the elements in sorted order with a series of **removeMin** operations
- ◆ The running time depends on the priority queue implementation:
 - Unsorted sequence gives selection-sort: $O(n^2)$ time
 - Sorted sequence gives insertion-sort: $O(n^2)$ time
- ◆ Can we do better? Balancing the above

Algorithm *PQ-Sort(S, C)*

Input sequence S , comparator C for the elements of S

Output sequence S sorted in increasing order according to C

$P \leftarrow$ priority queue with comparator C

while $\neg S.empty()$

$e \leftarrow S.front(); S.eraseFront()$

$P.insert(e, \emptyset)$

while $\neg P.empty()$

$e \leftarrow P.removeMin()$

$S.insertBack(e)$

21

We will have these results soon ...

Sequence-based

<i>Operation</i>	<i>Unsorted List</i>	<i>Sorted List</i>
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Heap-based

<i>Operation</i>	<i>Time</i>
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Key: Where were the “unnecessary repetitions” and “stupidity”?

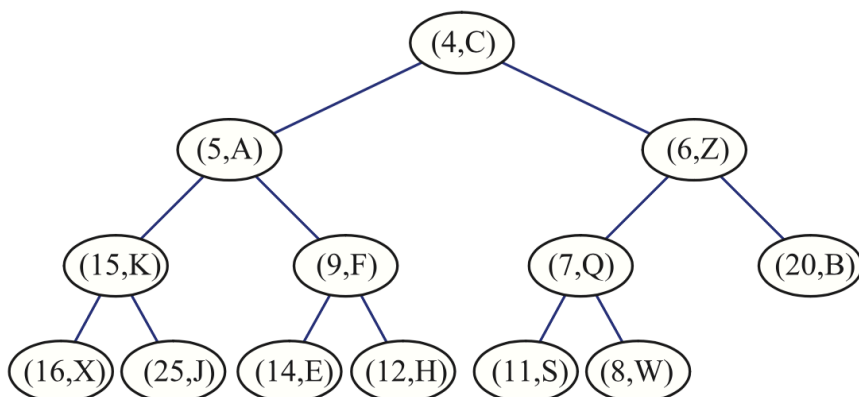
2. Complete binary tree property

◆ Complete Binary Tree

- Roughly speaking, every level, except for the last level, is completely filled, and all nodes in the last level are as far left as possible.

◆ let h be the height of the heap

- for $i = 0, \dots, h - 1$, there are 2^i nodes of depth i
- at depth $h - 1$, the internal nodes are to the left of the external nodes



25

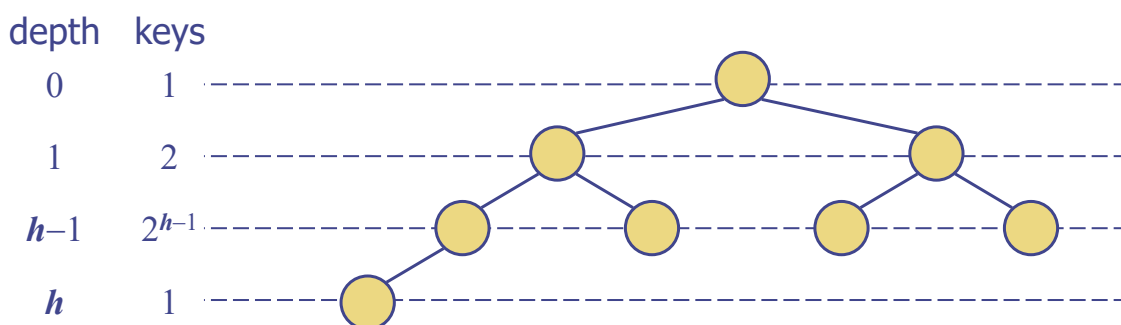
Height of a Heap of n elements



◆ Theorem: A heap storing n keys has height $O(\log n)$

Proof: (we apply the complete binary tree property)

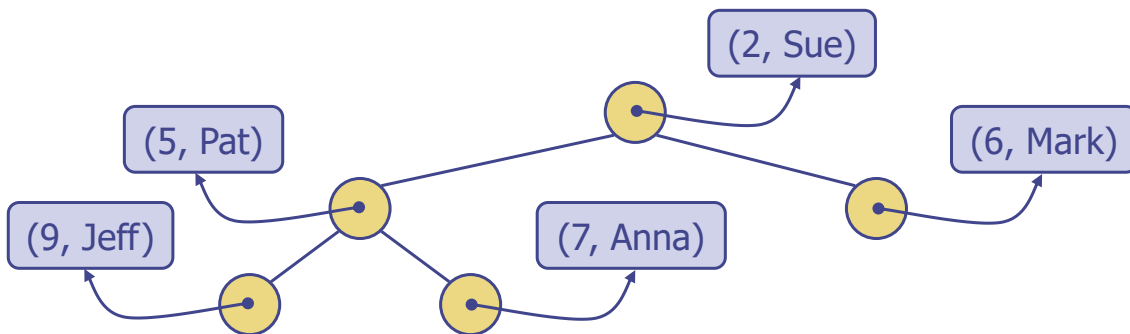
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h - 1$ and at least one key at depth h , we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$



26

Heaps and Priority Queues

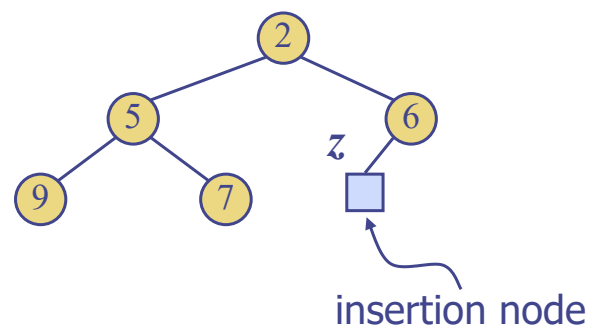
- ◆ We can use a heap to implement a priority queue
 - We say “heap-based PQ implementation”
- ◆ We store a (key, element) item at each internal node
- ◆ We keep track of the position of the last node
 - I am able to know who is the last node in $O(1)$ time
 - Easy



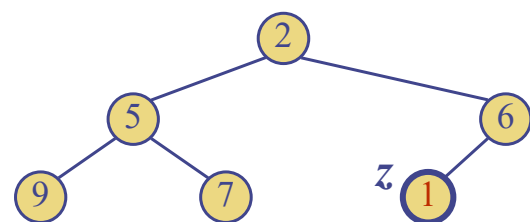
27

Insertion into a Heap

- ◆ Method **insert** of the priority queue ADT corresponds to the insertion of a key k to the heap



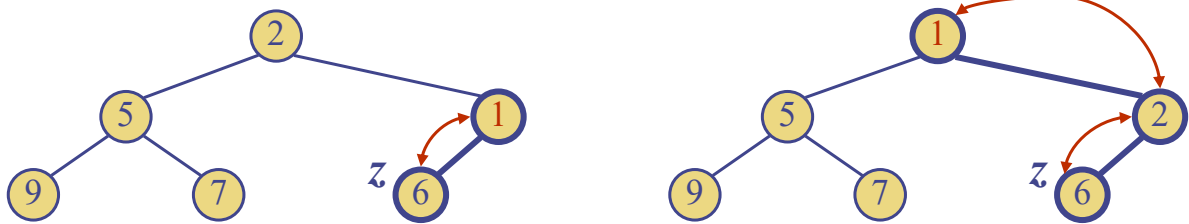
- ◆ The insertion algorithm consists of three steps
 - Find the **insertion node** z (the new last node)
 - ◆ How? discussed later
 - Store k at z
 - Restore the heap-order property (discussed next)



28

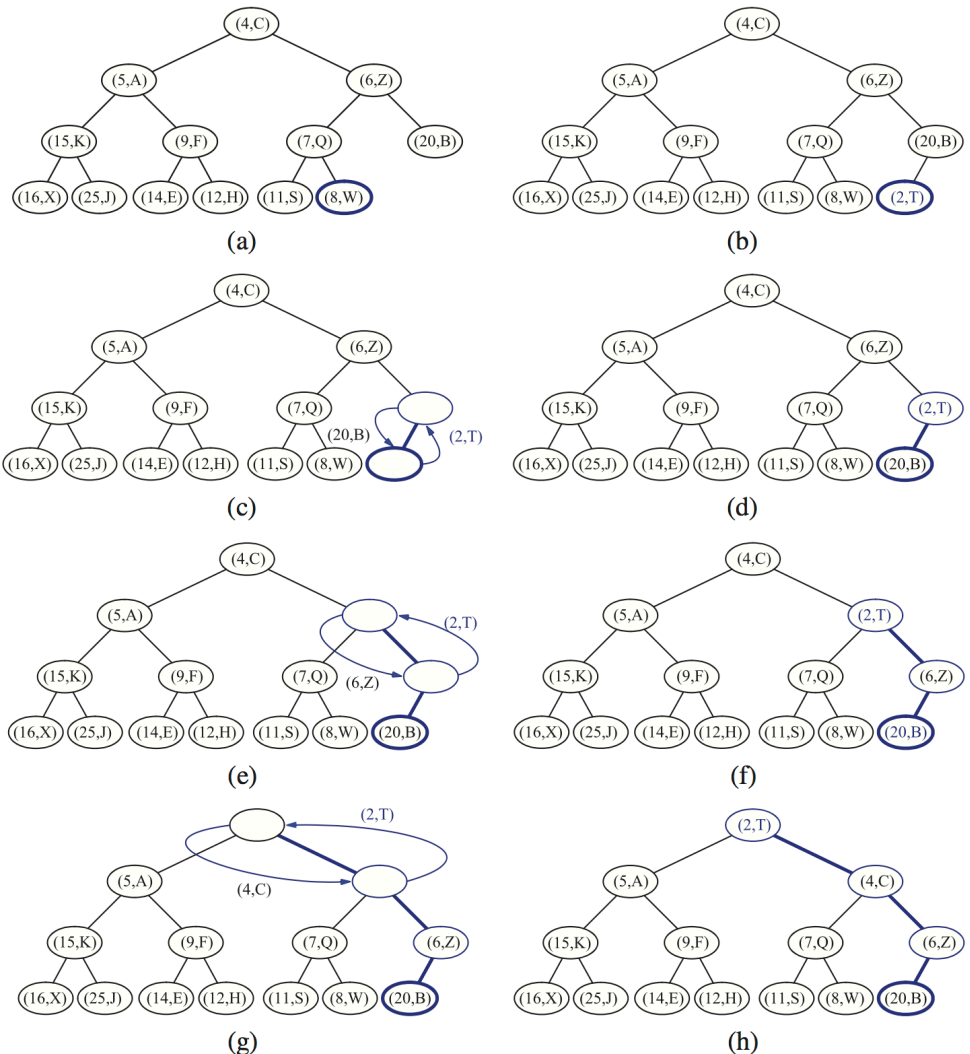
Upheap

- ◆ After the insertion of a new key k , the heap-order property may be violated
- ◆ Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- ◆ Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- ◆ Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



29

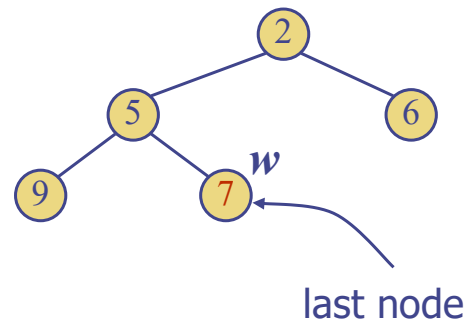
Insert: (2,T)



30

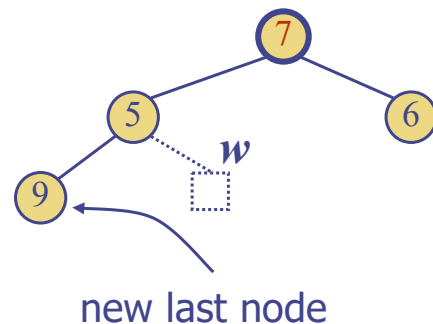
Removal from a Heap

◆ Method **removeMin** of the priority queue ADT corresponds to the removal of the root key from the heap



◆ The removal algorithm consists of three steps

- Replace the root key with the key of the last node w
- Remove w
- Restore the heap-order property (discussed next)



31

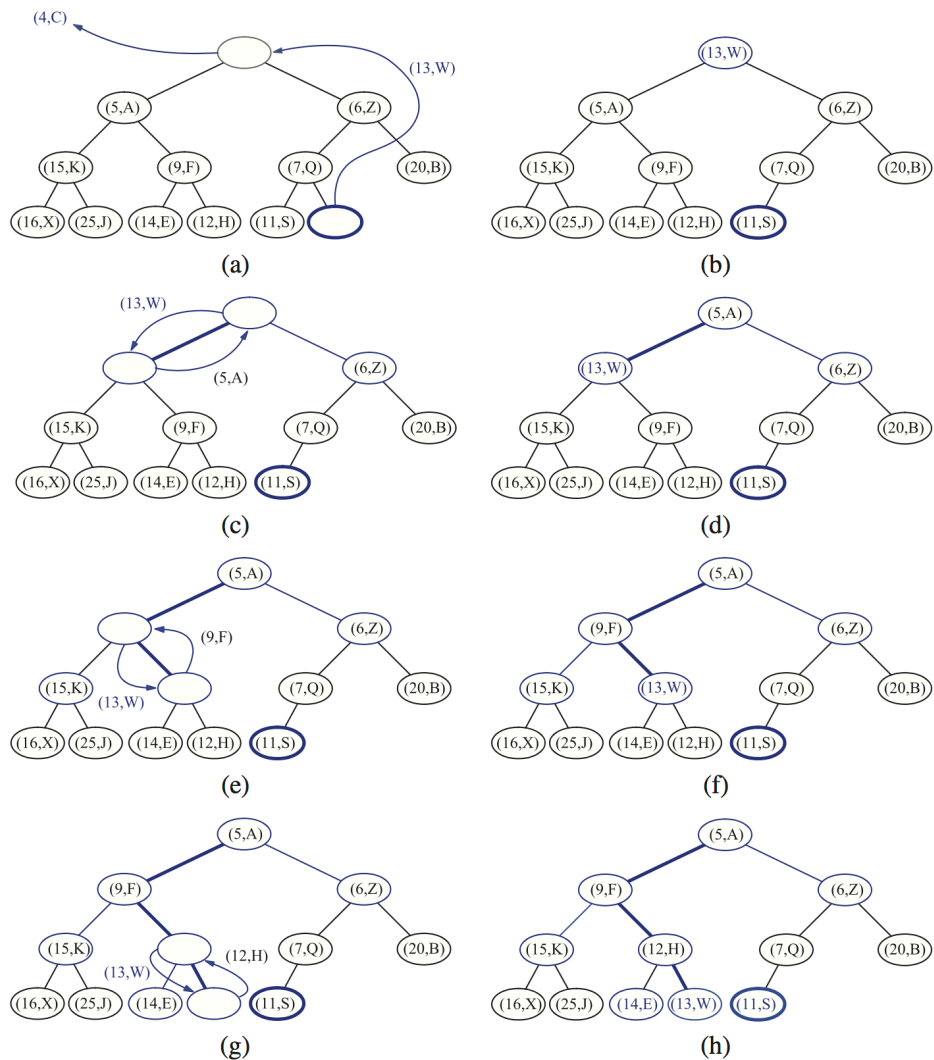
Downheap

- ◆ After replacing the root key with the key k of the last node, the heap-order property may be violated
- ◆ Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root (but which path?)
- ◆ Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- ◆ Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



32

removeMin

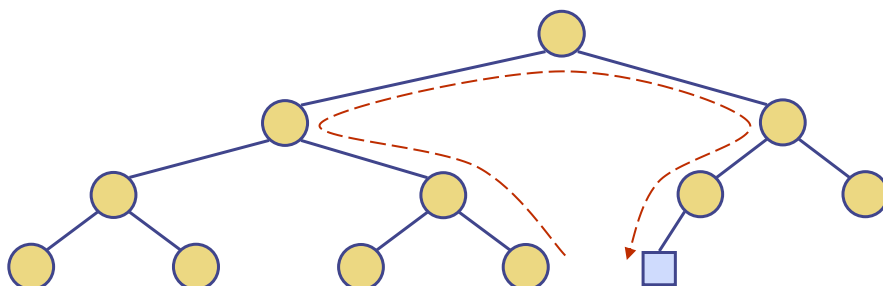


33

Updating the Last Node

- ◆ How can we find the insertion node (a new last node)?
 - The insertion node can be found by traversing a path of $O(\log n)$ nodes
 - (1) Go up until a left child or the root is reached
 - (2) If a left child is reached, go to the right child
 - (3) Go down left until a leaf is reached

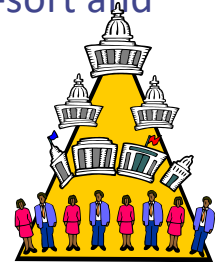
- ◆ Similar algorithm for updating the last node after a removal



34

Heap-Sort

- ◆ Consider a priority queue with n items implemented by means of a heap
 - the space used is $O(n)$
 - methods **insert** and **removeMin** take $O(\log n)$ time
 - methods **size**, **empty**, and **min** take time $O(1)$ time
- ◆ Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
 - Construction: n insertions
 - Actual sorting: n removals
- ◆ The resulting algorithm is called heap-sort
- ◆ Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort



35

Sequence-based vs. Heap-based

Sequence-based

<i>Operation</i>	<i>Unsorted List</i>	<i>Sorted List</i>
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Heap-based

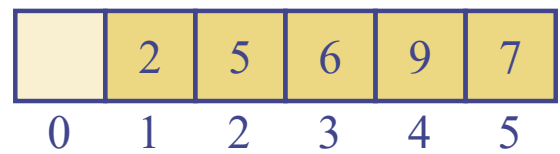
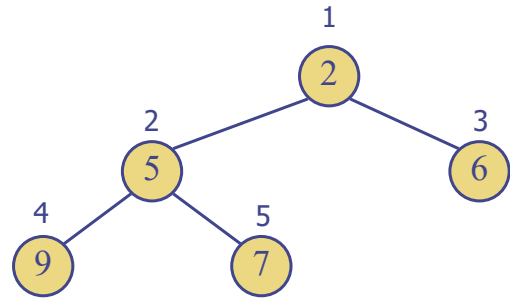
<i>Operation</i>	<i>Time</i>
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

How do we remove “stupid repetition”?

36

Vector-based Heap Implementation

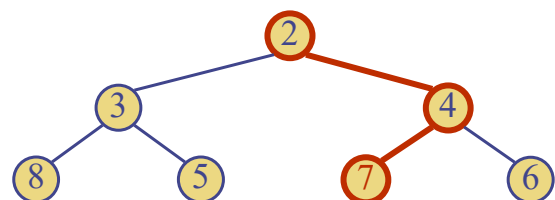
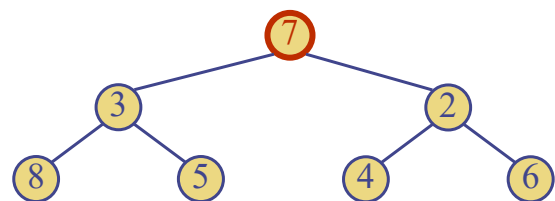
- ◆ We can represent a heap with n keys by means of a vector of length $n + 1$
- ◆ For the node at rank i
 - the left child is at rank $2i$
 - the right child is at rank $2i + 1$
- ◆ Links between nodes are not explicitly stored
- ◆ The cell of at rank 0 is not used
- ◆ Operation insert corresponds to inserting at rank $n + 1$
- ◆ Operation removeMin corresponds to removing at rank n



37

Merging Two Heaps

- ◆ We are given two heaps and a key k
- ◆ We create a new heap with the root node storing k and with the two heaps as subtrees
- ◆ We perform downheap to restore the heap-order property



38

Questions?