Lecture 7: Priority Queues and Heaps

(Chapter 8, Sections 8.1, 8.2, and 8.3 from the book)

Agenda

- Priority Queues
  - Concepts – priority queue, entry, key, total order relation, comparator
  - Priority queue implementation with a list
  - Priority queue sorting
    - Selection-sort
    - Insertion-sort
- Heaps
  - Priority queue implementation with a heap
  - Insertion into/Deletion from a heap
  - Heap-sort

Priority Queues

Priority Queue – Definition

- A priority queue is a collection of elements, called values, each having an associated key that is provided at the time the element is inserted.
- A key-value pair, (key,value), inserted into a priority queue is called an entry of the priority queue
- The name "priority queue" comes from the fact that keys determine the priority used to pick entries to be removed

Keys in Priority Queue

- Keys are parameters or properties according to which we compare the objects; Keys are assigned for each object in a collection
  - e.g. we can compare companies by earnings or by number of employees
- Formally: a key is an object that is assigned to an element as a specific attribute for that element, which can be used to identify or weigh that element.
- Keys in a priority queue can be arbitrary objects on which an order is defined
- Note! Two distinct entries in a priority queue can have the same key

Total Order Relations

- 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:
    $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Comparison rule that satisfies these three properties will never lead to a comparison contradiction. Such a rule defines a linear ordering relationship among a set of keys.

Priority Queue ADT

- Methods of the Priority Queue ADT
  - insert(k, x): inserts into priority queue P an entry with key k and value x; return the inserted entry; an error occurs if k is invalid (that is, k cannot be compared with other keys);
  - removeMin(): removes from P and returns an entry with smallest key; an error condition occurs if P is empty;
  - min(): returns, but does not remove, an entry of P with smallest key; an error condition occurs if P is empty;
  - size(): returns the number of entries in priority queue P;
  - isEmpty(): tests whether priority queue P is empty;
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue ADT – Java Interface

```
public interface PriorityQueue<K,V>
{
    public int size();
    public boolean isEmpty();
    public Entry<K,V> min() throws
EmptyPriorityQueueException;
    public Entry<K,V> insert(K key, V
value) throws InvalidKeyException;
    public Entry<K,V> removeMin() throws
EmptyPriorityQueueException;
}
```

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry
- Java Interface for a key-value pair entry

```
public interface Entry<K,V> {
  public   K getKey();
  public   V getValue();
 }
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A comparator is an object that compares two keys
- The comparator is external to the keys being compared
- A generic priority queue uses a default comparator
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT:
  - compare(x, y): returns an integer i such that
    - $i < 0$ if $x < y$,
    - $i = 0$ if $x = y$
    - $i > 0$ if $x > y$
    - An error occurs if a and b cannot be compared.
  - equals() compares a comparator to other comparator

Comparator – Java Interface

```
public interface Comparator
{
public int compare(Object o1, Object o2);
public boolean equals(Object obj);
}
```

Exercise 1 – Priority Queue ADT

- Starting from an empty priority queue, show the output and priority queue after each of the following operations:
- insert(7,A); insert(3,G); removeMin(); size(); insert(5,S), insert(4,T), removeMin(); min(); insert(ALA, W)

Exercise 1 – Priority Queue ADT – Answer

Operation;	Output;	Priority Queue
insert(7,A);	(7,A)	(7,A)
insert(3,G);	(3,G)	(7,A); (3,G)
removeMin();	(3,G)	(7,A)
size();	1	(7,A)
insert(5,S),	(5,S)	(7,A); (5,S)
insert(4,T),	(4,T)	(7,A); (5,S); (4,T)
removeMin();	(4,T)	(7,A); (5,S)
min();	(5,S)	(7,A); (5,S)
insert(ALA, W)	Error	(7,A); (5,S)

Sequence-based Priority Queue

- Implementation with an unsorted 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

- Implementation with a sorted list

- 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

- size and isEmpty take $O(1)$ in both cases

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Put the elements of sequence S into initially empty priority queue P by means of a series of insert operations, one for each element.
  2. Remove the elements in sorted order from the priority queue P with a series of removeMin operations, putting them back into S in order.
- The running time of this sorting method depends on the priority queue implementation

Priority Queue Sorting

Algorithm *PriorityQueueSort*(*S*, *P*)

  Input sequence *S* storing *n* elements, on which a total order relation is defined; priority queue *P*, that compares keys using the same total order relation

  Output sequence *S* sorted by the total order relation

  while !*S.isEmpty* () **do**

    e ← *S.removeFirst* ()

    P.insert (*e*, ∅) {a null value is used}

  while !*P.isEmpty*() **do**

    e ← *P.removeMin*().*getKey*()

    S.addLast(*e*) {the smallest key in *P* is added to the end of *S*}

Selection–Sort

- Selection-sort is the variation of *PriorityQueueSort* 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
     $$O(n+(n-1)+\ldots+2+1)=O(n(n+1)/2)=O(n^2)$$
- Selection-sort runs in $O(n^2)$ time

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)	()

Insertion-Sort

- Insertion-sort is the variation of *PriorityQueueSort* 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

     $$O(n+(n-1)+\ldots+2+1)=O(n(n+1)/2)=O(n^2)$$

  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)	()

Exercise 2 – Selection-Sort

- Illustrate the execution of the selection-sort algorithm on the following input sequence:

  (23,98,12,99,1,78,9)

Exercise 2 – Selection-Sort – Answer

	Sequence S	Priority queue P
Input:	(23,98,12,99,1,78,9)	()
Phase 1		
(a)	(98,12,99,1,78,9)	(23)
(b)	(12,99,1,78,9)	(23,98)
(c)	(99,1,78,9)	(23,98,12)
(d)	(1,78,9)	(23,98,12,99)
(e)	(78,9)	(23,98,12,99,1)
(f)	(9)	(23,98,12,99,1,78)
(g)	()	(23,98,12,99,1,78,9)
Phase 2		
(a)	(1)	(23,98,12,99,78,9)
(b)	(1,9)	(23,98,12,99,78)
(c)	(1,9,12)	(23,98,99,78)
(d)	(1,9,12,23)	(98,99,78)
(e)	(1,9,12,23,78)	(98,99)
(f)	(1,9,12,23,78,98)	(99)
(g)	(1,9,12,23,78,98,99)	()

Exercise 3 – Insertion-Sort

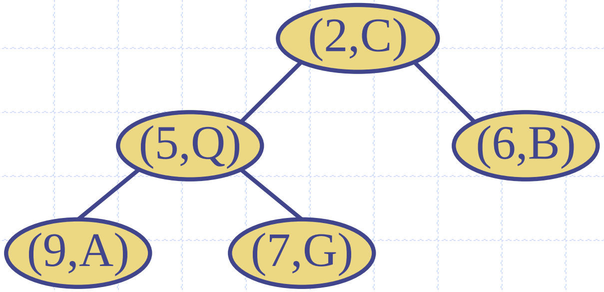
- Illustrate the execution of the insertion-sort algorithm on the following input sequence:

  (23,98,12,99,1,78,9)

Exercise 3 – Insertion-Sort – Answer

	Sequence S	Priority queue P
Input:	(23,98,12,99,1,78,9)	()
Phase 1		
(a)	(98,12,99,1,78,9)	(23)
(b)	(12,99,1,78,9)	(23,98)
(c)	(99,1,78,9)	(12,23,98)
(d)	(1,78,9)	(12,23,98,99)
(e)	(78,9)	(1,12,23,98,99)
(f)	(9)	(1,12,23,98,99)
(g)	()	(1,9,12,23,78,98,99)
Phase 2		
(a)	(1)	(9,12,23,78,98,99)
(b)	(1,9)	(12,23,78,98,99)
(c)	(1,9,12)	(23,78,98,99)
(d)	(1,9,12,23)	(78,98,99)
(e)	(1,9,12,23,78)	(98,99)
(f)	(1,9,12,23,78,98)	(99)
(g)	(1,9,12,23,78,98,99)	()

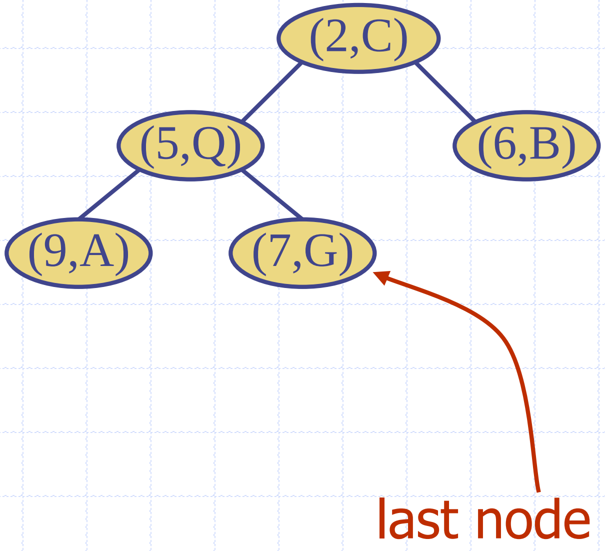
Heaps

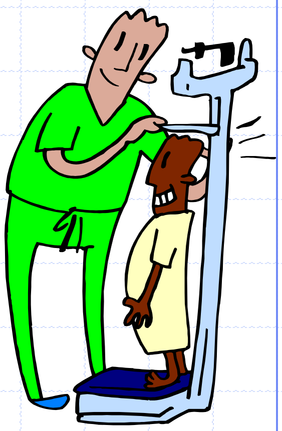


Heaps

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

- The last node of a heap is the rightmost node of maximum depth

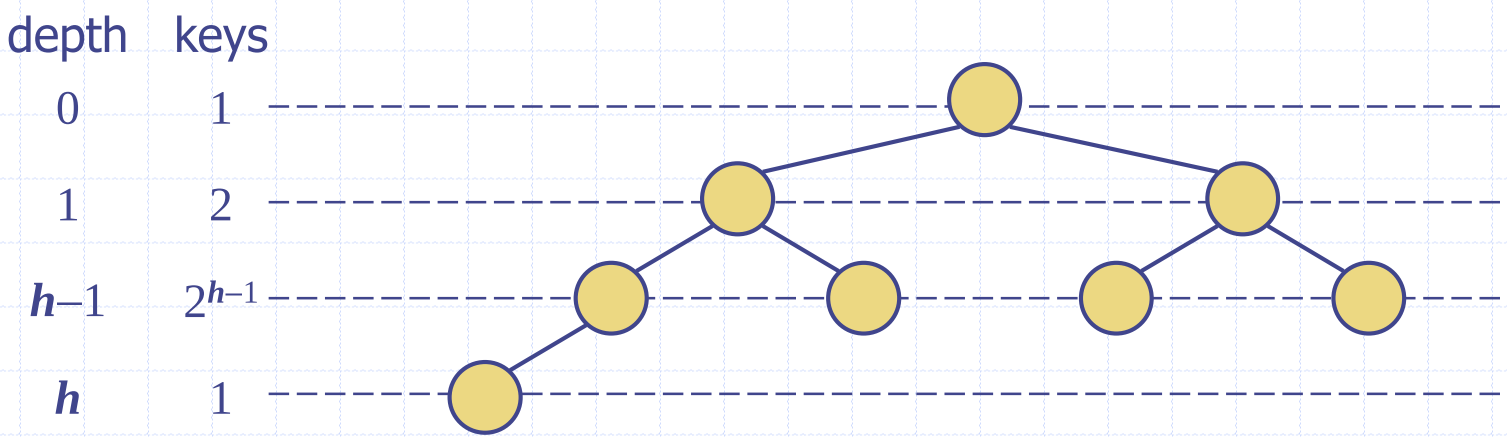




Height of a Heap

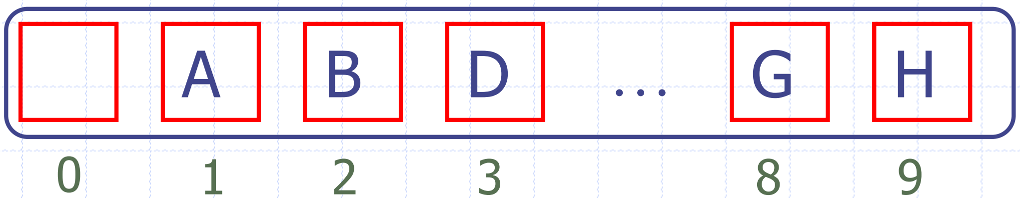
- **Theorem:** A heap storing n keys has height $O(\log n)$

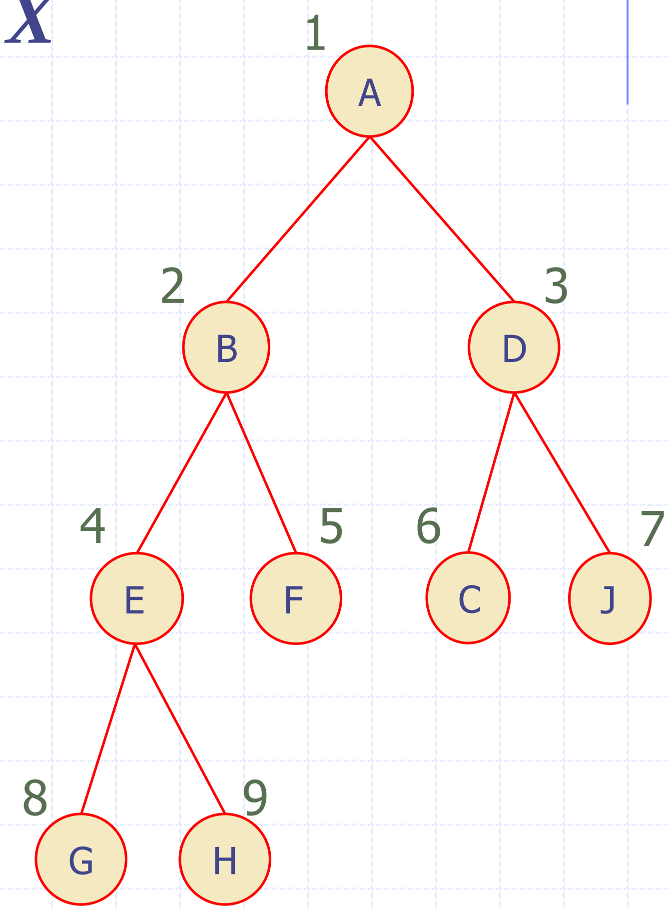
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Array-Based Representation of **Complete** Binary Trees

- Nodes are stored in an array X





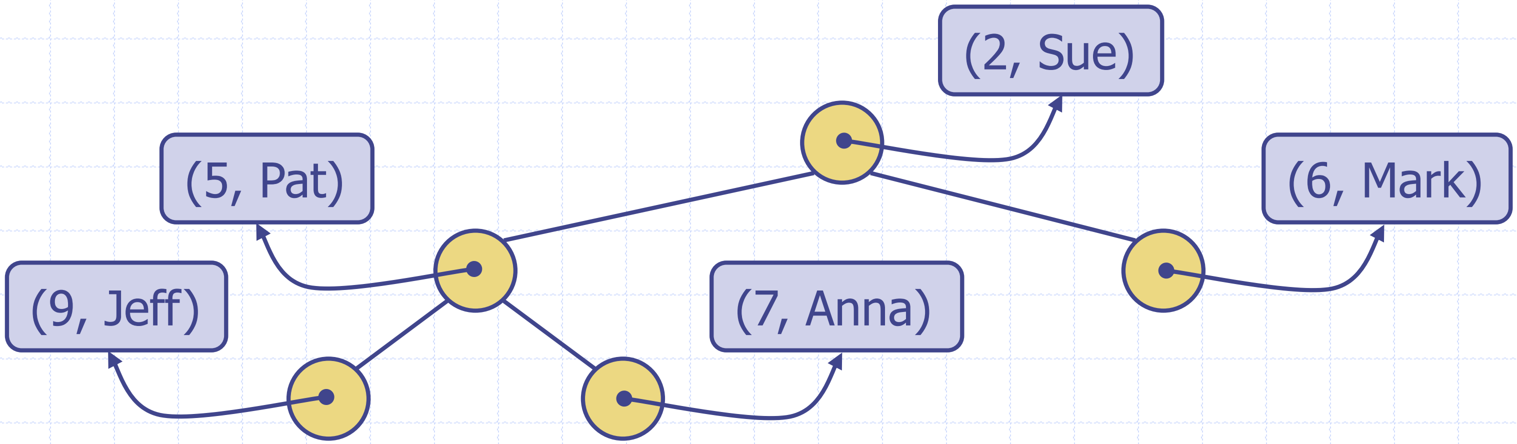
- Node v is stored at X[rank(v)]
  - rank(root) = 1
  - if node v is the left child of parent(v), rank(v) = 2 · rank(parent(v))
  - if node v is the right child of parent(v), rank(v) = 2· rank(parent(v)) + 1

Complete Binary Tree ADT

- Complete binary T supports all methods of binary tree ADT, plus the following:
  - add(o): add to T and return a new external node v storing element o such that the resulting tree is a complete binary tree with last node v
  - remove(): remove the last node of T and return its element

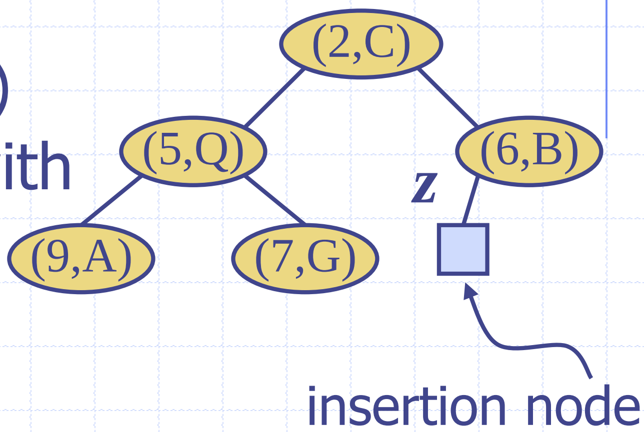
Heaps and Priority Queues

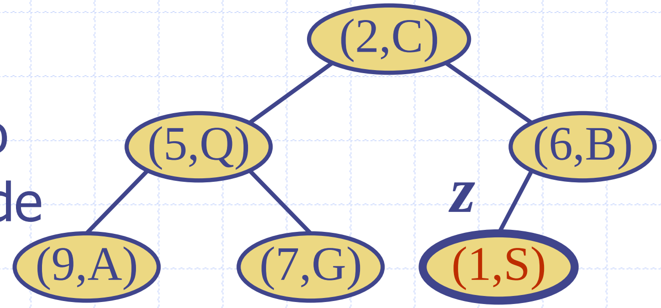
- We can use a heap to implement a priority queue
- We store one (key, value) item at one node
- We keep track of the position of the last node



Insertion into a Heap

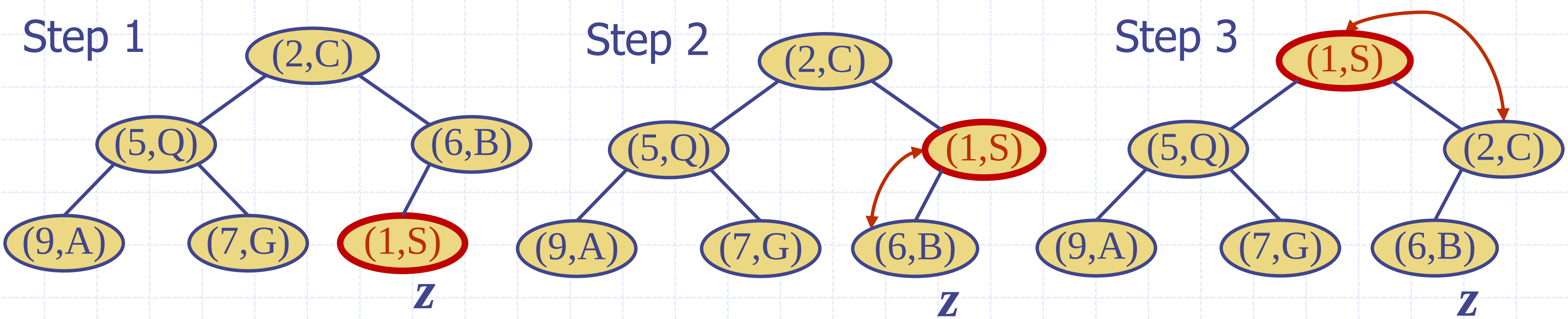
- Consider inserting entry $(\boldsymbol{k},\boldsymbol{x})$=(1, S) to the priority queue implemented with a heap $\boldsymbol{T}$
- The insertion algorithm (insert(k,x) from the priority queue ADT) is as follows:
  - Add a node z to T with operation add so that this new node becomes the last node of T and stores entry (k,x)
  - Restore the heap-order property that may be violated by the previous action (discussed next)





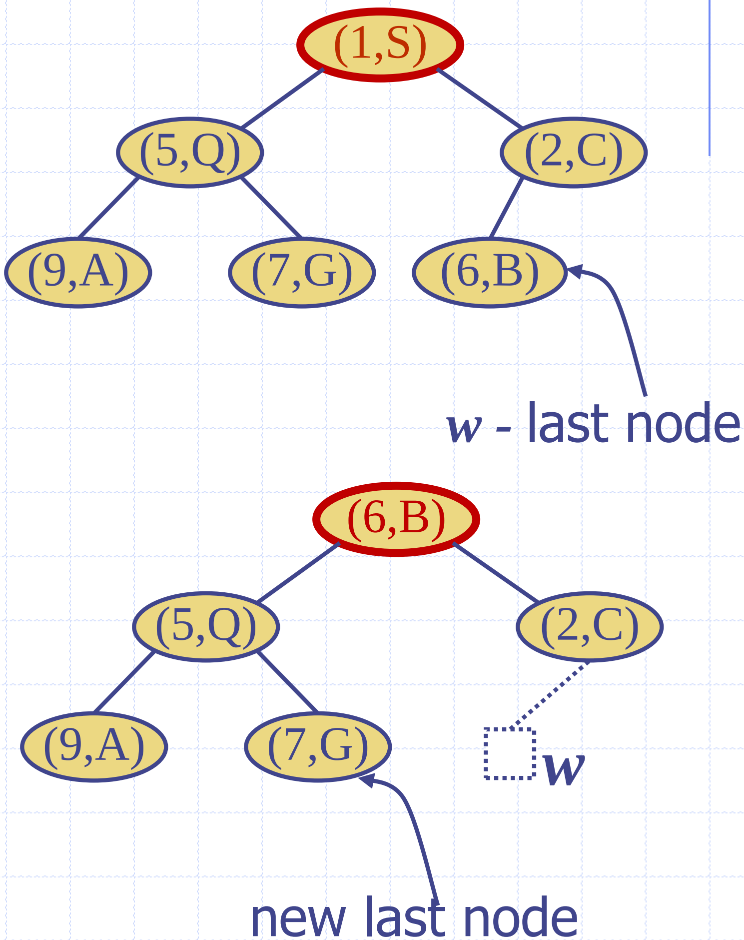
Up-heap bubbling

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



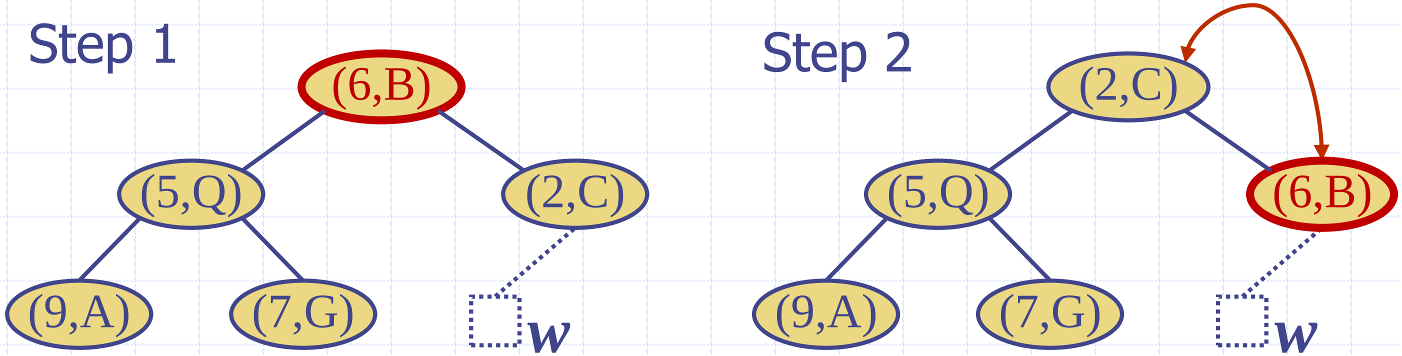
Removal from a Heap

- 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 element with the entry that is in the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (discussed next)



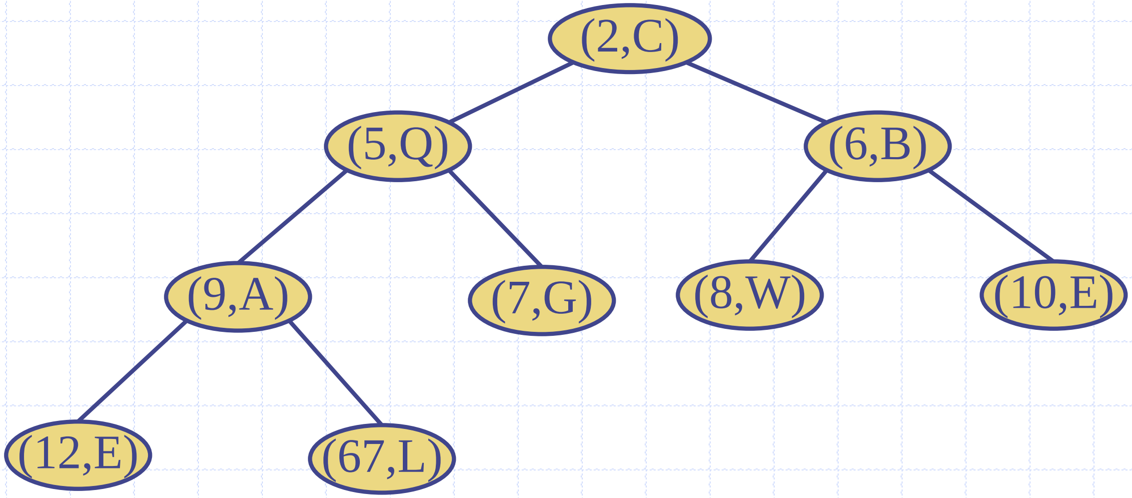
Down-heap bubbling

- After replacing the root element with the entry with key k of the last node, the heap-order property may be violated
- Algorithm down-heap restores the heap-order property by swapping entry with key k along a downward path from the root (swap the entry with key k with its child with the smallest key)
- Down-heap 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 $O(\log n)$, down-heap runs in $O(\log n)$ time



Exercise 4 – Insert

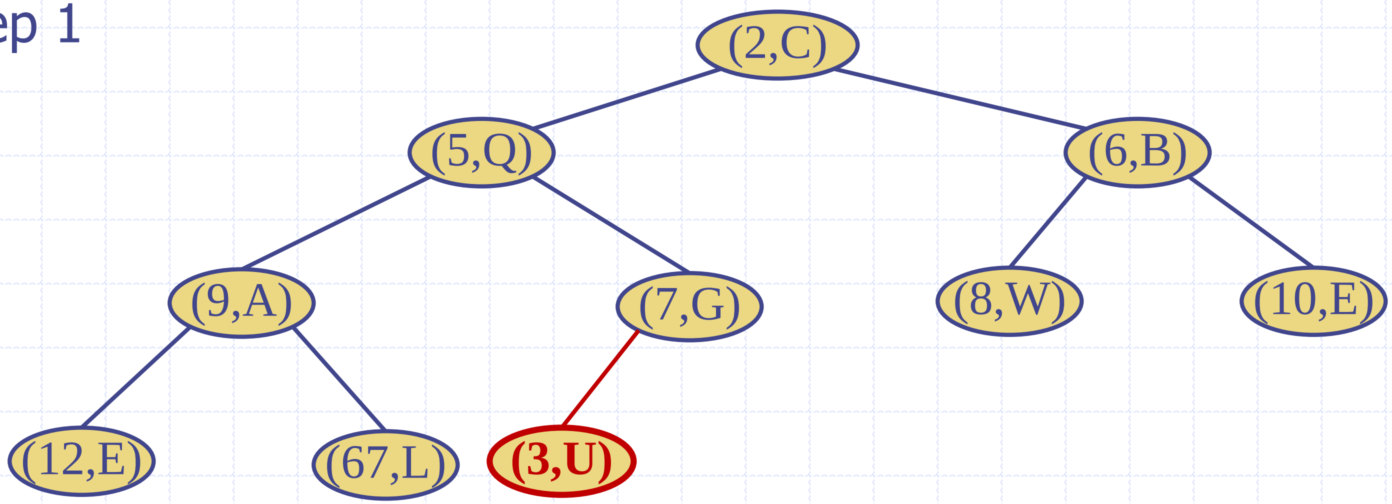
- Insert entry (3,U) into the priority queue implemented with a heap:



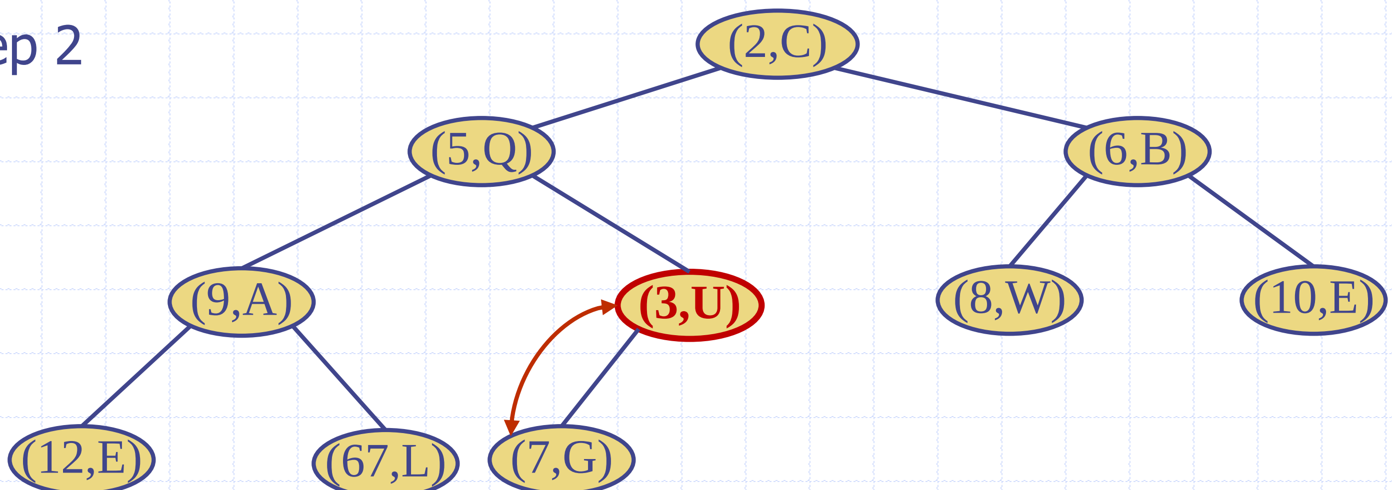
- Present the process step by step

Exercise 4 – Insert – Answer

Step 1

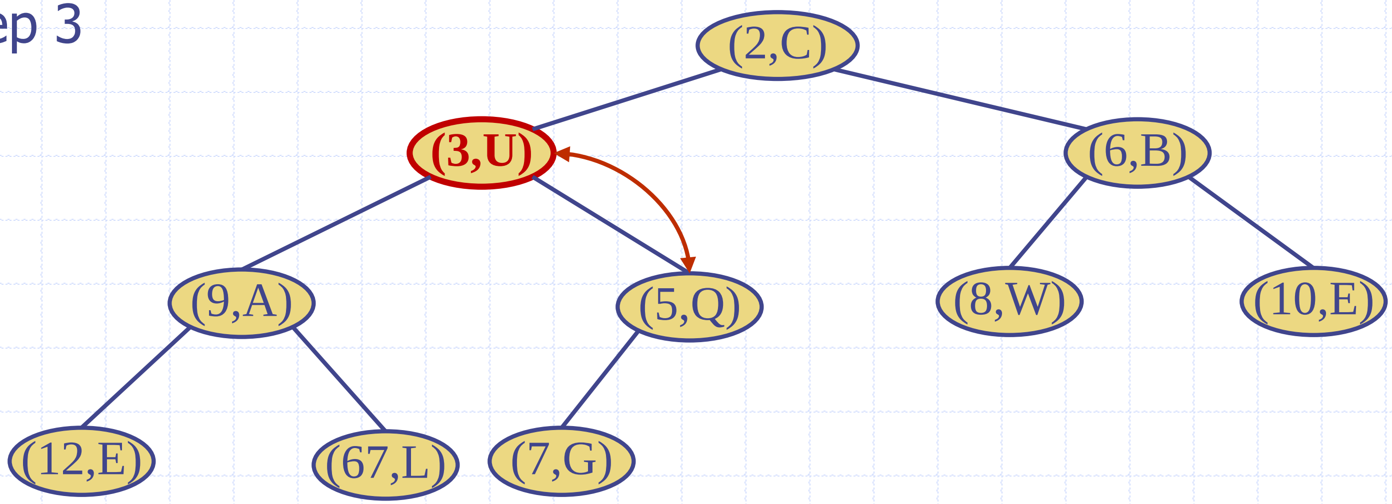


Step 2



Exercise 4 – Insert – Answer

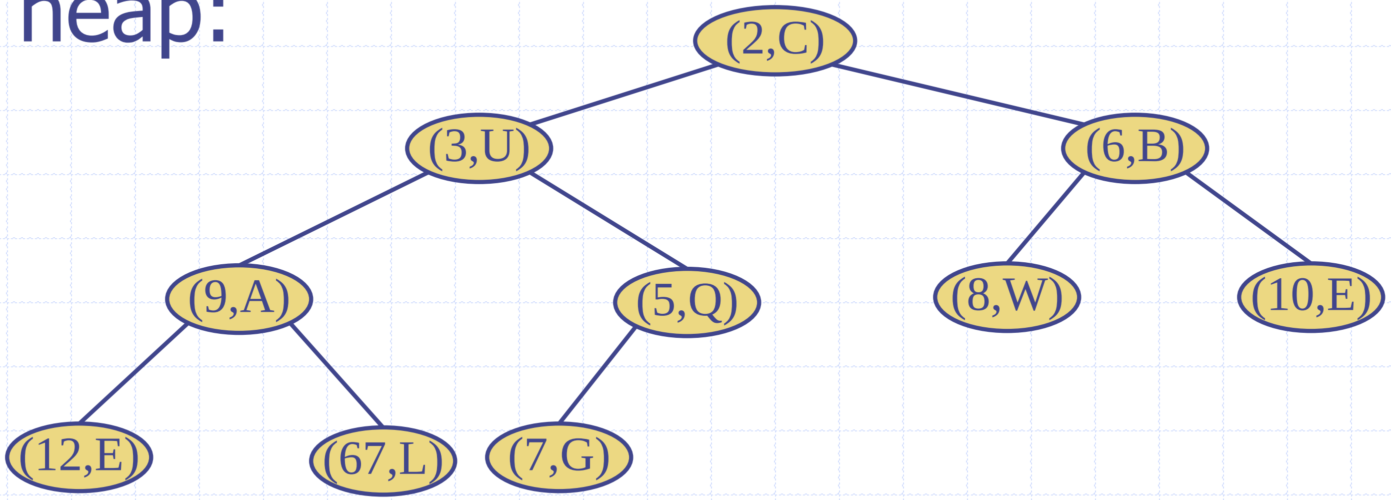
Step 3



END

Exercise 5 – Remove

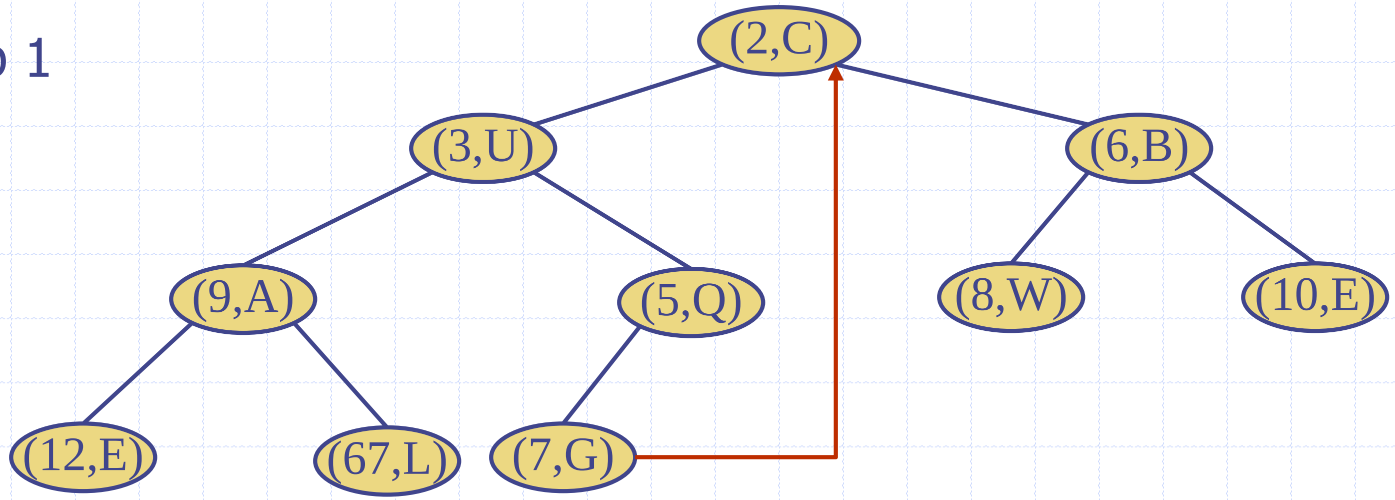
- Perform removal of entry from the priority queue implemented with a heap:



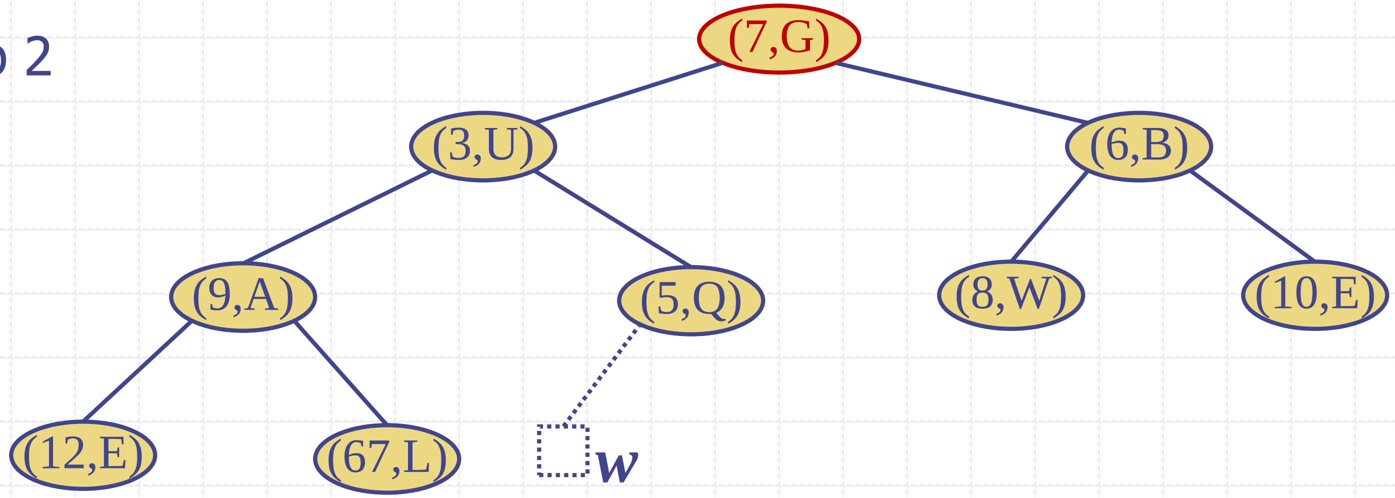
- Present the process step by step

Exercise 5 – Remove – Answer

Step 1

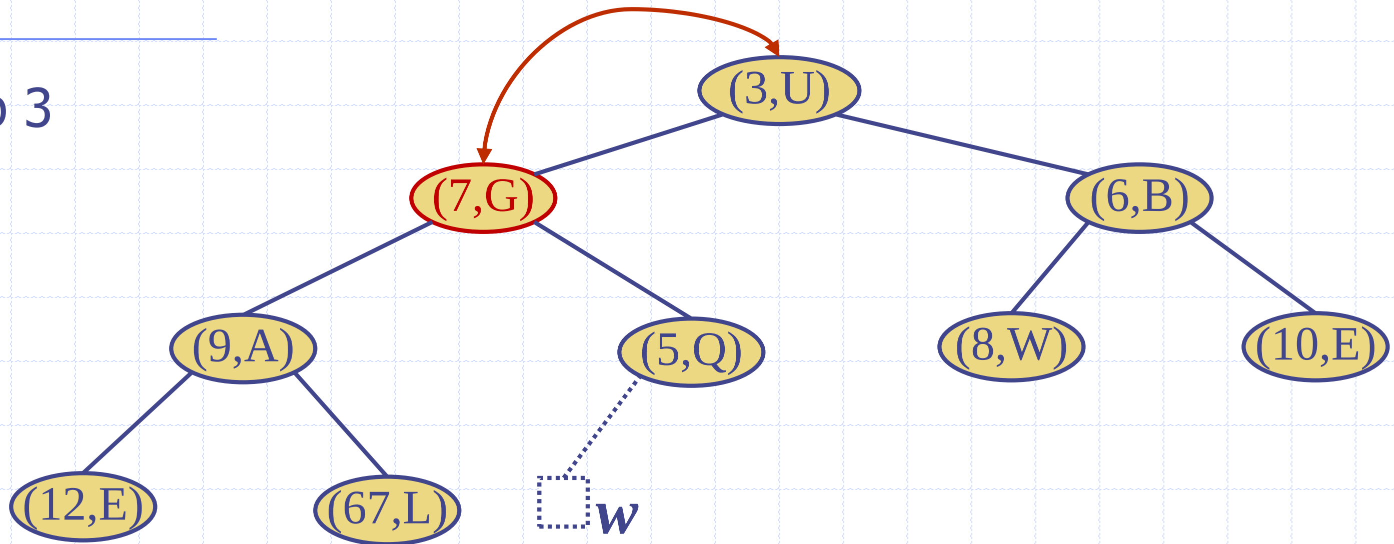


Step 2

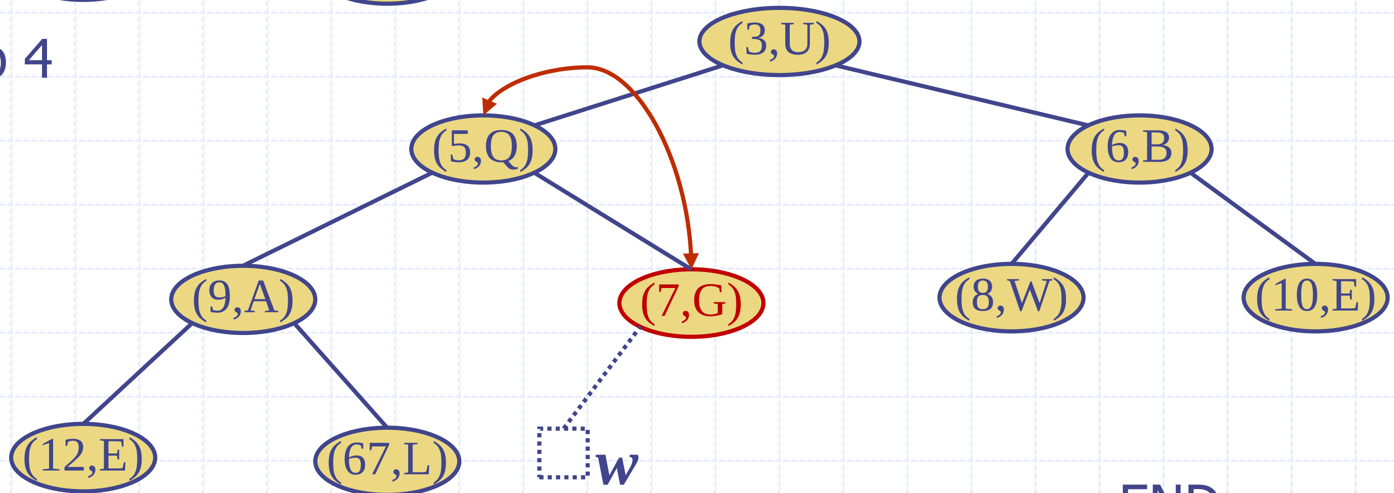


Exercise 5 – Remove – Answer

Step 3

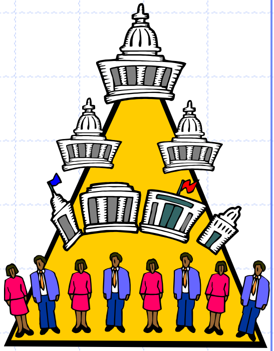


Step 4



END

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, isEmpty, 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
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Exercise 6 – Heap Sort

- Illustrate the execution of the heap-sort algorithm on the following input sequence:

  (14, 45, 23, 98, 12, 99, 1, 78)

Appendix
Java Implementations

Interface for the priority queue ADT

```
/** Interface for the priority queue ADT */
public interface PriorityQueue<K,V> {
  /** Returns the number of items in the priority queue. */
  public int size();
  /** Returns whether the priority queue is empty. */
  public boolean isEmpty();
  /** Returns but does not remove an entry with minimum key. */
  public Entry<K,V> min() throws EmptyPriorityQueueException;
  /** Inserts a key-value pair and return the entry created. */
  public Entry<K,V> insert(K key, V value) throws InvalidKeyException;
  /** Removes and returns an entry with minimum key. */
  public Entry<K,V> removeMin() throws EmptyPriorityQueueException;
}
```

DefaultComparator.java

```
import java.util.Comparator;
import java.io.Serializable;
/** Comparator based on the natural ordering */
public class DefaultComparator<E> implements Comparator<E> {
/** Compares two given elements
* @return a negative integer if a is less than b,
* zero if a equals b, or a positive integer if a is greater than b
*/
public int compare(E a, E b) throws ClassCastException {
  return ((Comparable<E>) a).compareTo(b);
  }
}
```

SortedListPriorityQueue.java

```
/** Realization of a priority queue by means of a sorted node list in  nondecreasing
    order. */
public class SortedListPriorityQueue<K,V> implements PriorityQueue<K,V> {
  protected PositionList<Entry<K,V>> entries;
  protected Comparator<K> c;
  protected Position<Entry<K,V>> actionPos; // variable used by subclasses
  /** Inner class for entries */
  protected static class MyEntry<K,V> implements Entry<K,V> {
    protected K k; // key
    protected V v; // value
    public MyEntry(K key, V value) {
      k = key;
      v = value;
    }
    // methods of the Entry interface
    public K getKey() { return k; }
    public V getValue() { return v; }
```

SortedListPriorityQueue.java

```
/** Creates the priority queue with the default comparator. */
 public SortedListPriorityQueue () {
   entries = new NodePositionList<Entry<K,V>>();
   c = new DefaultComparator<K>();
 }
 /** Creates the priority queue with the given comparator. */
 public SortedListPriorityQueue (Comparator<K> comp) {
   entries = new NodePositionList<Entry<K,V>>();
   c = comp;
 }
/** Returns the number of elements in the priority queue. */
 public int size () {return entries.size();
 }
 /** Returns whether the priority queue is empty. */
 public boolean isEmpty () {return entries.isEmpty();
 }
```

SortedListPriorityQueue.java

```
/** Inserts a key-value pair and return the entry created. */
 public Entry<K,V> insert (K k, V v) throws InvalidKeyException {
   checkKey(k);              // auxiliary key-checking method (could throw exception)
   Entry<K,V> entry = new MyEntry<K,V>(k, v);
   insertEntry(entry);       // auxiliary insertion method
   return entry; }
 /** Auxiliary method used for insertion. */
 protected void insertEntry(Entry<K,V> e) {
   if (entries.isEmpty()) {
     entries.addFirst(e);           // insert into empty list
     actionPos = entries.first(); }  // insertion position
   else if (c.compare(e.getKey(), entries.last().element().getKey()) > 0) {
     entries.addLast(e);            // insert at the end of the list
     actionPos = entries.last(); }    // insertion position
   else {
     Position<Entry<K,V>> curr = entries.first();
     while (c.compare(e.getKey(), curr.element().getKey())> 0) {
        curr = entries.next(curr); }   // advance toward insertion position
     entries.addBefore(curr, e);
     actionPos = entries.prev(curr); }} // insertion position
```

SortedListPriorityQueue.java

```
/** Returns but does not remove an entry with minimum key. */
 public Entry<K,V> min () throws EmptyPriorityQueueException {
  if (entries.isEmpty())
   throw new EmptyPriorityQueueException("priority queue is empty");
  else
   return entries.first().element();
 }
/** Removes and returns an entry with minimum key. */
 public Entry<K,V> removeMin() throws EmptyPriorityQueueException {
  if (entries.isEmpty())
   throw new EmptyPriorityQueueException("priority queue is empty");
  else
   return entries.remove(entries.first());
 }
```