PRIORITY QUEUES & HEAPS

Dr. Megha Ummat

Priority Queue

- A ***priority queue*** is an abstract data type for storing a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time.
- The priority queue ADT stores elements according to their priorities, and has no external notion of “position.”

Key

- ***Key*** is defined as an object that is assigned to an element as a specific attribute for that element and that can be used to identify, rank, or weigh that element.
- Note that the key is assigned to an element, typically by a user or application; hence, a key might represent a property that an element did not originally possess.
- For example, we can compare companies by earnings or by number of employees; hence, either of these parameters can be used as a key for a company, depending on the information we wish to extract.
- Likewise, we can compare restaurants by a critic's food quality rating or by average entree price.

Key

- A key can sometimes be a more complex property that cannot be quantified with a single number. For example, the priority of standby passengers is usually determined by taking into account a host of different factors, including frequent-flyer status, the fare paid, and check-in time.
- In some applications, the key for an object is data extracted from the object itself (for example, the list price of a book, or the weight of a car).
- In other applications, the key is not part of the object but is externally generated by the application (for example, the quality rating given to a stock by a financial analyst).

Comparing Keys with Total Orders

- A priority queue needs a comparison rule that never contradicts itself.
- In order for a comparison rule, which we denote by $\leq$, to be robust in this way, it must define a ***total order*** relation, which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties:
  - **Reflexive property** : $k \leq k$
  - **Antisymmetric property**: if $k1 \leq k2$ and $k2 \leq k1$, then $k1 = k2$
  - **Transitive property**: if $k1 \leq k2$ and $k2 \leq k3$, then $k1 \leq k3$
- Any comparison rule, ≤, that satisfies these three properties never leads to a comparison contradiction.
- If a finite collection of keys has a total order defined for it, then the notion of the ***smallest*** key, k_{min}, is well defined

Priority Queue

- A ***priority queue*** is a container of elements, each associated with a key. The keys determine the "priority" used to pick elements to be removed.
- The fundamental functions of a priority queue *P* are as follows:
  - insert(*e*): Insert the element *e* (with an implicit associated key value) into *P*.
  - min(): Return an element of *P* with the smallest associated key value, that is, an element whose key is less than or equal to that of every other element in *P*.
  - removeMin(): Remove from *P* the element min().

Priority Queue Example

- Suppose a certain flight is fully booked an hour prior to departure. Because of the possibility of cancellations, the airline maintains a priority queue of standby passengers hoping to get a seat.
- The priority of each passenger is determined by the fare paid, the frequent-flyer status, and the time when the passenger is inserted into the priority queue.
- When a passenger requests to fly standby, the associated passenger object is inserted into the priority queue with an insert operation.
- Shortly before the flight departure, if seats become available (for example, due to last-minute cancellations), the airline repeatedly removes a standby passenger with first priority from the priority queue, using a combination of min and removeMin operations, and lets this person board.

Priority Queue ADT

- As an ADT, a priority queue P supports the following functions:
  - size(): Return the number of elements in P.
  - empty(): Return true if P is empty and false otherwise.
  - insert(e): Insert a new element e into P.
  - min(): Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.
  - removeMin(): Remove from P the element referenced by min(); an error condition occurs if the priority queue is empty.
- Note that we allow a priority queue to have multiple entries with the same key.

Priority Queue ADT

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

Heap

- A heap is a binary tree T that stores a collection of elements with their associated keys at its nodes and that satisfies two additional properties:
- a **relational property**, defined in terms of the way keys are stored in T, and
- a **structural property**, defined in terms of the nodes of T itself.

Heap

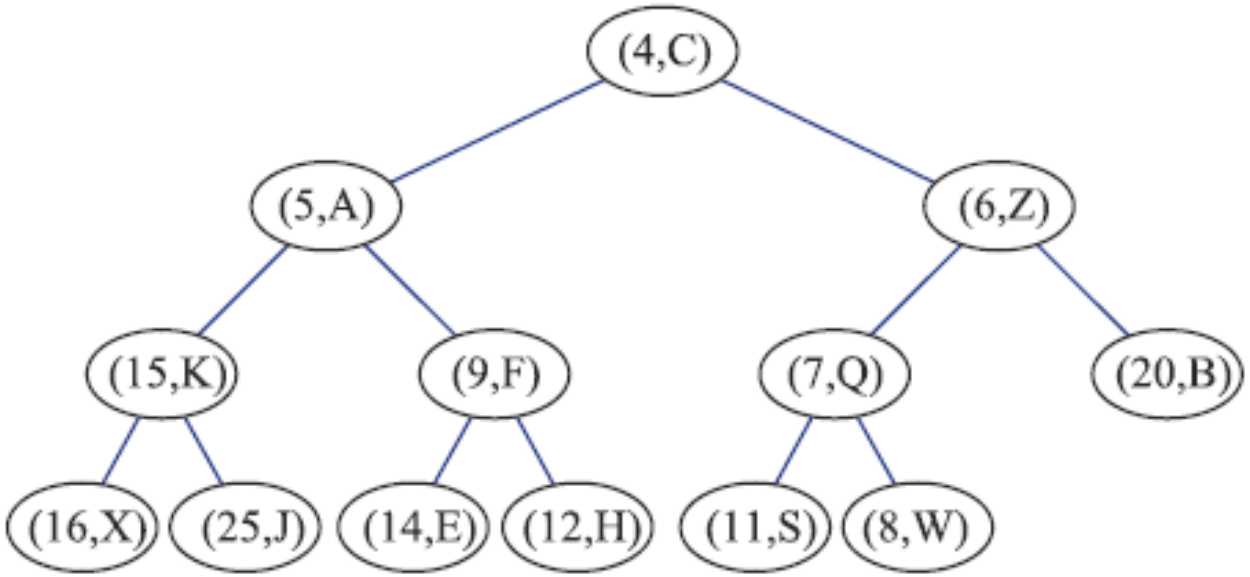


Figure 8.3: Example of a heap storing 13 elements. Each element is a key-value pair of the form (k, v). The heap is ordered based on the key value, k, of each element.

Heap

- **Heap-Order Property:** In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.
- As a consequence of the heap-order property, the keys encountered on a path from the root to an external node of T are in nondecreasing order.
- Also, a minimum key is always stored at the root of T. This is the most important key and is informally said to be "at the top of the heap".

Heap

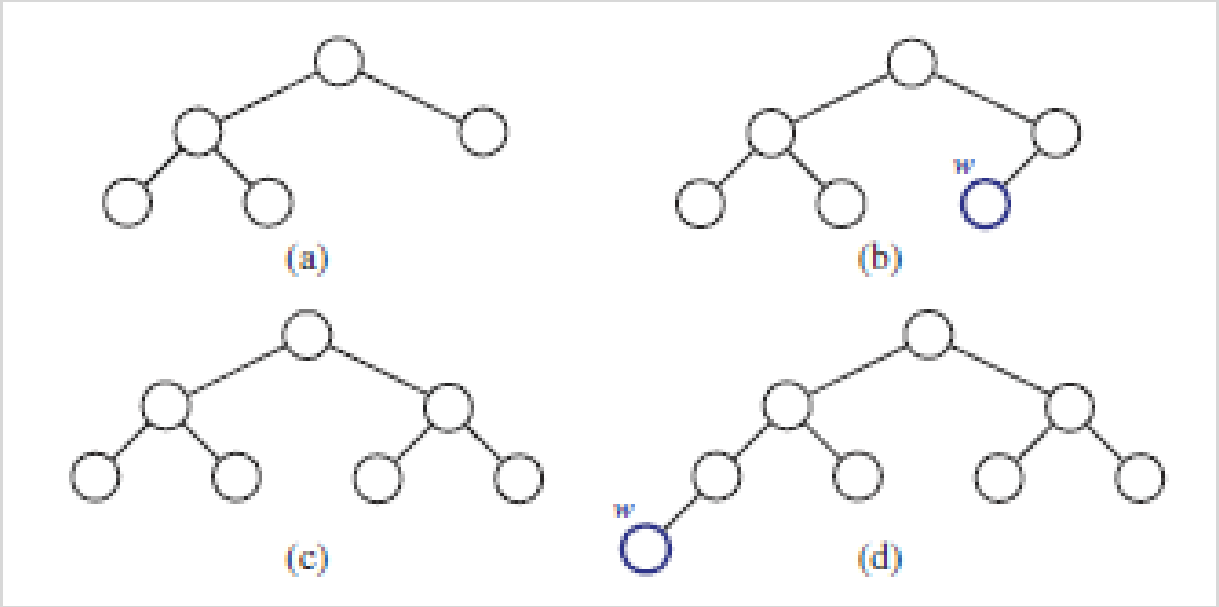
- We enforce that the heap T satisfy an additional structural property, it must be ***complete***.
- **Complete Binary Tree Property:** A heap T with height h is a ***complete*** binary tree, that is, levels 0,1,2, . . . ,$h-1$ of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h-1$) and the nodes at level h fill this level from left to right.
- A heap T storing n entries has height $h = \lfloor \log n \rfloor$.
- It implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time.

Complete Binary Tree ADT

- As an abstract data type, a complete binary tree T supports all the functions of the binary tree ADT, plus the following two functions:
- add(e): Add to T and return a new external node v storing element e, 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.
- By using only these update operations, the resulting tree is guaranteed to be a complete binary.

Complete Binary Tree ADT

- If the bottom level of *T* is not full, then add inserts a new node on the bottom level of *T*, immediately after the rightmost node of this level (that is, the last node); hence, *T*'s height remains the same.
- If the bottom level is full, then add inserts a new node as the left child of the leftmost node of the bottom level of *T*; hence, *T*'s height increases by one.

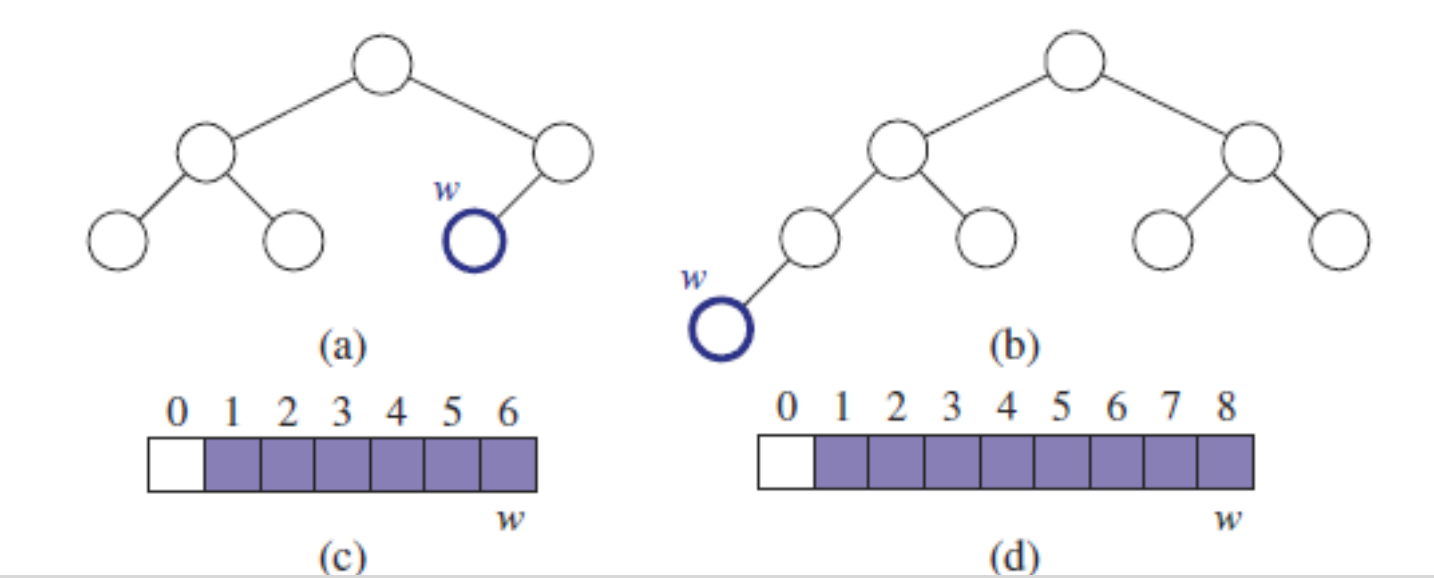


Vector Representation of Complete Binary Tree

- The nodes of T are stored in a vector A such that node v in T is the element of A with index equal to the level number $f(v)$ defined as follows:
  - If v is the root of T, then $f(v) = 1$
  - If v is the left child of node u, then $f(v) = 2\,f(u)$
  - If v is the right child of node u, then $f(v) = 2\,f(u)+1$
- With this implementation, the nodes of T have contiguous indices in the range $[1,n]$ and the last node of T is always at index n, where n is the number of nodes of T.

Vector Representation of Complete Binary Tree

- Assuming that no array expansion is necessary, functions add and remove can be performed in $O(1)$ time because they simply involve adding or removing the last element of the vector.



C++ Implementation of a Complete Binary Tree

- The tree ADT stores elements at the nodes of the tree. Because nodes are internal aspects of our implementation, we do not allow access to them directly. Instead, each node of the tree is associated with a ***position*** object, which provides public access to nodes.

```
template <typename E>
class CompleteTree {                          // left-complete tree interface
public:                                       // publicly accessible types
  class Position;                             // node position type
  int size() const;                           // number of elements
  Position left(const Position& p);           // get left child
  Position right(const Position& p);          // get right child
  Position parent(const Position& p);         // get parent
  bool hasLeft(const Position& p) const;      // does node have left child?
  bool hasRight(const Position& p) const;     // does node have right child?
  bool isRoot(const Position& p) const;       // is this the root?
  Position root();                            // get root position
  Position last();                            // get last node
  void addLast(const E& e);                   // add a new last node
  void removeLast();                          // remove the last node
  void swap(const Position& p, const Position& q); // swap node contents
};
```

C++ Implementation of a Complete Binary Tree

- In order to implement this interface, we store the elements in an STL vector, called *V*. We implement a tree position as an iterator to this vector.
- To convert from the index representation of a node to this positional representation, we provide a function pos. The reverse conversion is provided by function idx.

```
private:                                          // member data
  std::vector<E> V;                               // tree contents
public:                                           // publicly accessible types
  typedef typename std::vector<E>::iterator Position; // a position in the tree
protected:                                        // protected utility functions
  Position pos(int i)                             // map an index to a position
    { return V.begin() + i; }
  int idx(const Position& p) const                // map a position to an index
    { return p - V.begin(); }
```

Implementation of a Complete Binary Tree

- Given the index of a node i, the function pos maps it to a position by adding i to V.begin().
- Here we are exploiting the fact that the STL vector supports a ***random-access iterator***.
- In particular, given an integer i, the expression V.begin()+i yields the position of the i^{th} element of the vector, and,given a position p, the expression $p-V$.begin() yields the index of position p.

Vector based Implementation

```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}                  // constructor
  int size() const                                { return V.size() - 1; }
  Position left(const Position& p)                { return pos(2*idx(p)); }
  Position right(const Position& p)               { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)              { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const           { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const          { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const            { return idx(p) == 1; }
  Position root()                                 { return pos(1); }
  Position last()                                 { return pos(size()); }
  void addLast(const E& e)                        { V.push_back(e); }
  void removeLast()                               { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                                  { E e = *q; *q = *p; *p = e; }
};
```

Vector based Implementation

- As STL vectors are indexed starting at 0, our constructor creates the initial vector with one element. This element at index 0 is never used. As a consequence, the size of the priority queue is one less than the size of the vector.
- Given a node at index i, its left and right children are located at indices $2i$ and $2i+1$, respectively. Its parent is located at index $\lfloor i/2 \rfloor$.
- Given a position p, the functions left, right, and parent first convert p to an index using the utility idx, which is followed by the appropriate arithmetic operation on this index, and finally they convert the index back to a position using the utility pos.

Vector based Implementation

- We determine whether a node has a child by evaluating the index of this child and testing whether the node at that index exists in the vector.
- Operations add and remove are implemented by adding or removing the last entry of the vector, respectively.

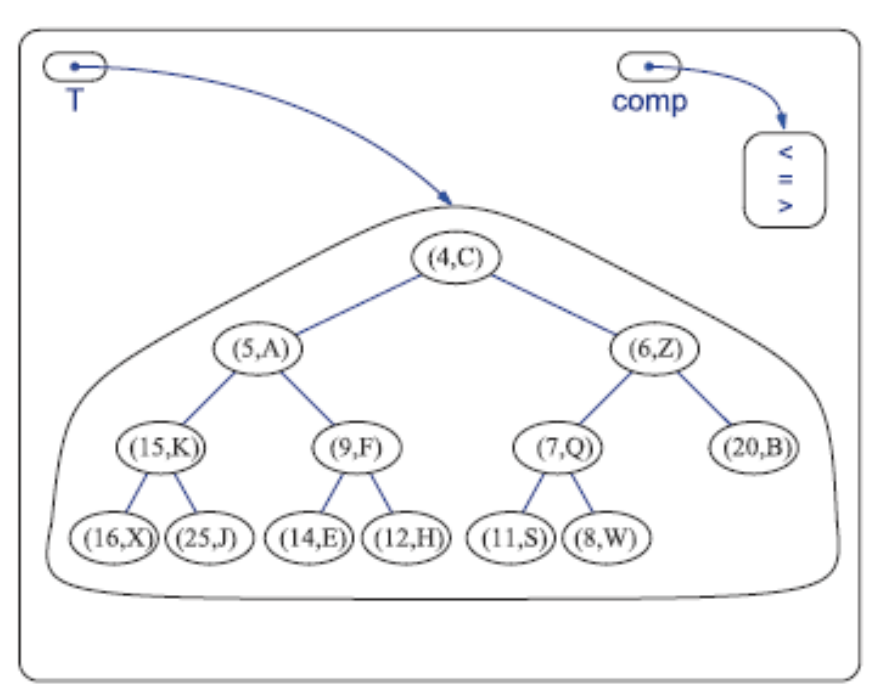
Implementing Priority Queues with Heap

- Our heap-based representation for a priority queue P consists of the following:

- ***heap***: A complete binary tree T whose nodes store the elements of the queue and whose keys satisfy the heap-order property. For each node v of T, we denote the associated key by $k(v)$.

- ***comp***: A comparator that defines the total order relation among the keys.

- With this data structure, functions size and empty take $O(1)$ time, as usual. In addition, function min can also be easily performed in $O(1)$ time by accessing the entry stored at the root of the heap.

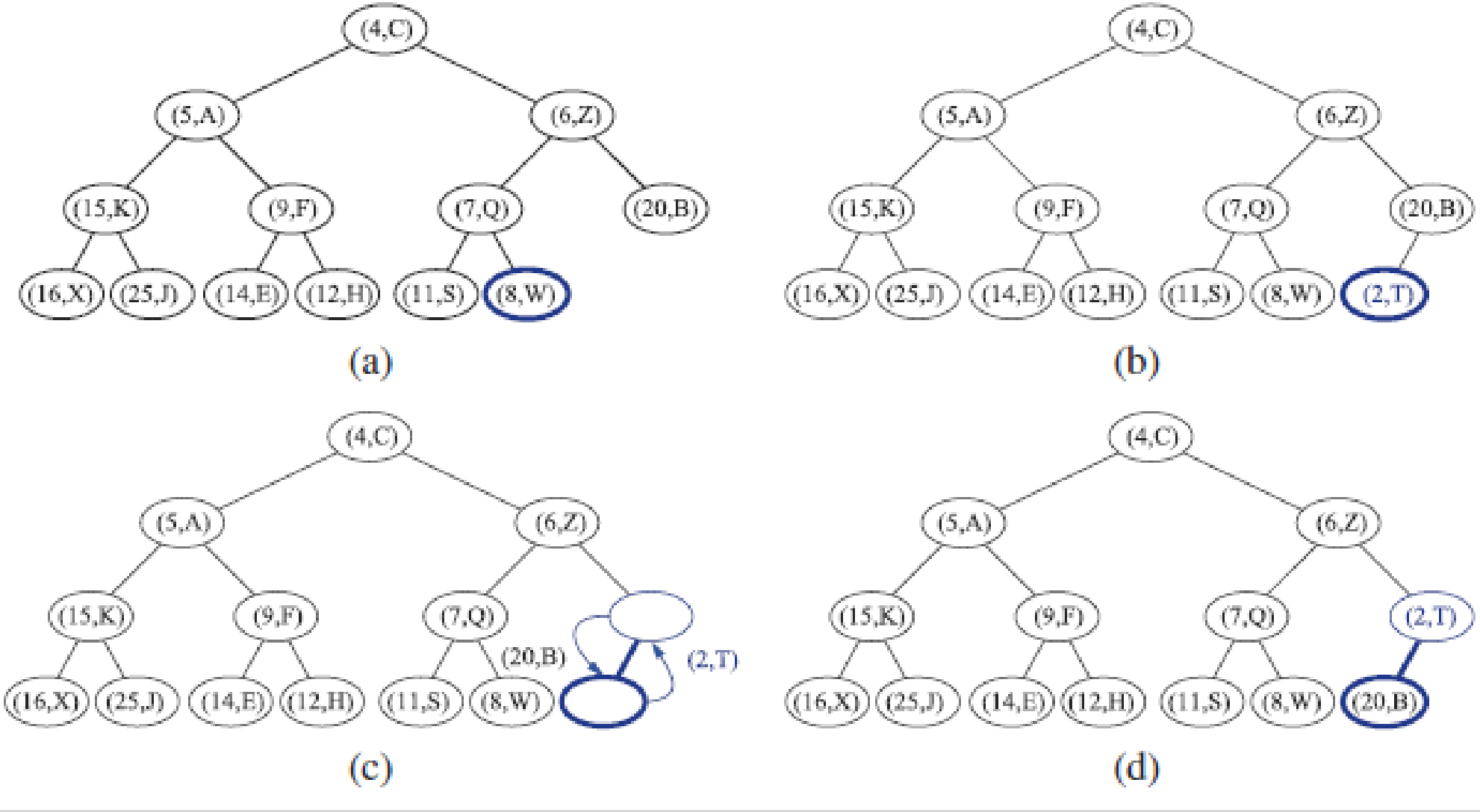


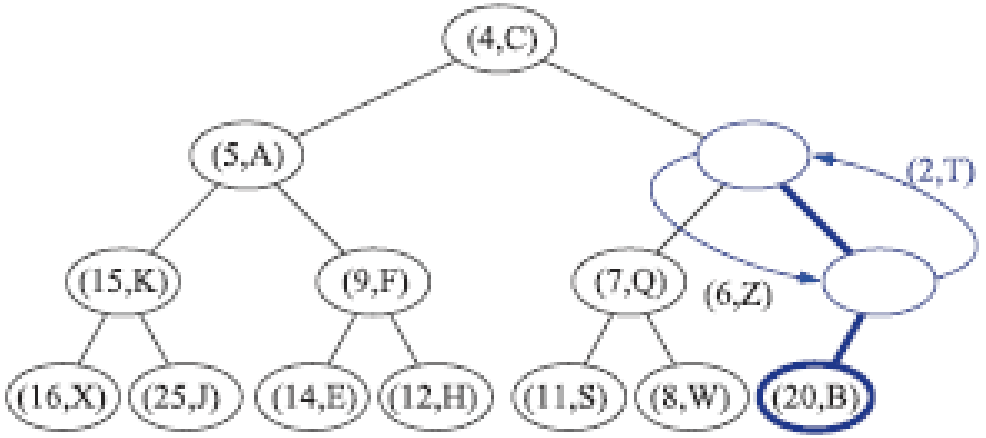
Insertion

- To store a new element e in T, we add a new node z to T with operation add, so that this new node becomes the last node of T, and then store e in this node.
- After this action, the tree T is complete, but it may violate the heap-order property.
- Hence, unless node z is the root of T (that is, the priority queue was empty before the insertion), we compare key $k(z)$ with the key $k(u)$ stored at the parent u of z.
- If $k(z) \geq k(u)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(z) < k(u)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at z and u.

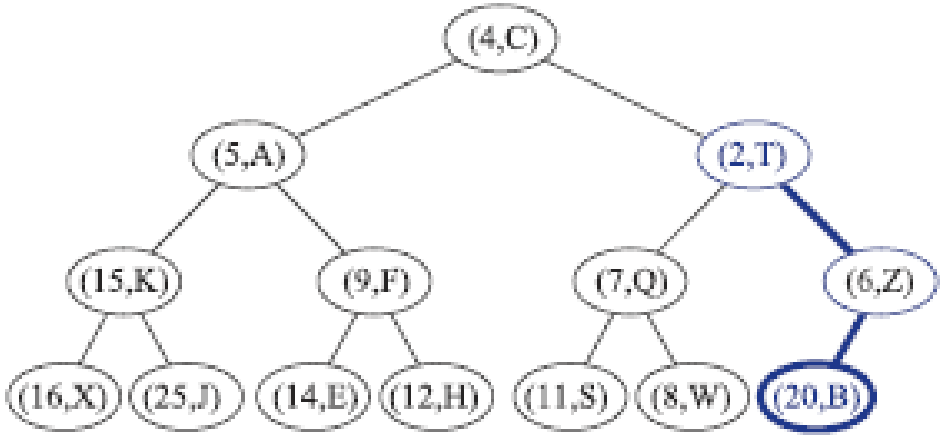
Insertion

- This swap causes the new entry (k,e) to move up one level. Again, the heap-order property may be violated, and we continue swapping, going up in T until no violation of the heap-order property occurs.
- The upward movement of the newly inserted entry by means of swaps is conventionally called ***up-heap bubbling***.
- A swap either resolves the violation of the heap-order property or propagates it one level up in the heap.
- In the worst case, upheap bubbling causes the new entry to move all the way up to the root of heap T. Thus, in the worst case, the number of swaps performed in the execution of function insert is equal to the height of T, that is, it is $\lfloor \log n \rfloor$.

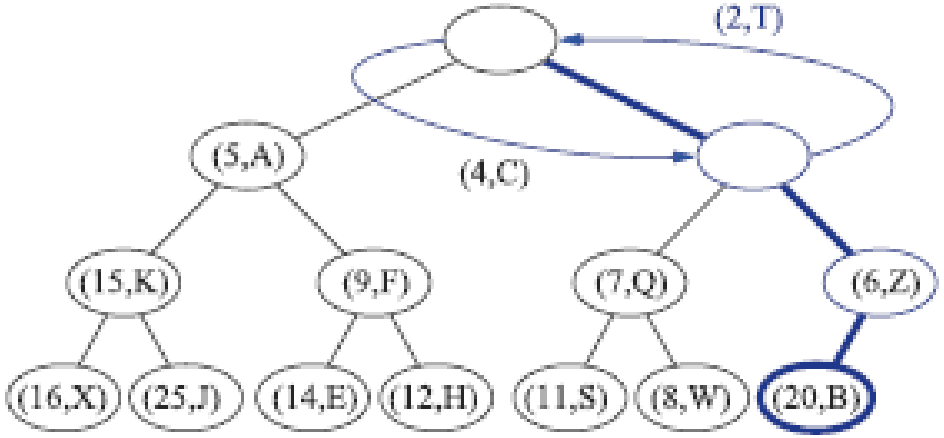

(4,C)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(8,W)
(2,T)
(a)
(b)
(c)
(d)


(4,C)
(5,A)
(2,T)
(15,K)
(9,F)
(7,Q)
(6,Z)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(8,W)
(20,B)

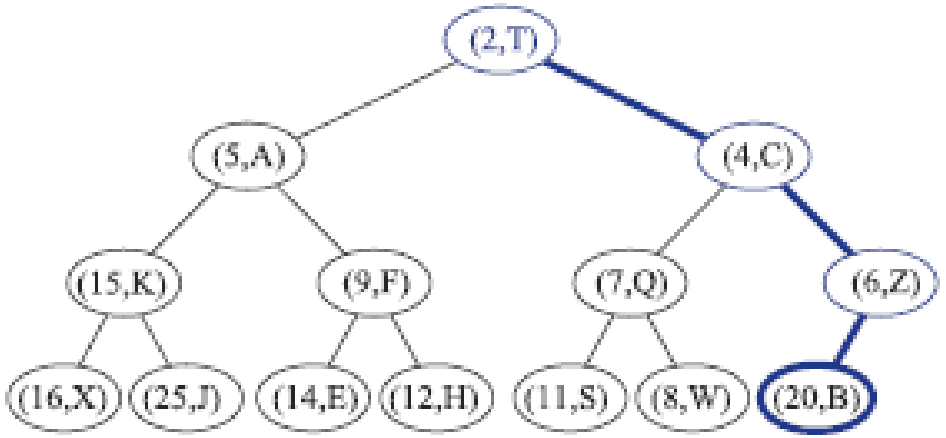
(e)


(4,C)
(5,A)
(2,T)
(15,K)
(9,F)
(7,Q)
(6,Z)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(8,W)
(20,B)

(f)


(2,T)
(5,A)
(4,C)
(15,K)
(9,F)
(7,Q)
(6,Z)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(8,W)
(20,B)

(g)


(2,T)
(5,A)
(4,C)
(15,K)
(9,F)
(7,Q)
(6,Z)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(8,W)
(20,B)

(h)

Removal

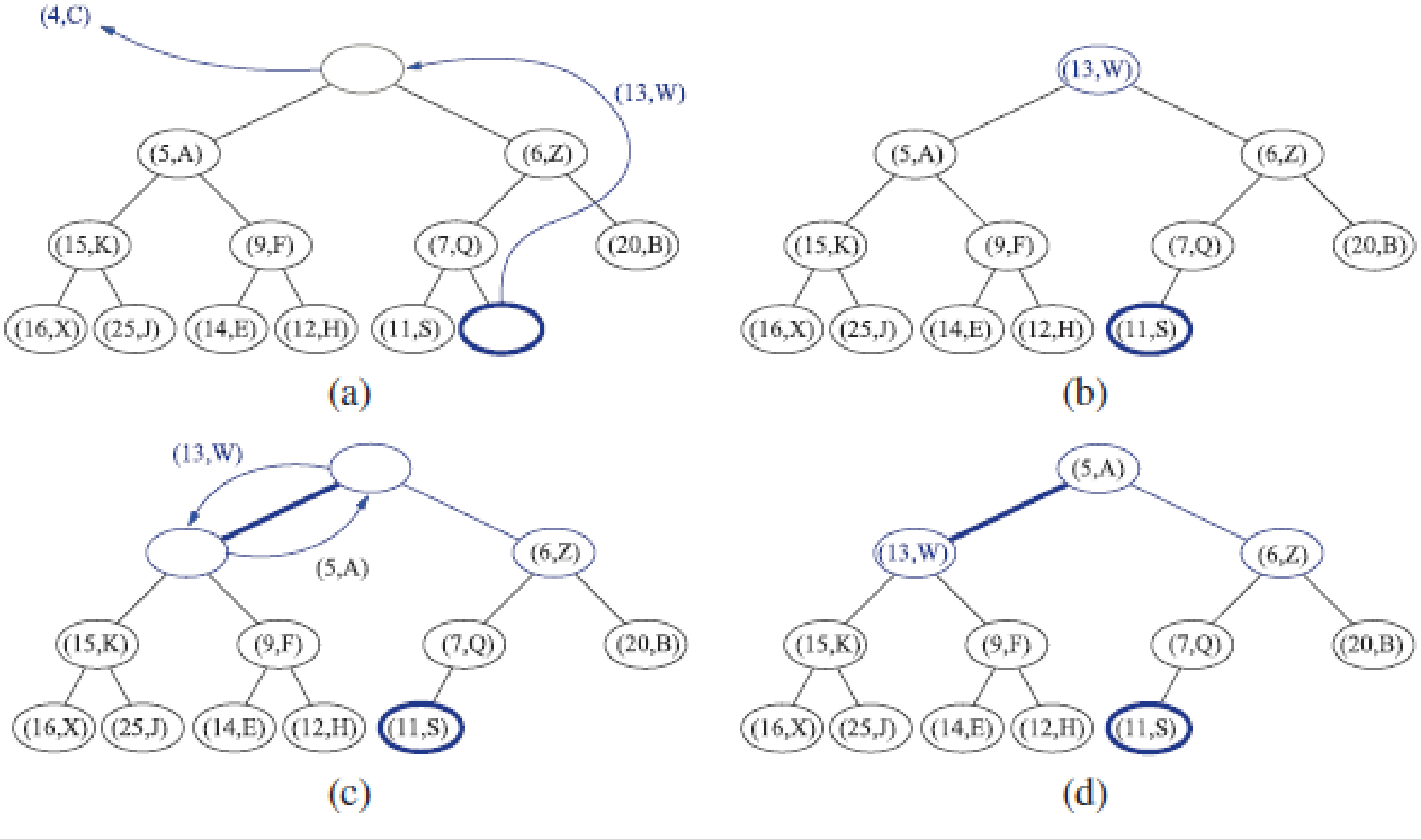
- Performed using function removeMin of the priority queue ADT.
- We know that an element with the smallest key is stored at the root *r* of *T*.
- However, unless *r* is the only node of *T*, we cannot simply delete node *r*, because this action would disrupt the binary tree structure.
- Instead, we access the last node *w* of *T*, copy its entry to the root *r*, and then delete the last node by performing operation remove of the complete binary tree ADT.

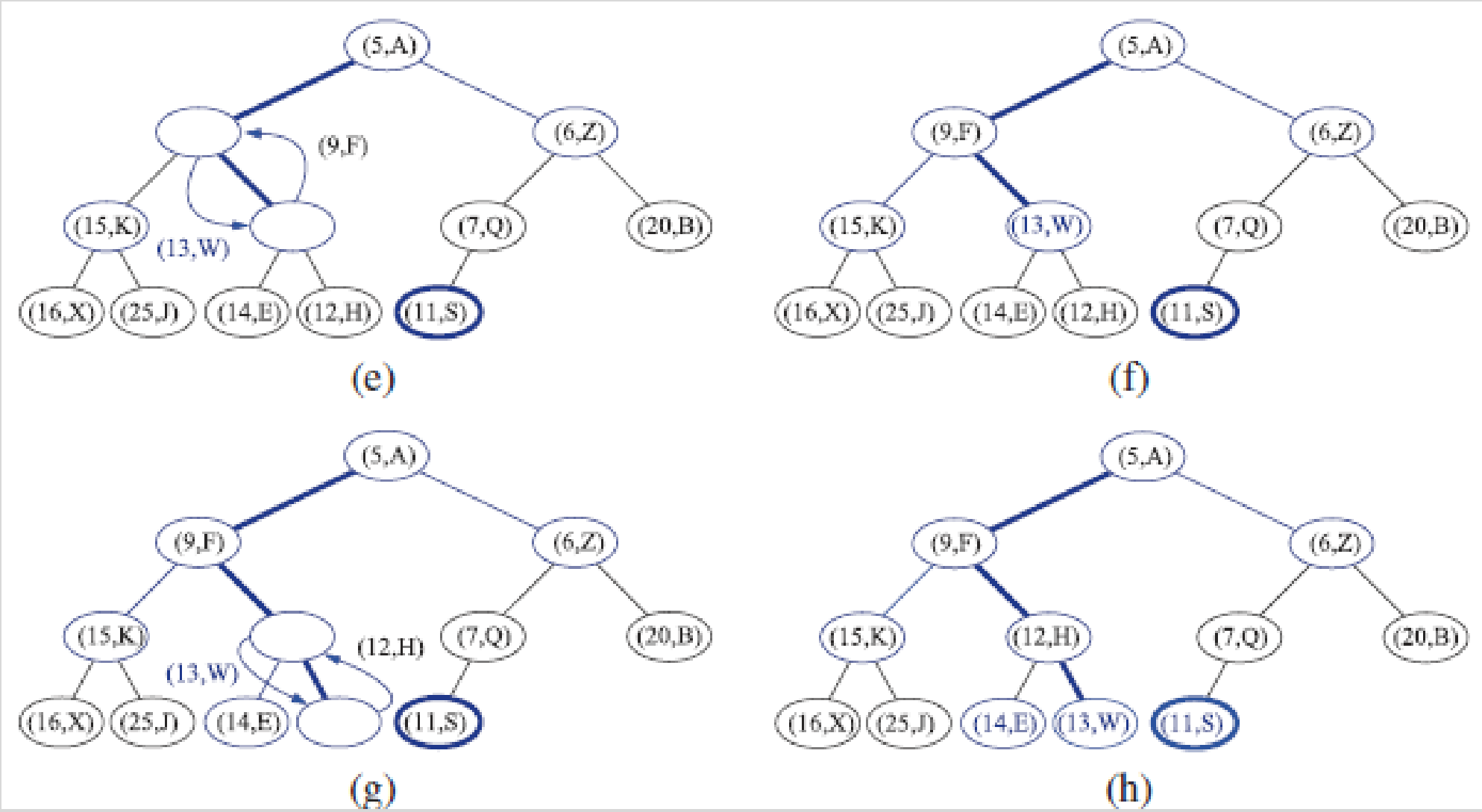
Down-Heap Bubbling after a Removal

- We are not necessarily done, however, for, even though T is now complete, T may now violate the heap-order property.
- If T has only one node (the root), then the heap-order property is trivially satisfied and the algorithm terminates.
- Otherwise, we distinguish two cases, where r denotes the root of T:
  - If r has no right child, let s be the left child of r
  - Otherwise (r has both children), let s be a child of r with the smaller key
- If $k(r) \leq k(s)$, the heap-order property is satisfied and the algorithm terminates.
- If instead $k(r) > k(s)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at r and s.

Down-Heap Bubbling after a Removal

- The swap we perform restores the heap-order property for node r and its children, but it may violate this property at s; hence, we may have to continue swapping down T until no violation of the heap-order property occurs.
- This downward swapping process is called ***down-heap bubbling***.
- In the worst case, an entry moves all the way down to the bottom level. Thus, the number of swaps performed in the execution of function removeMin is, in the worst case, equal to the height of heap T, that is, it is $\lfloor \log n \rfloor$.


(4,C)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(a)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(b)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(c)
(5,A)
(13,W)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(d)


(5,A)
(9,F)
(6,Z)
(15,K)
(13,W)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(e)
(5,A)
(9,F)
(6,Z)
(15,K)
(13,W)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(f)
(5,A)
(9,F)
(6,Z)
(15,K)
(12,H)
(13,W)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(11,S)
(g)
(5,A)
(9,F)
(6,Z)
(15,K)
(12,H)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(13,W)
(11,S)
(h)

Analysis

- Performance of a priority queue realized by means of a heap, which is in turn implemented with a vector or linked structure.
- We denote with n the number of entries in the priority queue at the time a method is executed. The space requirement is $O(n)$.
- Operations `add` and `remove` on T take either $O(1)$ amortized time (vector representation) or $O(\log n)$ worst-case time.

Operation	*Time*
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

We conclude that the heap data structure is a very efficient realization of the priority queue ADT.

C++ Implementation

- We present a heap-based priority queue implementation. The heap is implemented using the vector-based complete tree implementation discussed earlier.

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                          // number of elements
  bool empty() const;                        // is the queue empty?
  void insert(const E& e);                   // insert element
  const E& min();                            // minimum element
  void removeMin();                          // remove minimum
private:
  VectorCompleteTree<E> T;                   // priority queue contents
  C isLess;                                  // less-than comparator
                                             // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

C++ Implementation

- The class's data members consists of the complete tree, named *T*, and an instance of the comparator object, named *isLess*.
- We have also provided a type definition for a node position in the tree, called Position.

C++ Implementation

```
template <typename E, typename C>      // number of elements
int HeapPriorityQueue<E,C>::size() const
  { return T.size(); }

template <typename E, typename C>      // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
  { return size() == 0; }

template <typename E, typename C>      // minimum element
const E& HeapPriorityQueue<E,C>::min()
  { return *(T.root()); }                 // return reference to root element
```

The function min returns a reference to the root's element through the use of the "*" operator, which is provided by the Position class of VectorCompleteTree.

Insert Function

```
template <typename E, typename C>      // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                       // add e to heap
  Position v = T.last();              // e's position
  while (!T.isRoot(v)) {              // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;       // if v in order, we're done
    T.swap(v, u);                     // ...else swap with parent
    v = u;
  }
}
```

This works by adding the new element to the last position of the tree and then it performs up-heap bubbling by repeatedly swapping this element with its parent until its parent has a smaller key value.

removeMin()

```
template <typename E, typename C>      // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)                    // only one node?
    T.removeLast();                   // ...remove it
  else {
    Position u = T.root();            // root position
    T.swap(u, T.last());              // swap last with root
    T.removeLast();                   // ...and remove last
    while (T.hasLeft(u)) {            // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
        v = T.right(u);               // v is u's smaller child
      if (isLess(*v, *u)) {           // is u out of order?
        T.swap(u, v);                 // ...then swap
        u = v;
      }
      else break;                     // else we're done
    }
  }
}
```

- If the tree has only one node, then we simply remove it.
- Otherwise, we swap the root's element with the last element of the tree and remove the last element.
- We then apply down-heap bubbling to the root. Letting *u* denote the current node, this involves determining *u*'s smaller child, which is stored in *v*. If the child's key is smaller than *u*'s, we swap *u*'s contents with this child's.