CSE214 Data Structures
Heaps

YoungMin Kwon

Priority Queues

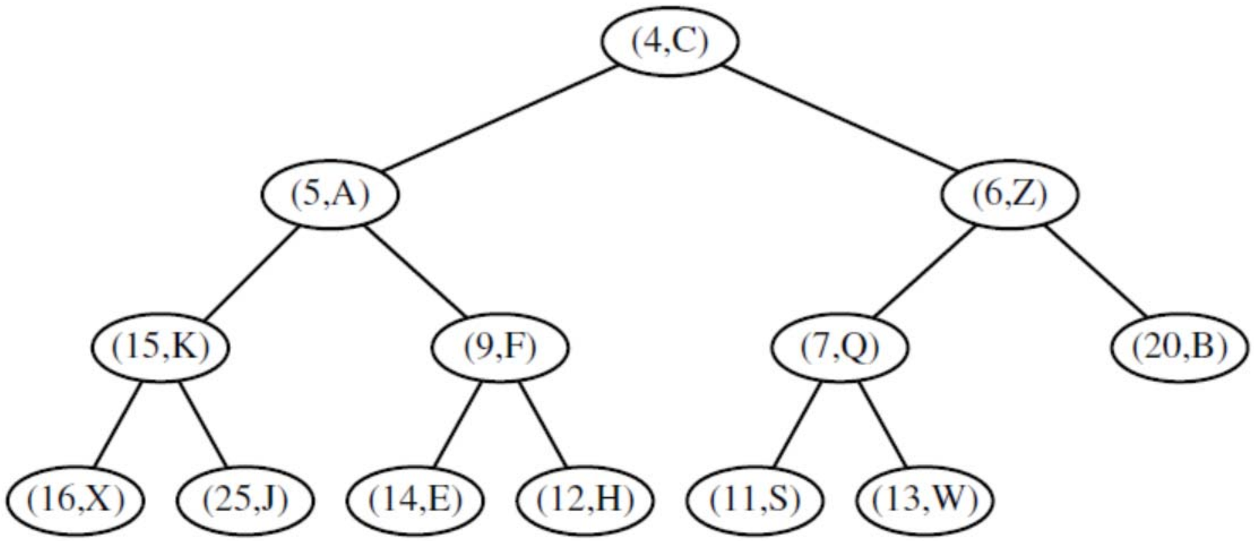
- Priority queue by unsorted list
  - O(1) to insert
  - O(n) to find or remove
- Priority queue by sorted list
  - O(n) to insert
  - O(1) to find or remove

Priority Queues

- Priority queue by **binary heap**
  - O(log n) to insert
  - O(log n) to remove
  - O(1) to find
- Binary heap
  - Use the structure of a binary tree
  - Data is neither entirely unsorted nor perfectly sorted

Heap Data Structure

- A heap is a binary tree T that satisfies
  - A ***relational*** property and a ***structural*** property



SUNY Korea

Heap Data Structure

- *Heap-order* property (*relational* property)
  - For every non-root position p in T, *the key of p is greater than or equal to its parent's key*
  - The keys encountered in a path from the root is non-decreasing order
  - The minimal key is always stored at the root

Heap Data Structure

- *Complete binary tree* property (*structural* 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)
  - The remaining nodes at level *h* resides in the leftmost possible positions at that level

Height of a Heap

- Proposition
  - A heap T storing n entries has height $h = \lfloor \log n \rfloor$
- Proof
  - From the completeness,
    - The number of nodes in level 0 through h-1 is $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$
    - The number of nodes in level h is at least 1 and at most 2^h
  - Hence, $2^h - 1 + 1 \leq n \leq 2^h - 1 + 2^h$

    $$2^h \leq n \leq 2^{h+1} - 1$$
  - Take log on both sides: $h \leq \log n$ and $h \geq \log(n+1) - 1$
  - Because h is an integer, $h = \lfloor \log n \rfloor$

Adding to the Heap

- Complete binary tree property
  - New node should be placed at just beyond the rightmost node at the bottom level
  - Or the leftmost position of a new level

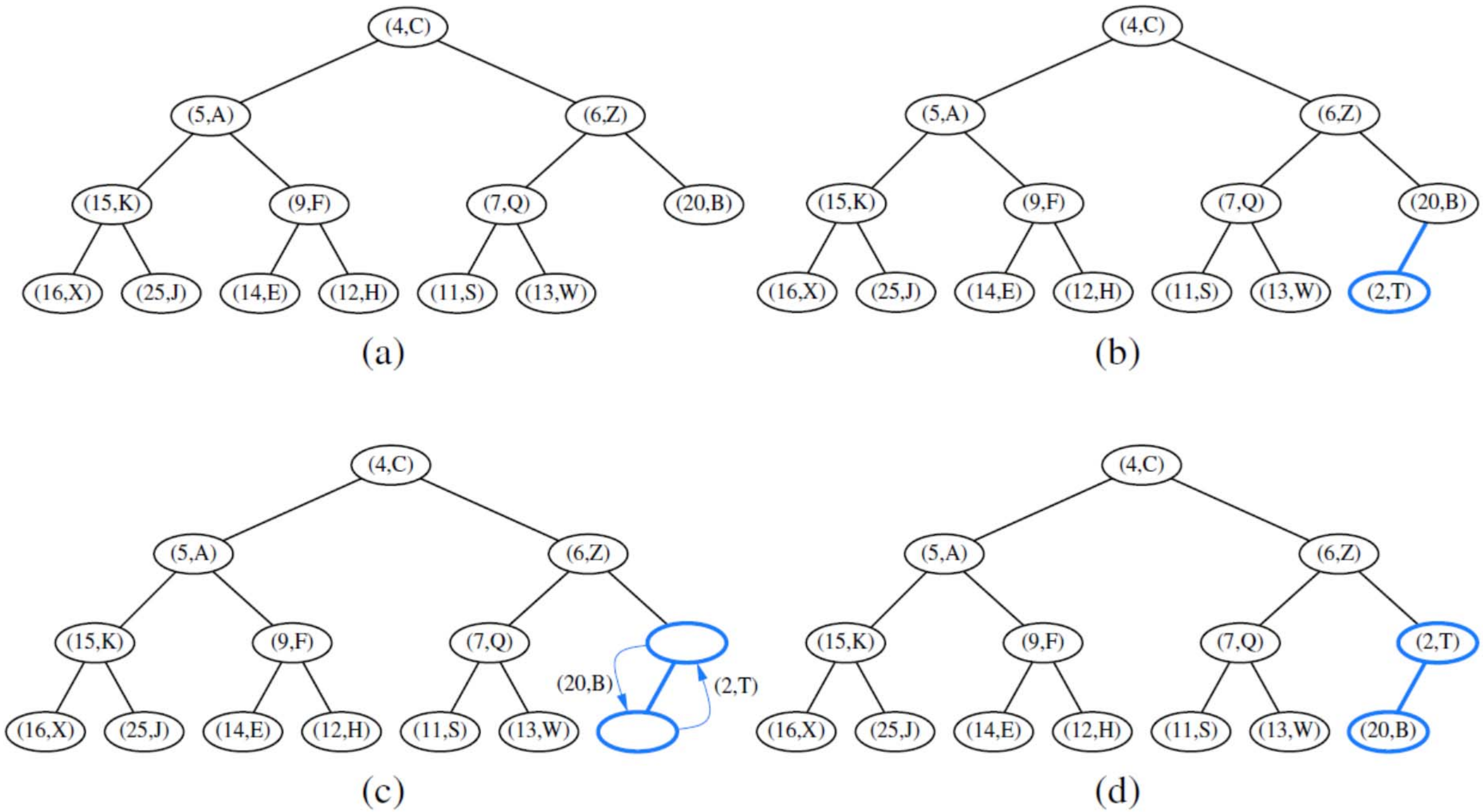
Adding to the Heap

- Adding the new node may break the heap-order property

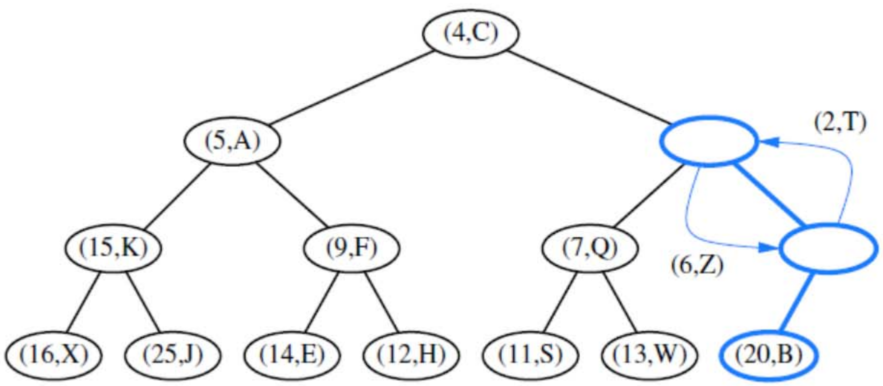
- Up-heap bubbling
  - Compare the key of position p and its parent's key
  - If the parent has a larger key, swap the entries
  - Repeat the above until the swapping stops

- The bound of up-heap bubbling is $\lfloor \log n \rfloor$

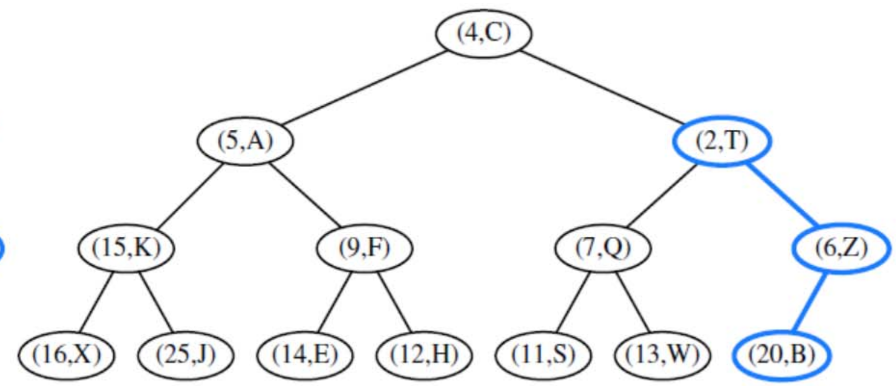
Adding to the Heap



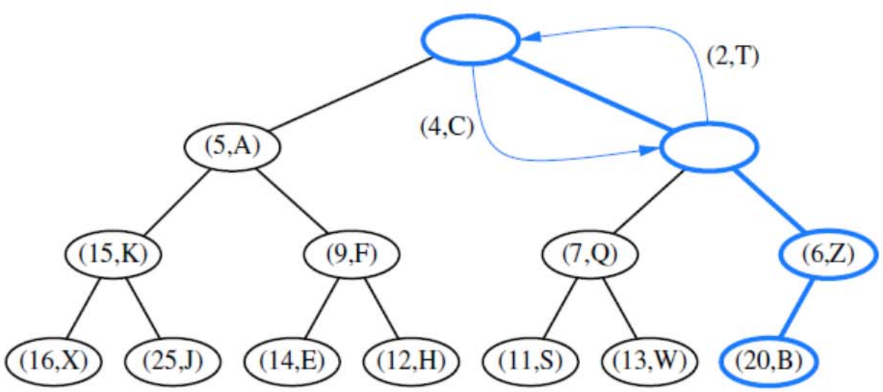
Adding to the Heap



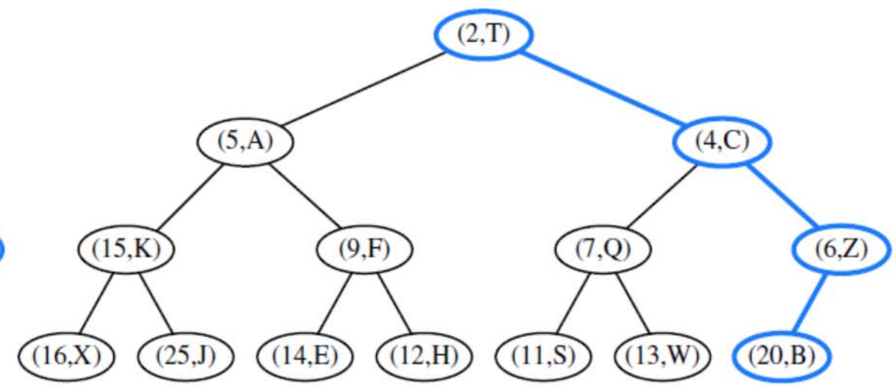
(e)



(f)



(g)



(h)

Removing from the Heap

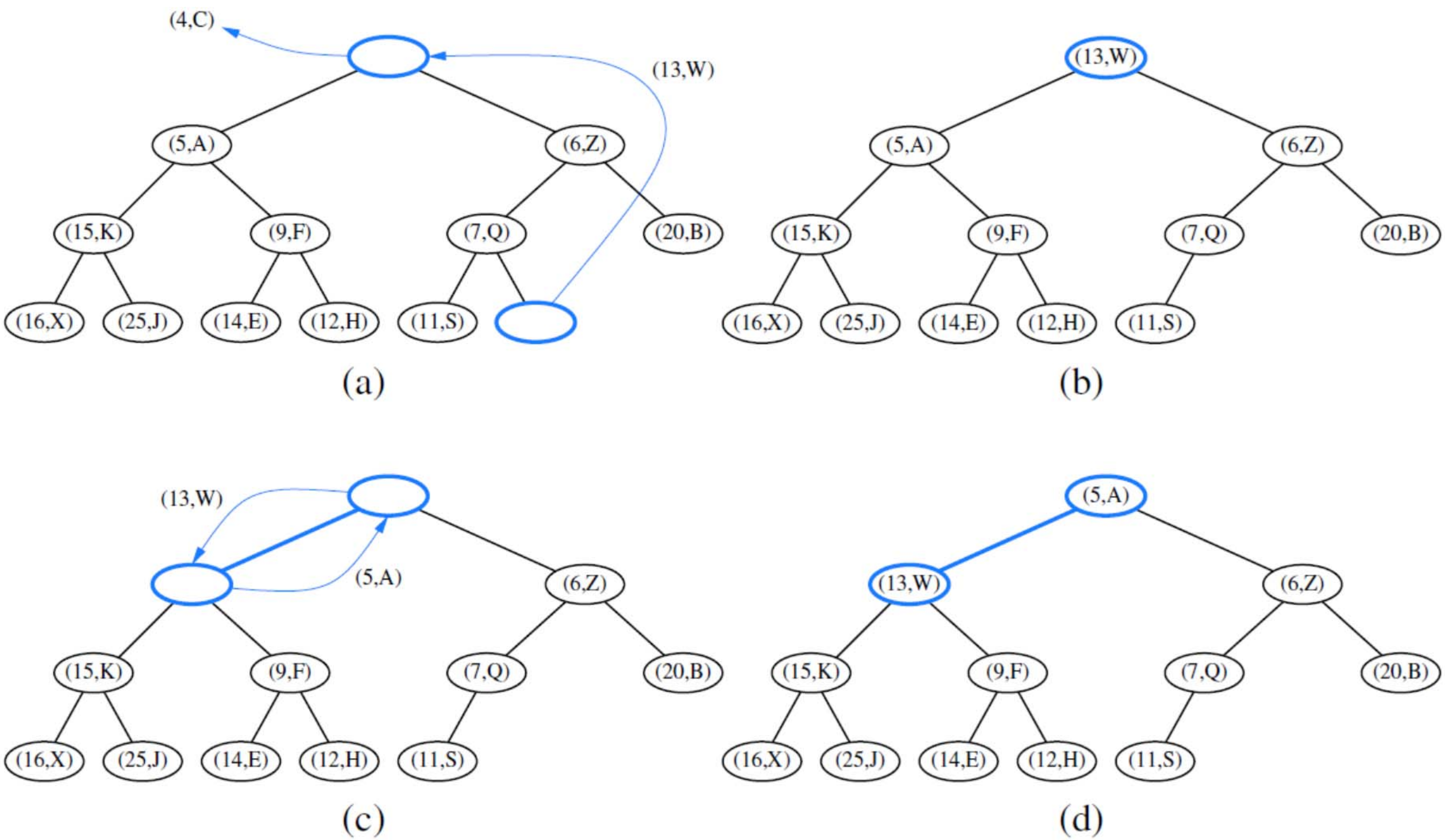
- Removing the entry with the minimal key
  - The entry with the smallest key is at the root
  - Deleting the root would leave two disconnected trees
- To maintain complete binary tree property
  - After deleting the root, move the rightmost node at the bottom level to the root

Removing from the Heap

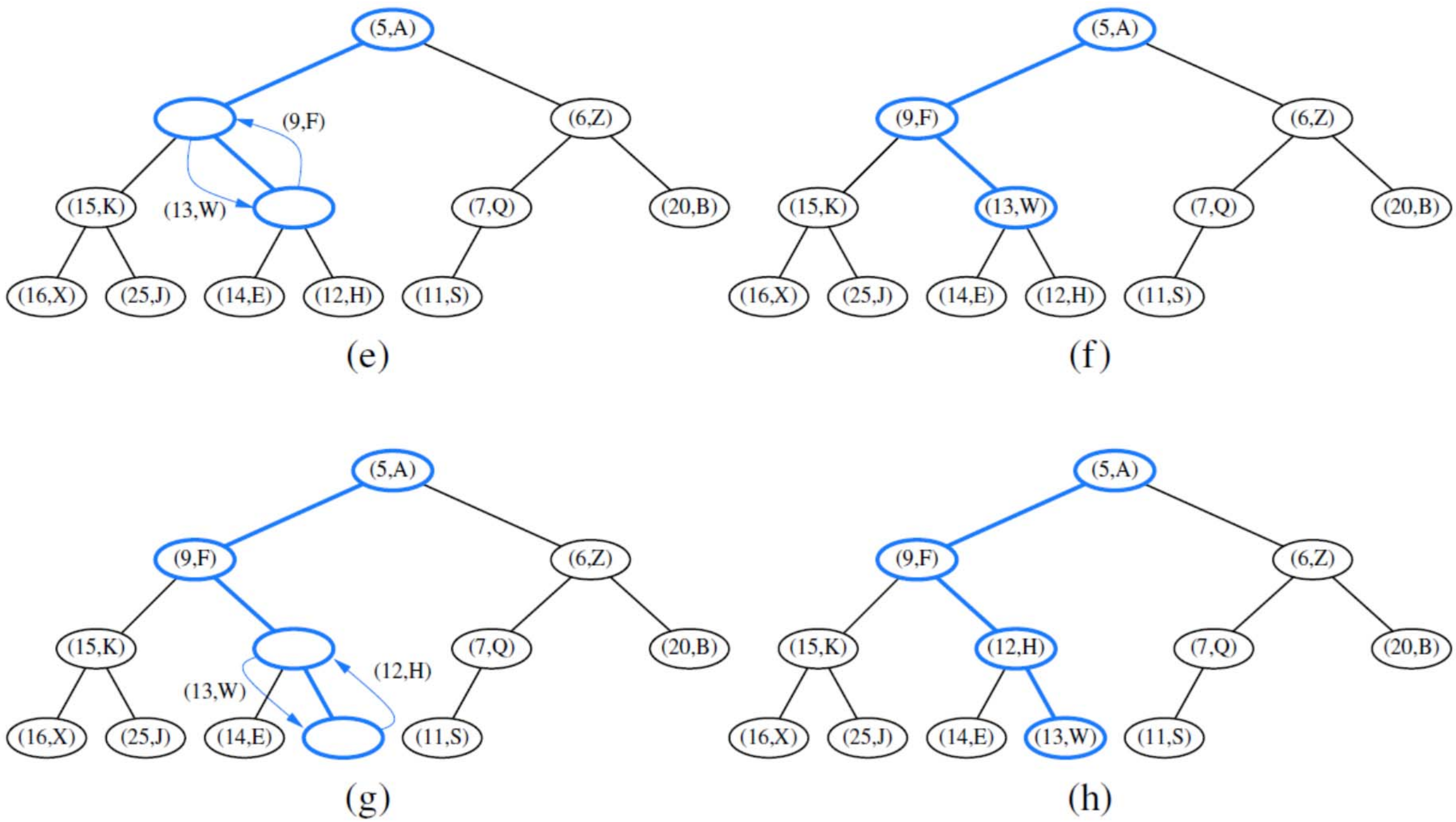
- Moving the node may break the heap-order property
- Down-heap bubbling
  - From a position *p*, find the child *c* that has the *smaller key*
  - Compare the key of *p* and the key of *c*
  - If the *p* has a larger key, swap the entries
  - Repeat the above until the swapping stops or *p* reaches the bottom level
- The bound of down-heap bubbling is $\lfloor \log n \rfloor$

SUNY Korea

Removing from the Heap



Removing from the Heap

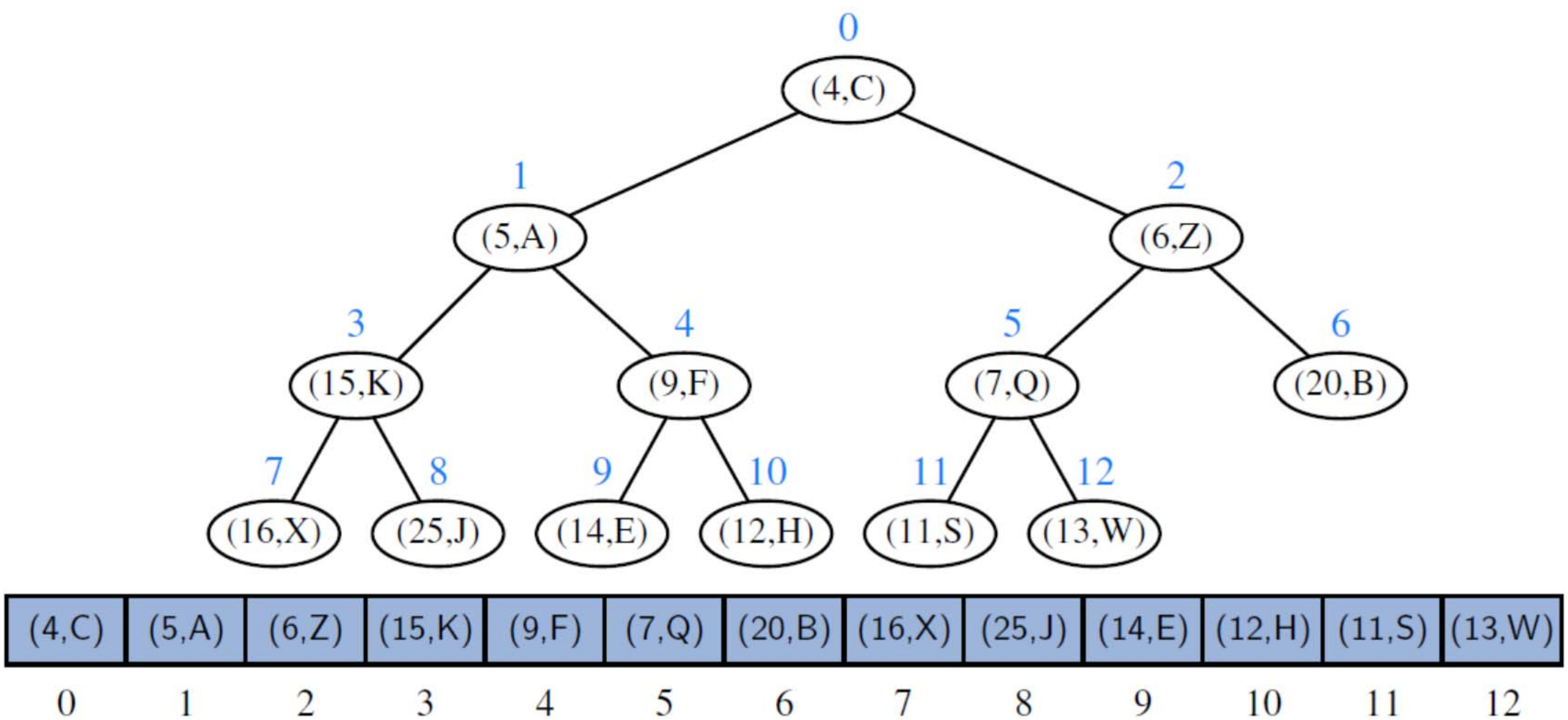


Complete Binary Tree by Array

- Array based representation of binary trees
  - Especially useful for a complete binary tree

- Let *index* (*p*) be the index of position *p*
  - If *p* is the root, then
    $index(p) = 0$
  - If *p* is the *left child* of *q*, then
    $index(p) = 2 \cdot index(q) + 1$
  - If *p* is the *right child* of *q*, then
    $index(p) = 2 \cdot index(q) + 2$

Complete Binary Tree by Array



Complete Binary Tree by Array

- Benefits of using arrays for heaps
  - *insert* and *removeMin* need to find the last position of the heap
  - In array, its position is n-1 if the heap size is n
- Space usage is O(n)
- Time complexity of *insert*
  - O($log(n)$) for static array, O(n) for dynamic array
- Time complexity of *removeMin*
  - O($log(n)$)

Java Implementation of Heap

```
public class HeapPriorityQueue<K extends Comparable<K>, V>
             extends AbstractPriorityQueue<K, V> {
    protected ArrayList<Entry<K, V>> heap;

    //constructors
    public HeapPriorityQueue() {
        super();
        heap = new DynamicArrayList<Entry<K, V>>();
    }
    public HeapPriorityQueue(Comparator<K> comp) {
        super(comp);
        heap = new DynamicArrayList<Entry<K, V>>();
    }

    //protected utilities
    protected int parent(int j)        { return (j-1)/2; }
    protected int left(int j)          { return 2*j + 1; }
    protected int right(int j)         { return 2*j + 2; }
    protected boolean hasLeft(int j)  { return left(j) < heap.size(); }
    protected boolean hasRight(int j) { return right(j) < heap.size(); }
```

```java
//exchange entries
protected void swap(int i, int j) {
    Entry<K, V> tmp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j,  tmp);
}

//up-heap
protected void upheap(int j) {
    while(j > 0) {
        int p = parent(j);
        if(compare(heap.get(j), heap.get(p)) >= 0)
            break;
        swap(j, p);
        j = p;
    }
}
```

```java
//down-heap
protected void downheap(int j) {
    while(hasLeft(j)) {
        int l = left(j);
        int c = l;  //child to compare (smaller of l and r)
        if(hasRight(j)) {
            int r = right(j);
            if(compare(heap.get(l), heap.get(r)) > 0)
                c = r;
        }

        if(compare(heap.get(c), heap.get(j)) >= 0)
            break;
        swap(j, c);
        j = c;
    }
}
```

```java
//public methods
public int size()          { return heap.size(); }
public Entry<K, V> min() {
    if(heap.isEmpty())
        return null;
    return heap.get(0);
}
public Entry<K, V> insert(K key, V value)
                   throws IllegalArgumentException {
    checkKey(key);
    Entry<K, V> newest = new PQEntry<K, V>(key, value);
    heap.add(heap.size(), newest);
    upheap(heap.size() - 1);
    return newest;
}
public Entry<K, V> removeMin() {
    if(heap.isEmpty())
        return null;
    Entry<K, V> ret = heap.get(0);
    swap(0, heap.size() - 1);
    heap.remove(heap.size() - 1);
    downheap(0);
    return ret;
}
```

```
    //unit test methods
    protected static void onFalseThrow(boolean b) {
        if(!b)
            throw new RuntimeException("Error: unexpected");
    }
    public static void main(String[] args) {
        HeapPriorityQueue<Integer, String> pq = new HeapPriorityQueue<>();
        pq.insert(9, "9");  pq.insert(6, "6");
        pq.insert(5, "5");  pq.insert(4, "4");
        pq.insert(2, "2");  pq.insert(8, "8");
        pq.insert(7, "7");  pq.insert(1, "1");
        pq.insert(3, "3");  pq.insert(0, "0");

        onFalseThrow(pq.size() == 10);
        for(int i = 0; i < 10; i++)
            onFalseThrow(pq.removeMin().getValue().equals("" + i));
        onFalseThrow(pq.isEmpty());
        System.out.println("Success!");
    }
}
```

Sorting with a Priority Queue

- Sorting algorithm
  - Phase 1: add elements of a list to a priority queue as keys using *insert*
  - Phase 2: extract elements from the priority queue and put the keys back to the list using *removeMin*

Sorting with a Priority Queue

```
/** Sorts sequence S, using initially empty priority queue P to produce the order. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
  int n = S.size();
  for (int j=0; j < n; j++) {
    E element = S.remove(S.first());
    P.insert(element, null);          // element is key; null value
  }
  for (int j=0; j < n; j++) {
    E element = P.removeMin().getKey();
    S.addLast(element);               // the smallest key in P is next placed in S
  }
}
```

SUNY Korea

Selection Sort

- If priority queue P is implemented as an *unsorted list*

		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)	()

Selection Sort

- Running time
  - Phase 1: insert takes O(1) time for each element
  - Phase 2: selecting min element in *removeMin* takes a time proportional to the number of elements

$$O\left(n + (n-1) + \cdots + 2 + 1\right) = O\left(\sum_{i=1}^{n} i\right)$$

  - Selection sort is O(n^2)

Insertion Sort

- If priority queue P is implemented as a *sorted list*

		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)	( )

Insertion Sort

- Running time
  - Phase 1: inserting an element to its position takes a time proportional to the number of elements

$$O\left(n+(n-1)+\cdots+2+1\right)=O\left(\sum_{i=1}^{n} i\right)$$

  - Phase 2: *removeMin* takes O(1) time for each element
  - Insertion sort is O(n^2)

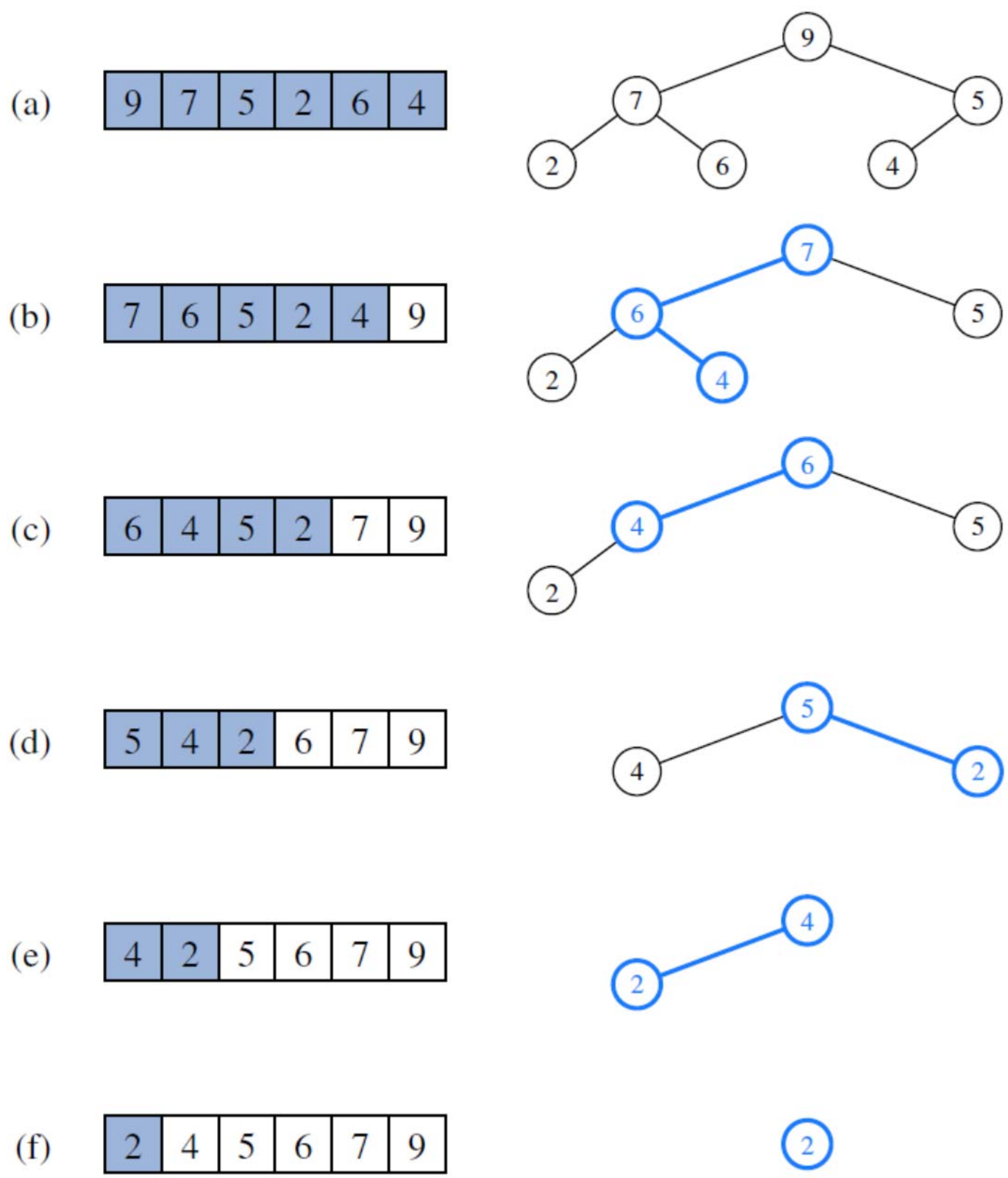
Heap Sort

- Running time
  - Phase 1: the i^{th} *insert* operation takes $O(\log i)$ time $\rightarrow$ this phase takes $O(n \log n)$ time
  - Phase 2: the j^{th} *removeMin* operation takes $O(\log (n - j + 1))$ $\rightarrow$ this phase takes $O(n \log n)$ time.

Heap Sort In-place

- Heap sort without using extra space
  - Redefine heap operations to be maximum-oriented (root is the maximum element)
  - Starting as an empty heap, insert array elements at 0 … *n*-1 to the heap
  - Remove elements from the heap and place them from the end of the array

Heap Sort In-place



Assignment 8

- In this assignment, we will
  - Implement key methods of a heap
  - Find a shortest path using priority queue
  - Play a Pac-Man game

- Download hw8.zip
  - Implement all TODO lines
  - Zip the java files you modified and submit it

- Due date: TBD

Assignment 8

- Java files to update
  - Heap.java: a heap
  - HeapQueue.java: a priority queue using Heap
  - Path.java: a shorted path algorithm
    - The paths with dots are shorter than the paths without them

Assignment 8

- Find who is the Pac-Man champion