Data Structures and Algorithms

Week 4

Priority Queues

- Each element in a queue is associated with a *key*.
- When an element is removed, an element with a *minimal* (or *maximal*) *key* is removed.
- Usually keys are numbers.
- Objects can be used as keys as far as there is a total ordering among those objects.

Priority Queues
ADT

- insert(*k*, *v*): Create an entry with key *k* and value *v* in the priority queue.
- min( ): Returns (but does not remove) an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- removeMin( ): Removes and returns an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- size( ): Returns the number of entries in the priority queue.
- isEmpty( ): Returns true if the priority queue is empty. Returns false, otherwise.

Priority Queues
ADT

- Illustration

Method	Return Value	Priority Queue Contents
insert(17, A)		{(17, A)}
insert(4, P)		{(4, P) , (17, A)}
insert(15, X)		{(4, P) , (15, X), (17, A)}
size( )	3	{(4, P) , (15, X), (17, A)}
isEmpty( )	false	{(4, P) , (15, X), (17, A)}
min( )	(4, P)	{(4, P) , (15, X), (17, A)}
removeMin( )	(4, P)	{(15, X), (17, A)}
removeMin( )	(15, X)	{(17, A)}
removeMin( )	(17, A)	{ }
removeMin( )	null	{ }
size( )	0	{ }
isEmpty( )	true	{ }

Priority Queues
Implementation

- An element in a priority queue has *key* and *value*.
- *Entry* interface is used to store a key-value pair.

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

Priority Queues
Implementation

- *PriorityQueue* interface

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

Priority Queues
Implementation

- Keys must have *total ordering*.
- Total ordering means there is a linear ordering among all keys.

- Total ordering of a comparison rule, $\leq$, satisfies the following properties:
  - Comparability property: $k_1 \leq k_2$ or $k_2 \leq k_1$.
  - Antisymmetric property: If $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$.
  - Transitive property: If $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$.
- If keys have total ordering, *minimal key* is well defined
- key_{min} is a key such that: $key_{min} \leq k$, for all k

Priority Queues
Implementation

- Two ways to compare objects in Java
  - *compareTo* and *compare*
- *compareTo* is defined in *java.util.Comparable* interface.
- A class must override and implement the *compareTo* method.
- *Ordering* defined in the *compareTo* method is called *natural ordering*.
- Usage: *a.compareTo*(*b*) returns
  - a negative number, if a < b
  - zero, if a = b
  - a positive number, if a > b
- Many Java classes implemented *Comparable* interface.

Priority Queues
Implementation

- *compare* is defined in *java.util.Comparator* interface.
- Use this to compare not by natural ordering
- Need to write a separate customized comparator
- Example: To compare strings by length (natural ordering is lexicograhic ordering).
- First, write a customized comparator method

```
public class StringLengthComparator implements Comparator<String> {
   public int compare(String a, String b){
               if (a.length() < b.length()) return -1;
               else if (a.length() == b.length()) return 0;
               else return 1;
   }
}
```

Priority Queues
Implementation

- Then, use it as follows:

```
public class ComparatorTest {
    public static void main(String[] args) {
        StringLengthComparator c = new StringLengthComparator();
        String s1 = "tiger";
        String s2 = "sugar";
        String s3 = "coffee";
        String s4 = "cat";
        System.out.println("Compare s1 and s2: " + c.compare(s1, s2)); // 0
        System.out.println("Compare s1 and s3: " + c.compare(s1, s3)); // -1
        System.out.println("Compare s1 and s4: " + c.compare(s1, s4)); // 1
    }
}
```

Priority Queues
AbstractPriorityQueue Base Class

- Provides common features for different concrete implementations.
- An entry in a priority queue is implemented as *PQEntry*:

```
protected static class PQEntry<K,V> implements Entry<K,V> {
    private K k;  // key
    private V v;  // value
    public PQEntry(K key, V value) {
        k = key;
        v = value;
    }
    public K getKey() { return k; }
    public V getValue() { return v; }
    protected void setKey(K key) { k = key; }
    protected void setValue(V value) { v = value; }
}
```

Priority Queues
Implementing Using a Heap

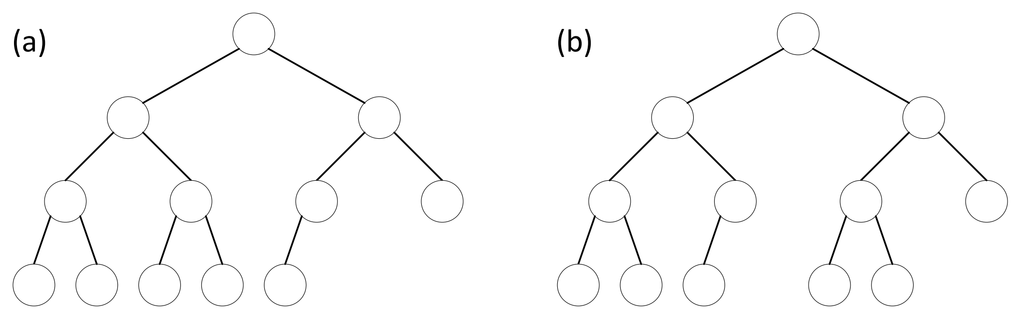
- Implementation with an unsorted list
- Implementation with a sorted list

- We will focus on implementation with *heap*.

- *Heap* is a binary tree with the following properties:
  - *Heap-order property*: In a heap *T*, for every position *p*, except the root, the key stored at *p* is greater than or equal to the key stored at *p*'s parent. (*minimum-oriented* heap)
  - *Complete binary tree property*: A heap is a complete binary tree.

Priority Queues
Implementing Using a Heap

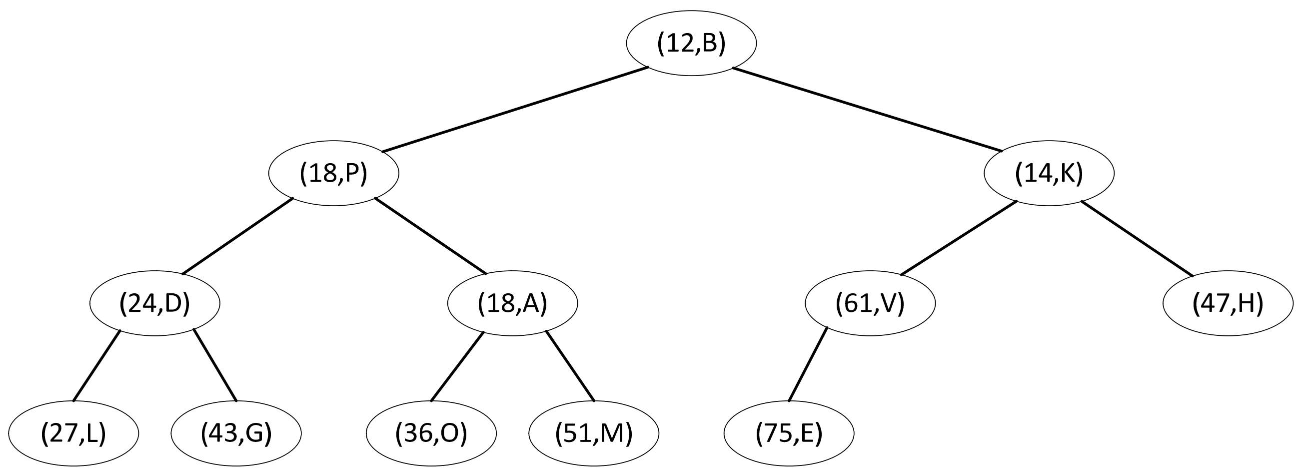
- Complete binary tree
  - Levels 0, 1, . . ., $h - 1$ of T have the maximal number of nodes (in other words, level i has 2^i nodes, where $0 \leq i \leq h - 1$), and
  - Nodes at level h are in the leftmost possible positions at that level.



yes no

Priority Queues
Implementing Using a Heap

- Priority queue implemented using a heap example:



- Height of a heap with *n* entries is $h = \lfloor \log n \rfloor$

Priority Queues

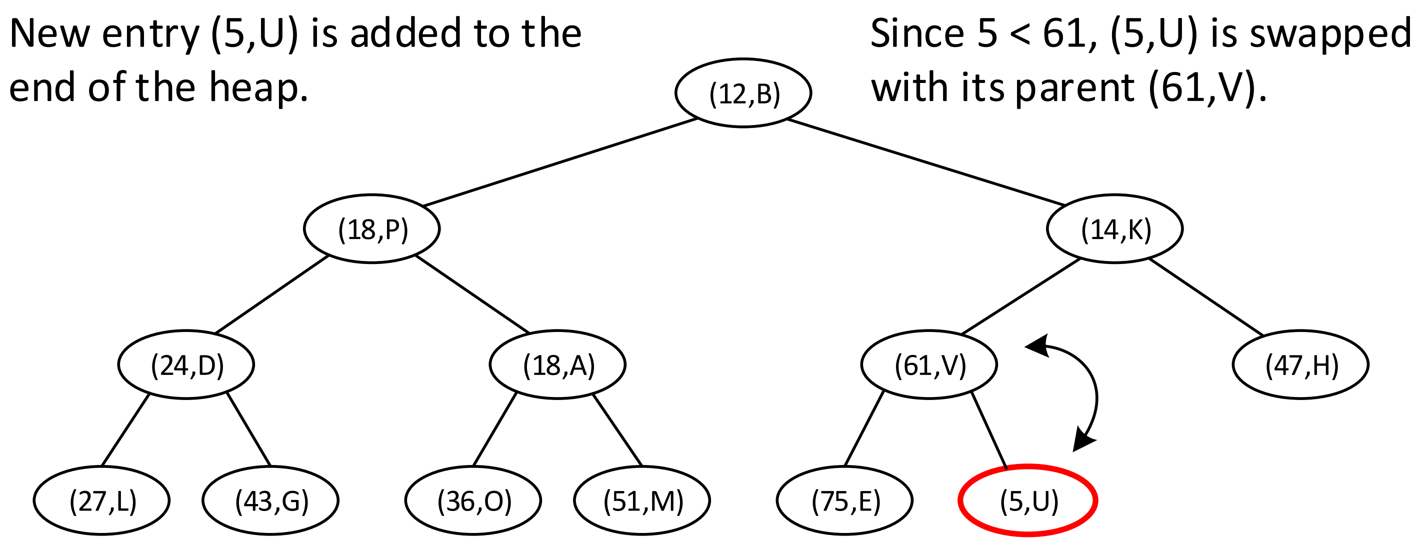
Implementing Using a Heap

- Adding an entry to a heap
  - Step 1: Add new entry at the “end” of the heap
  - Step 2: Reorganize the heap (because adding new entry may violate the heap-order property)

- Reorganization is done by *up-heap bubbling*.

Priority Queues
Implementing Using a Heap

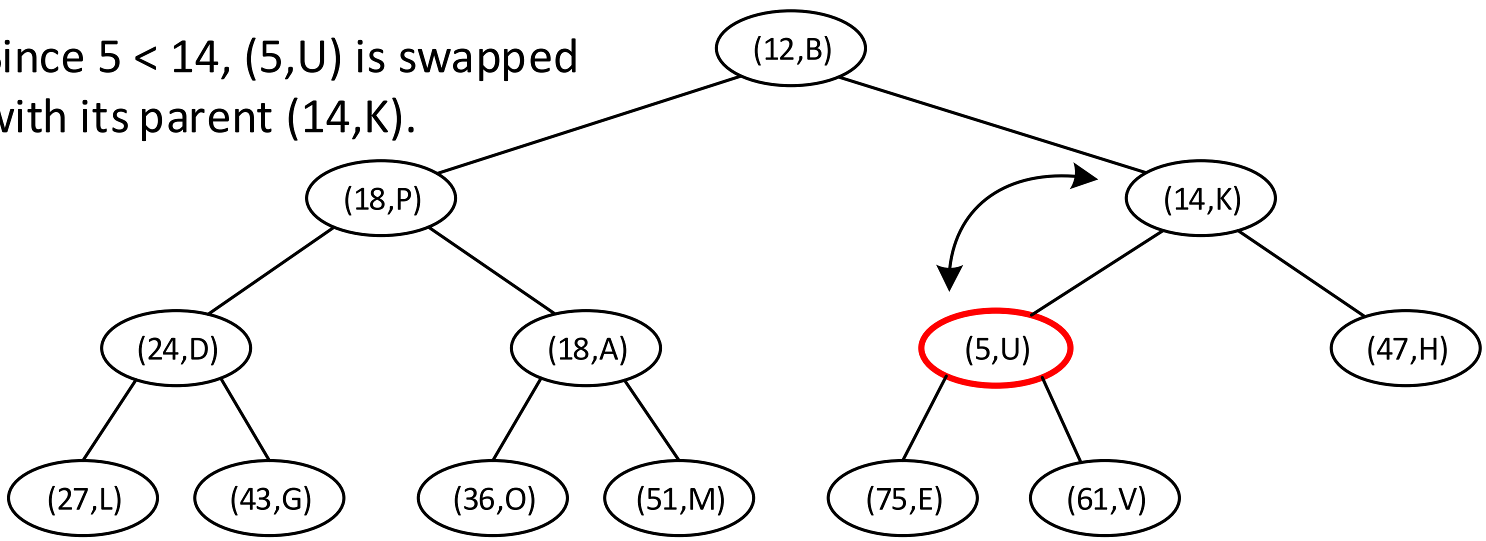
- Illustration



Priority Queues
Implementing Using a Heap

- Illustration

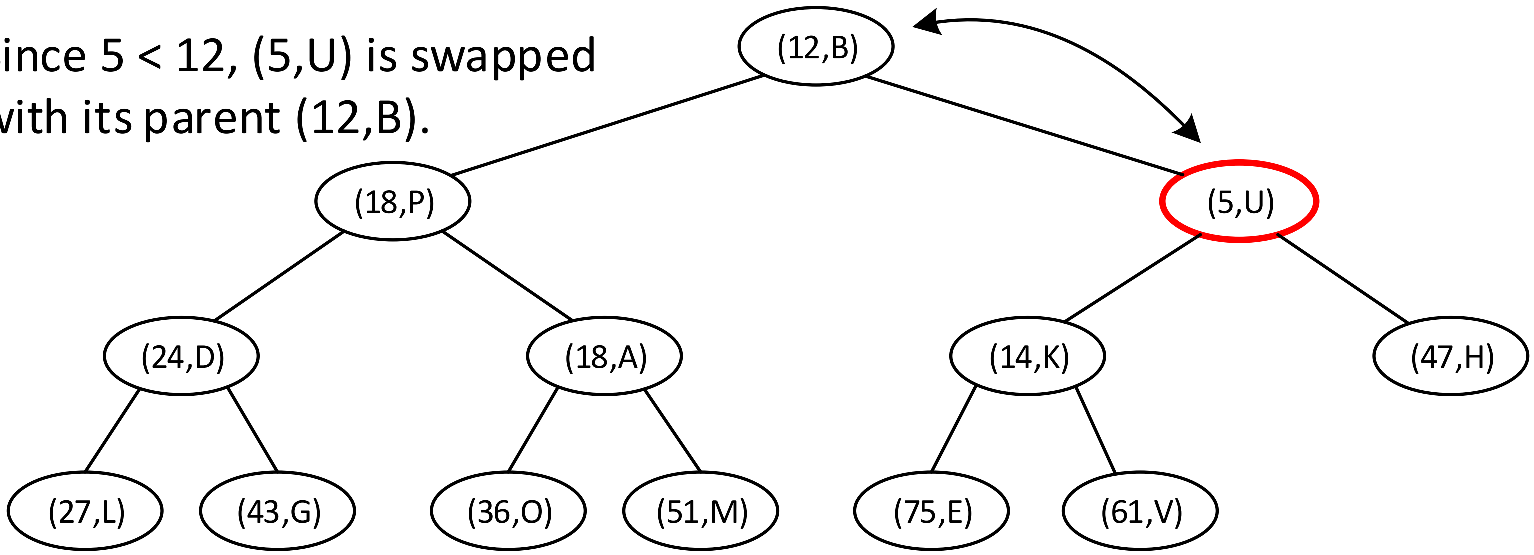
Since 5 < 14, (5,U) is swapped with its parent (14,K).



Priority Queues
Implementing Using a Heap

- Illustration

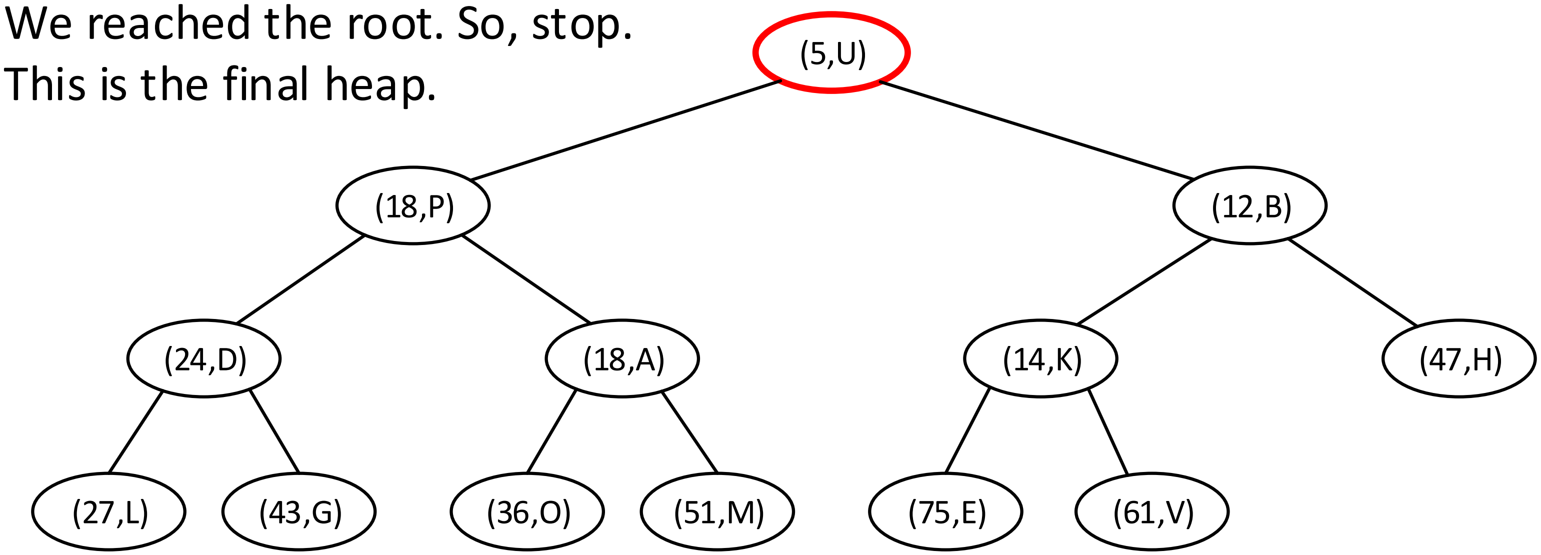
Since 5 < 12, (5,U) is swapped with its parent (12,B).



Priority Queues
Implementing Using a Heap

- Illustration

We reached the root. So, stop.
This is the final heap.



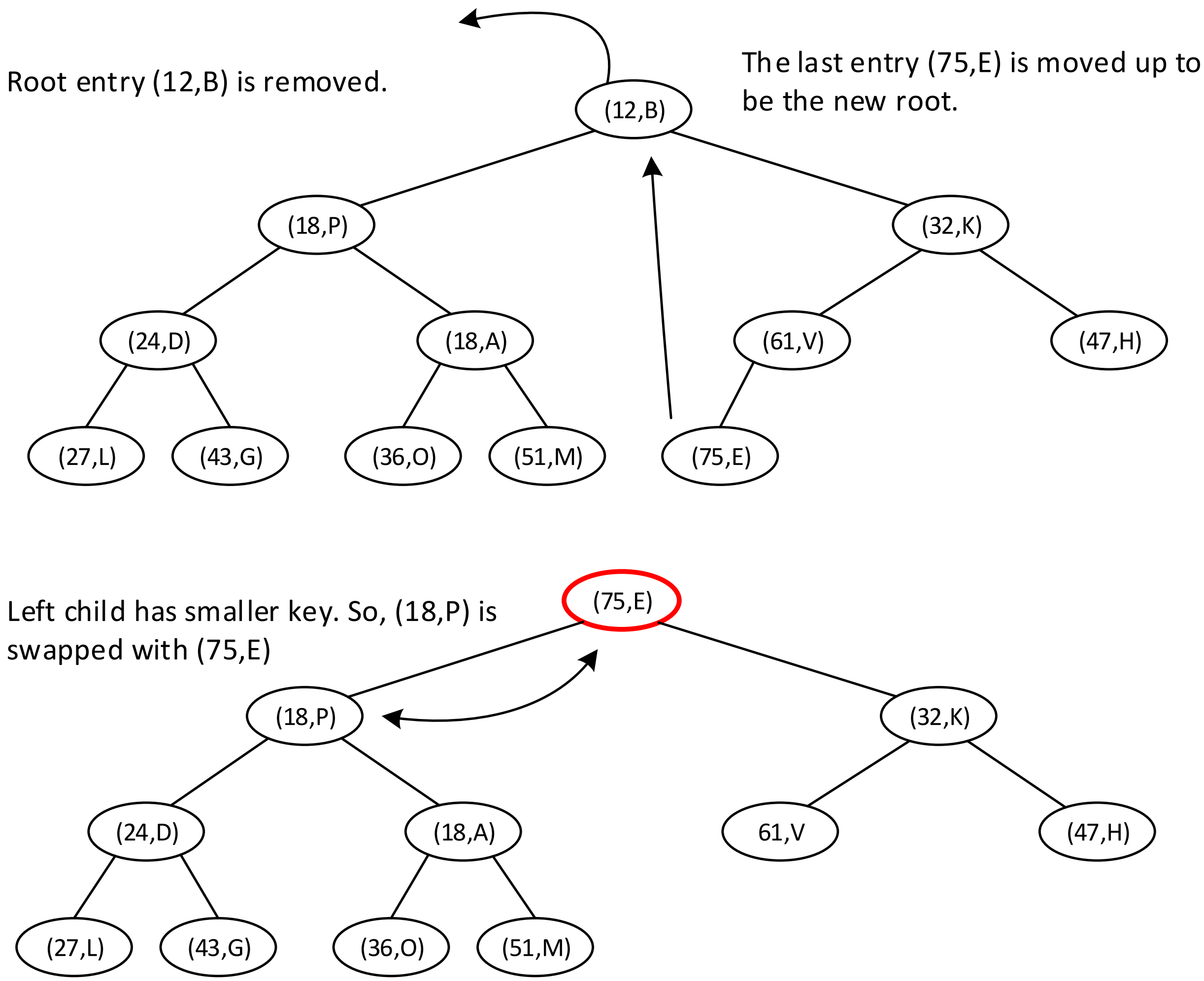
Priority Queues
Implementing Using a Heap

- Removing the entry with minimal key
  – Step1: Remove the root
  – Step 2: Last node is move up to the root and perform *down-heap bubbling*.

- Down-heap bubbling is opposite of up-heap bubbling.

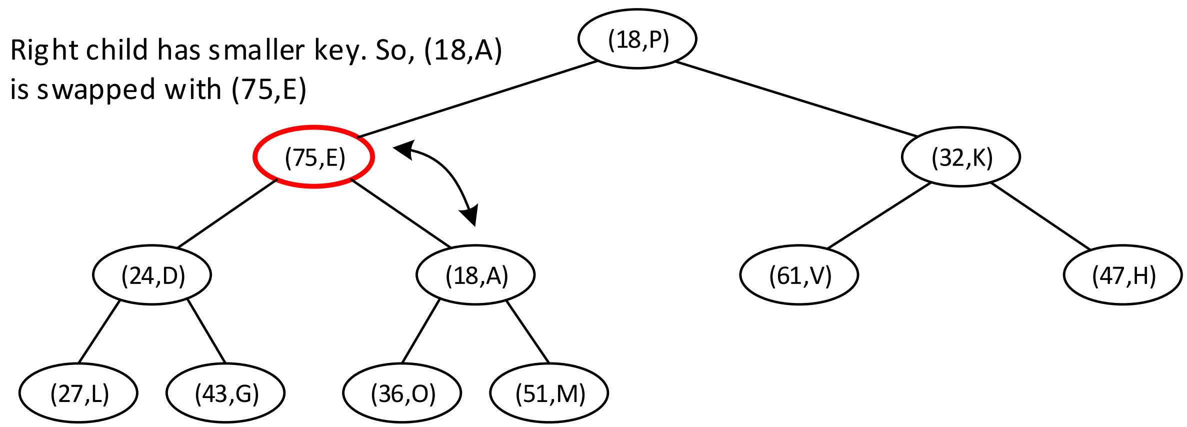
Priority Queues
Implementing Using a Heap

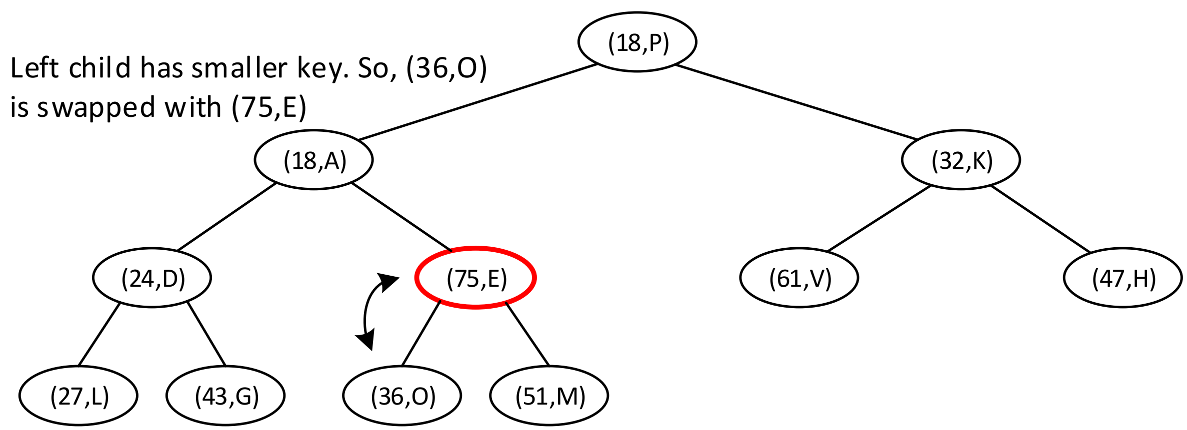
- Illustration



Priority Queues
Implementing Using a Heap

- Illustration

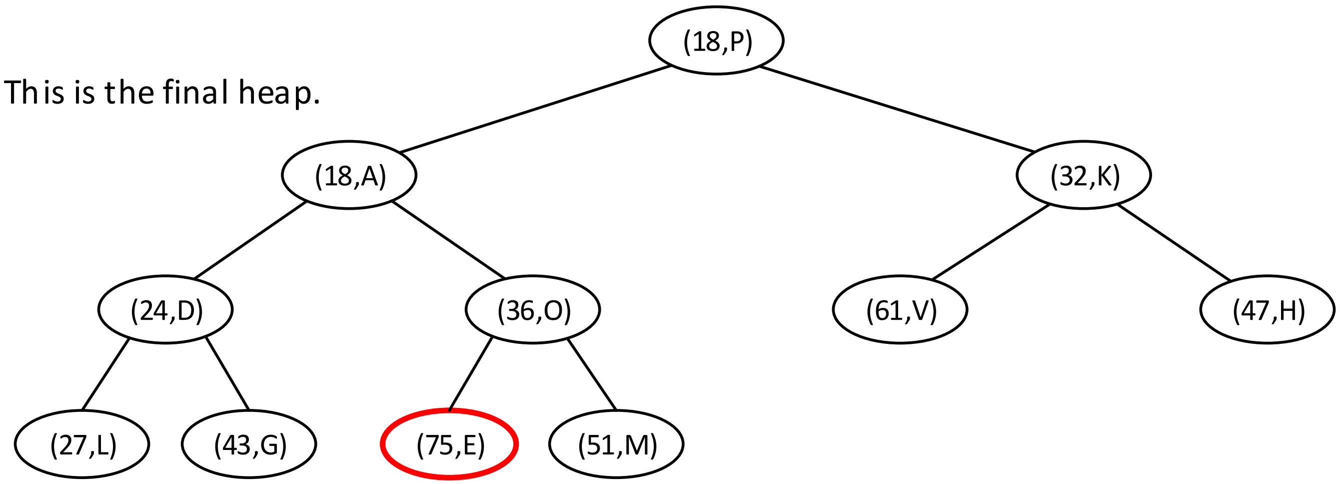




Priority Queues
Implementing Using a Heap

- Illustration

This is the final heap.



Priority Queues
Array-Based Heap

- The level number of a position p, $f(p)$, is defined as follow:
  - If p is the root, $f(p) = 0$
  - If p is the left child of position q, $f(p) = 2*f(q) + 1$
  - If p is the right child of position q, $f(p) = 2*f(q) + 2$
- The level number is used as the index in an array where the entry with position p is stored.

Priority Queues

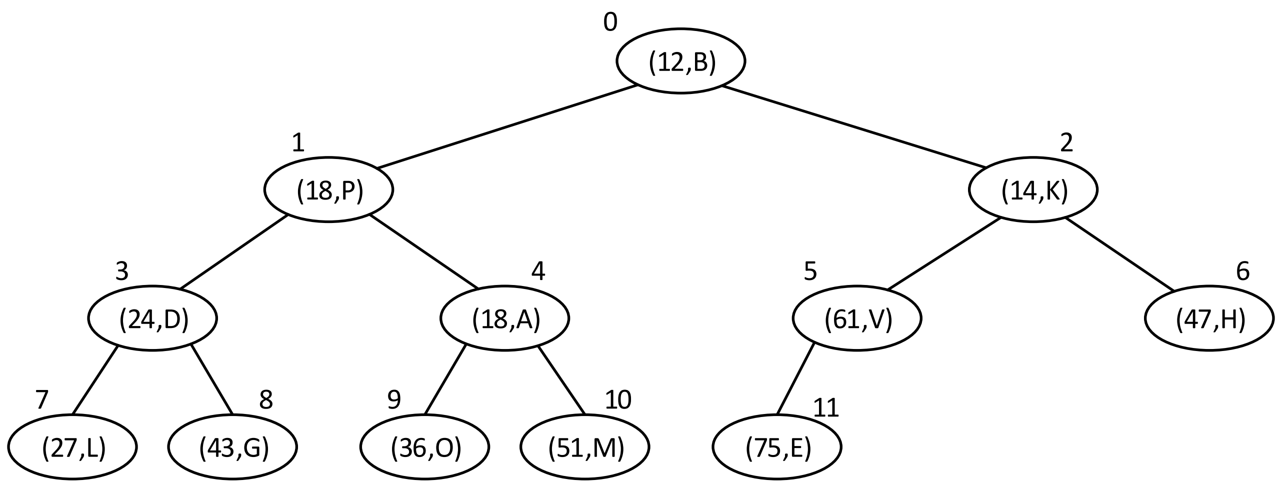
Array-Based Heap

- Then, the entry at position p is stored in $A[f(p)]$.

- Index of the root node is 0.
- Index of left child of p = $2*f(p) + 1$
- Index of right child of p = $2*f(p) + 2$
- Index of parent of p = $\lfloor (f(p)-1)/2 \rfloor$

Priority Queues
Array-Based Heap

- Example



(12,B)	(18,P)	(14,K)	(24,D)	(18,A)	(61,V)	(47,H)	(27,L)	(43,G)	(36,O)	(51,M)	(75,E)
0	1	2	3	4	5	6	7	8	9	10	11

Priority Queues
Array-Based Heap

- *HeapPriorityQueue* class implements a priority queue using a heap.
- A heap is implemented using *ArrayList*.
- Will briefly look at *upheap*, *downheap*, *insert*, and *removeMin* methods.

- *HeapPriorityQueue.java* code

Priority Queues
Analysis of Heap-Based Priority Queue

- insertion:
  - *upheap* method takes $O(\log n)$
  - So, insertion takes $O(\log n)$
- removeMin:
  - *downheap* method takes $O(\log n)$
  - So, removeMin takes $O(\log n)$

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

Priority Queues
Bottom-up Heap Construction

- Given *n* elements, we can build a heap with *n* successive insertions => takes $O(n \log n)$ time.

- O(*n*) time algorithm
- Assume $n = 2^{h+1} - 1$ (or every level is full)
- Step 1: Build $(n + 1) / 2$ heaps at height 0
- Step 2: Build $(n + 1) / 4$ heaps at height 1

  . . .
- Step *i*: Build $(n + 1) / 2^i$ heaps at height $i - 1$

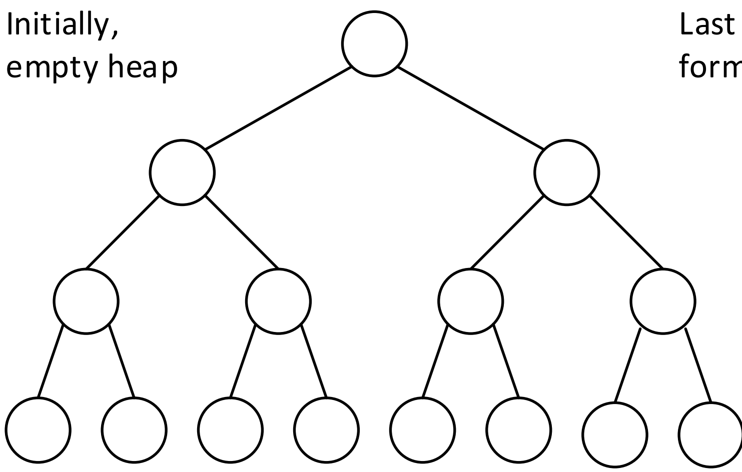
  . . .
- Step $h + 1$: A single heap is formed at height *h*.

Priority Queues
Bottom-up Heap Construction

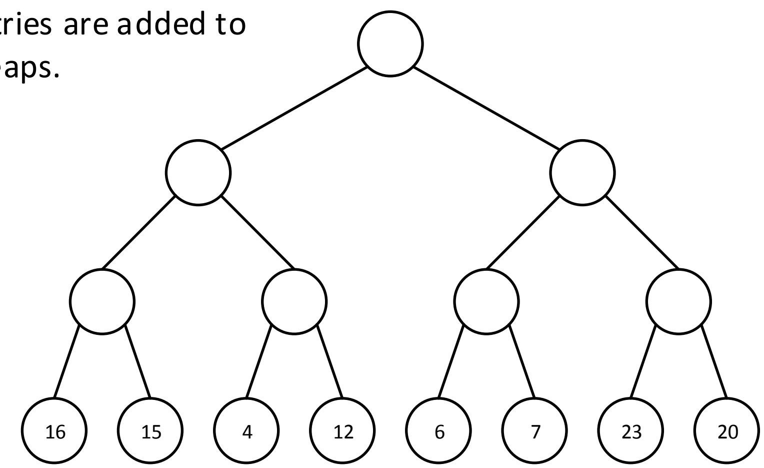
- Illustration

Given sequence of keys: 14, 5, 8, 25, 9, 11, 17, 16, 15, 4, 12, 6, 7, 23, 20

Initially,
empty heap

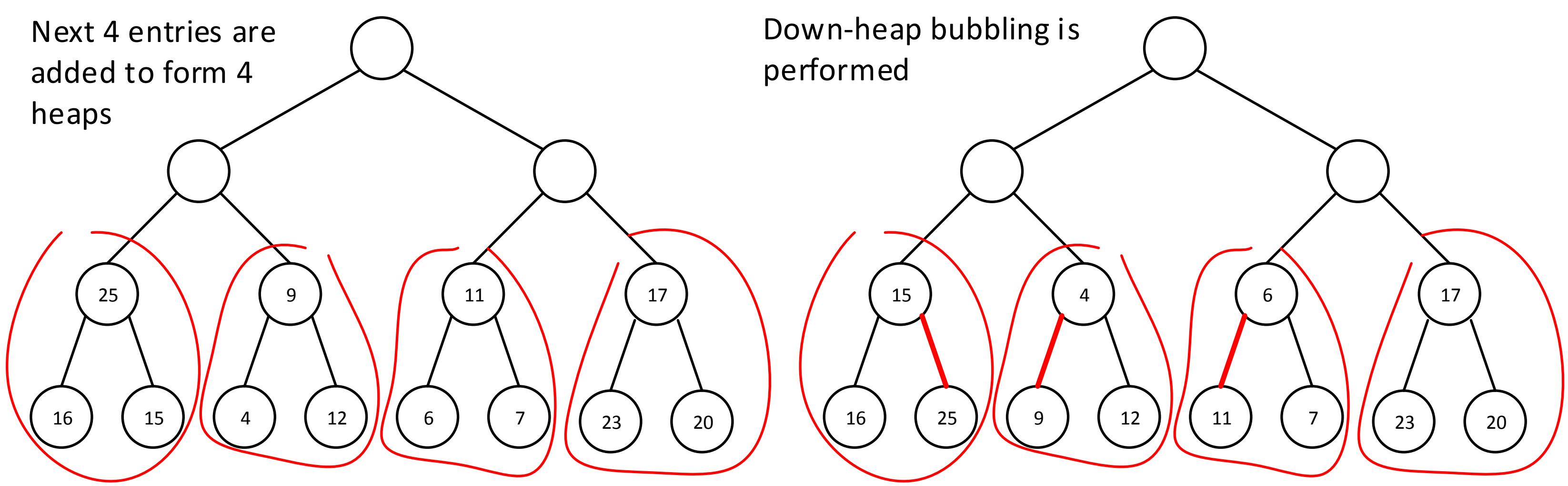


Last 8 entries are added to
form 8 heaps.



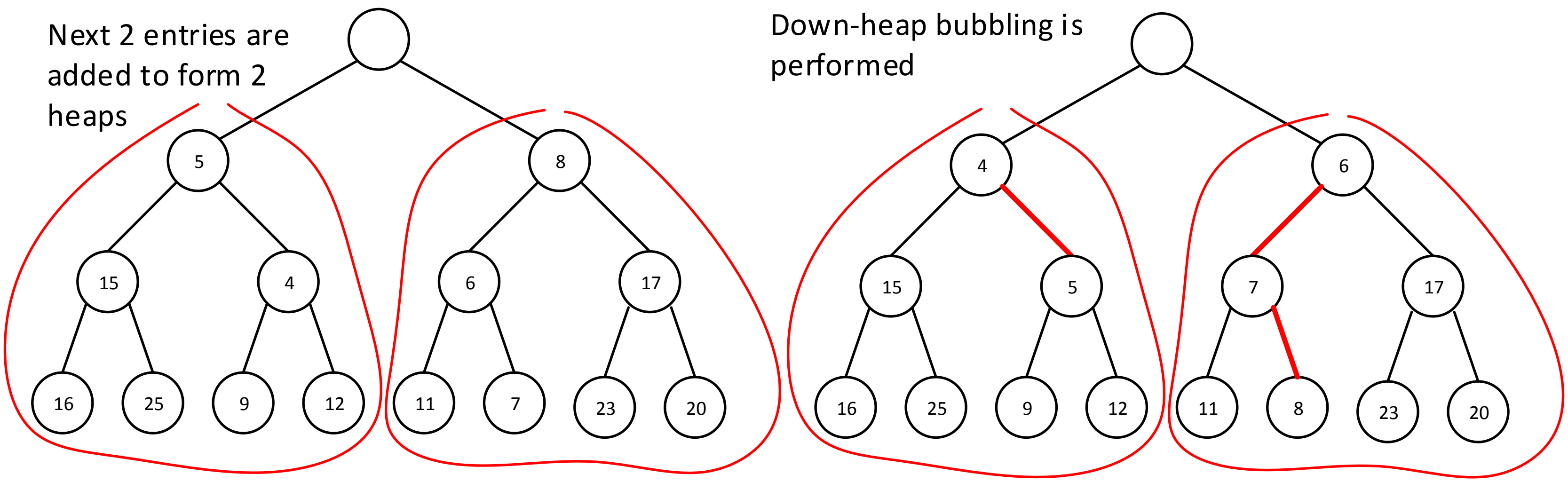
Priority Queues
Bottom-up Heap Construction

- Illustration



Priority Queues
Bottom-up Heap Construction

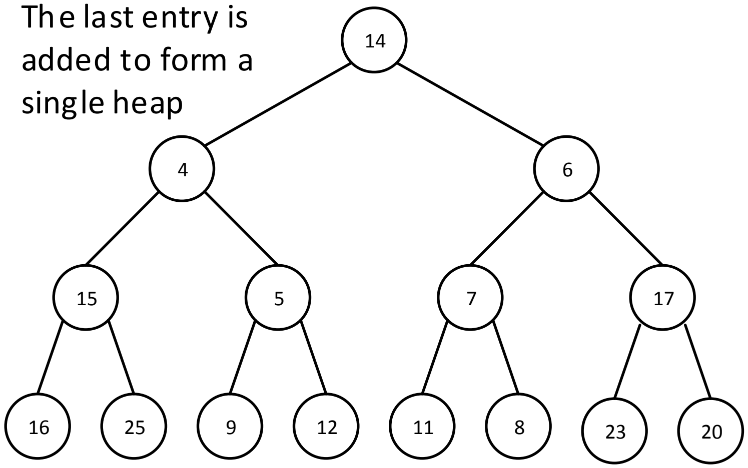
- Illustration



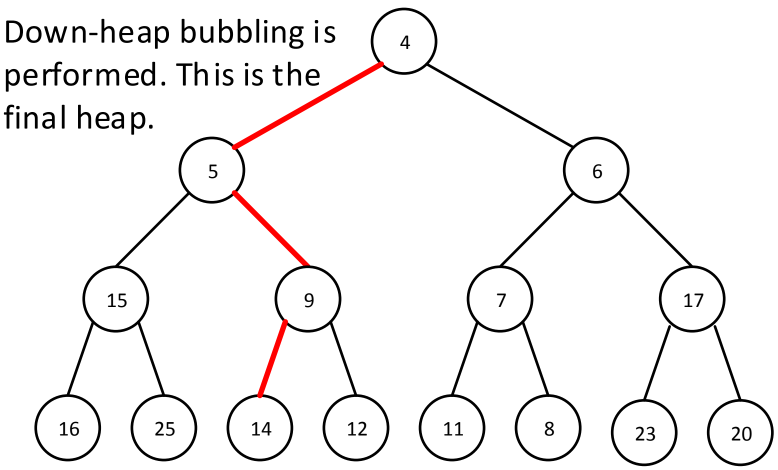
Priority Queues
Bottom-up Heap Construction

- Illustration

The last entry is added to form a single heap



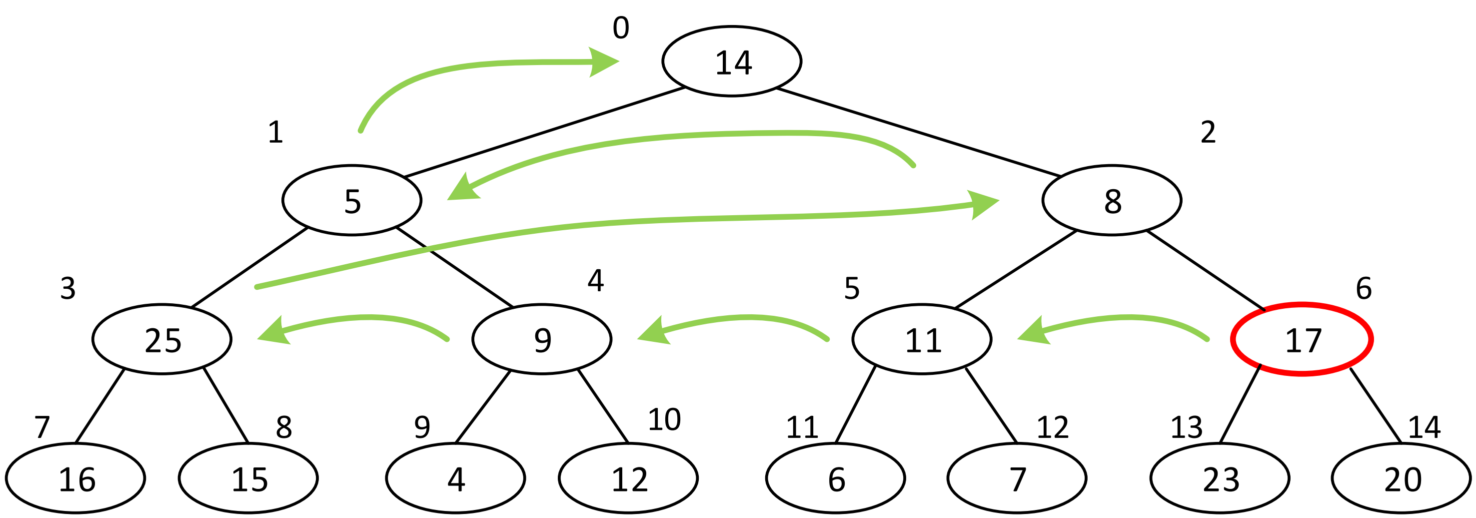
Down-heap bubbling is performed. This is the final heap.



Priority Queues
Bottom-up Heap Construction

- Java implementation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	5	8	25	9	11	17	16	15	4	12	6	7	23	20



Priority Queues
Bottom-up Heap Construction

- Java implementation

```
public HeapPriorityQueue(K[ ] keys, V[ ] values) {
  super();
  for (int j=0; j < Math.min(keys.length, values.length); j++)
      heap.add(new PQEntry<>(keys[j], values[j]));
  heapify();
}

protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
      downheap(j);
}
```

Priority Queues
Java's Priority Queue

- *java.util.PriorityQueue*
- An entry is a single element.
- Some operations in Java's *PriorityQueue*
  - add(E e): Inserts the specified element *e* to the priority queue.
  - isEmpty( ): Returns true if the priority queue contains no element.
  - peek( ): Retrieves, but does not remove, a minimal element from the priority queue.
  - remove( ): Removes a minimal element from the priority queue.
  - size( ): Returns the number of elements in the priority queue.

Priority Queues
Heap-Sort

- Uses array-based heap data structure.
- In-place sorting: no additional storage is used.
- Uses a *maximum-oriented* heap.
- *maximum-oriented* heap: In a heap T, for every position p, except the root, the key stored at p is *smaller* than or equal to the key stored at p's parent.
- Sorting steps:
  1. Given n elements are inserted into a maximum-oriented heap.
  2. Repeat the following until only one node is left in the heap:

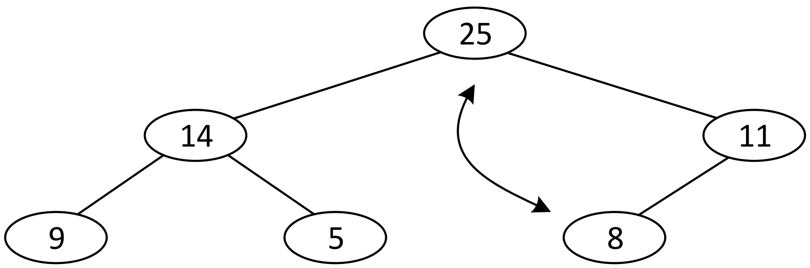
     Root is swapped with the last node, heap size is decremented, perform down-heap bubbling.

Priority Queues
Sorting with Priority Queue

- Illustration

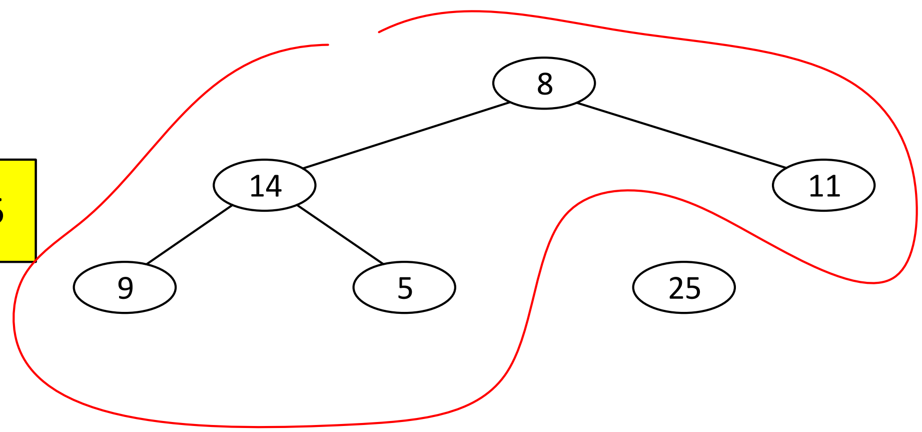
The maximum-oriented heap after the first step.
The root node and the last node is swapped.
Heap size is decremented.

25	14	11	9	5	8



Down-heap bubbling is applied on the root.

8	14	11	9	5	25

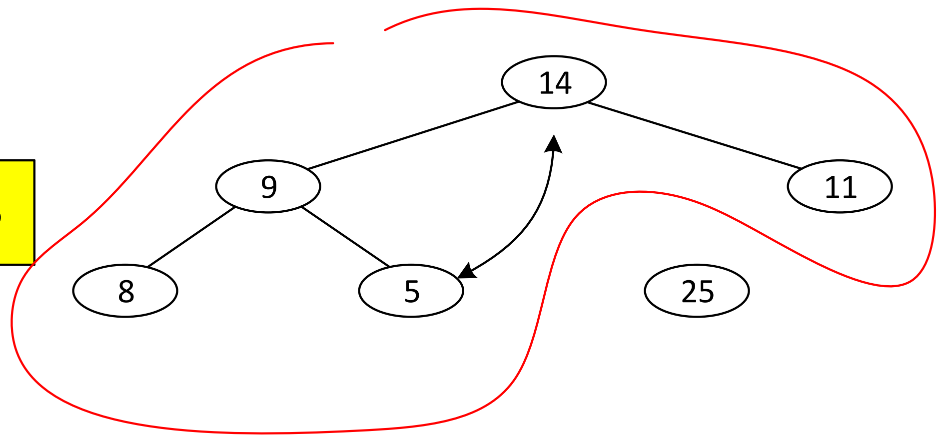


Priority Queues
Sorting with Priority Queue

- Illustration

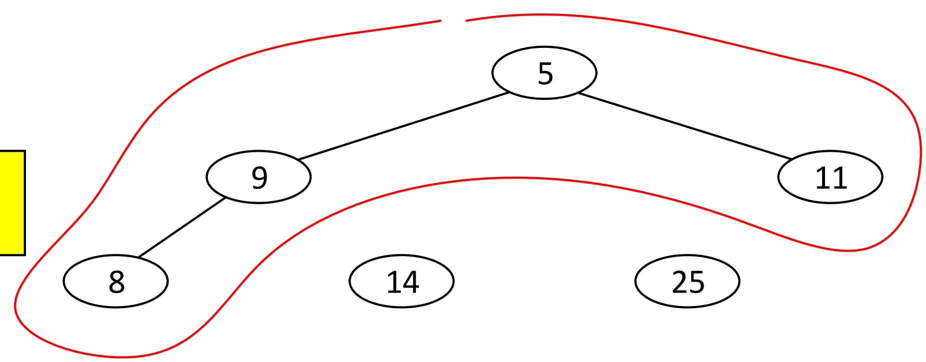
The root node is swapped with the last node. Heap size is decremented.

14	9	11	8	5	25



Down-heap bubbling is applied on the root.

5	9	11	8	14	25



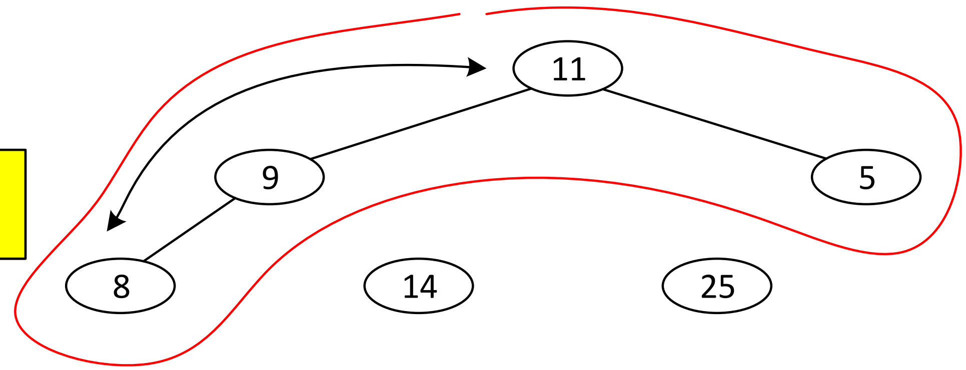
Priority Queues

Sorting with Priority Queue

- Illustration

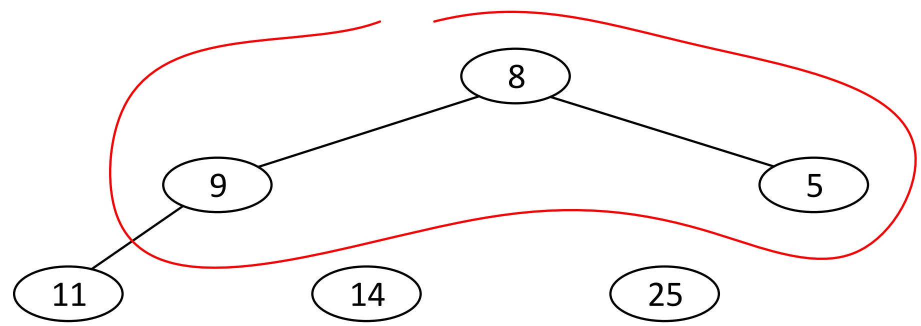
The root node is swapped with the last node. Heap size is decremented.

11	9	5	8	14	25



Down-heap bubbling is applied on the root.

8	9	5	11	14	25



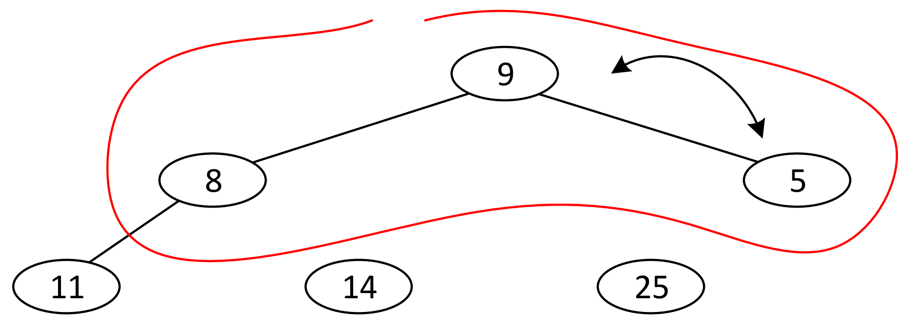
Priority Queues

Sorting with Priority Queue

- Illustration

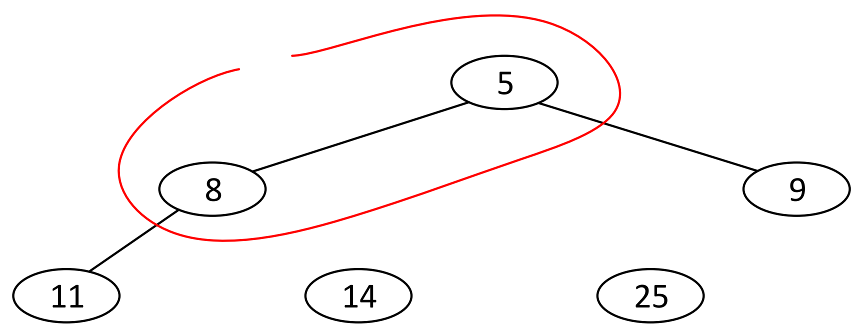
The root node is swapped with the last node. Heap size is decremented.

9	8	5	11	14	25



Down-heap bubbling is applied on the root.

5	8	9	11	14	25

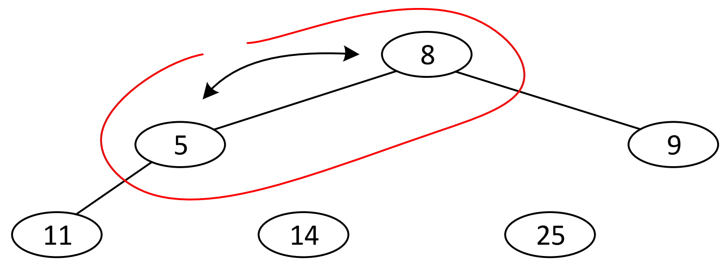


Priority Queues
Sorting with Priority Queue

- Illustration

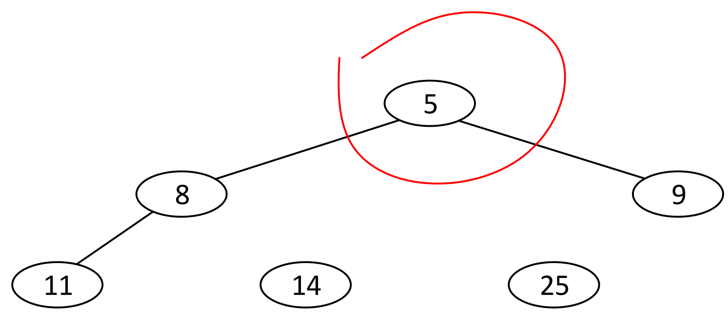
The root node is swapped with the last node. Heap size is decremented.

8	5	9	11	14	25



At this time the array is sorted.

5	8	9	11	14	25



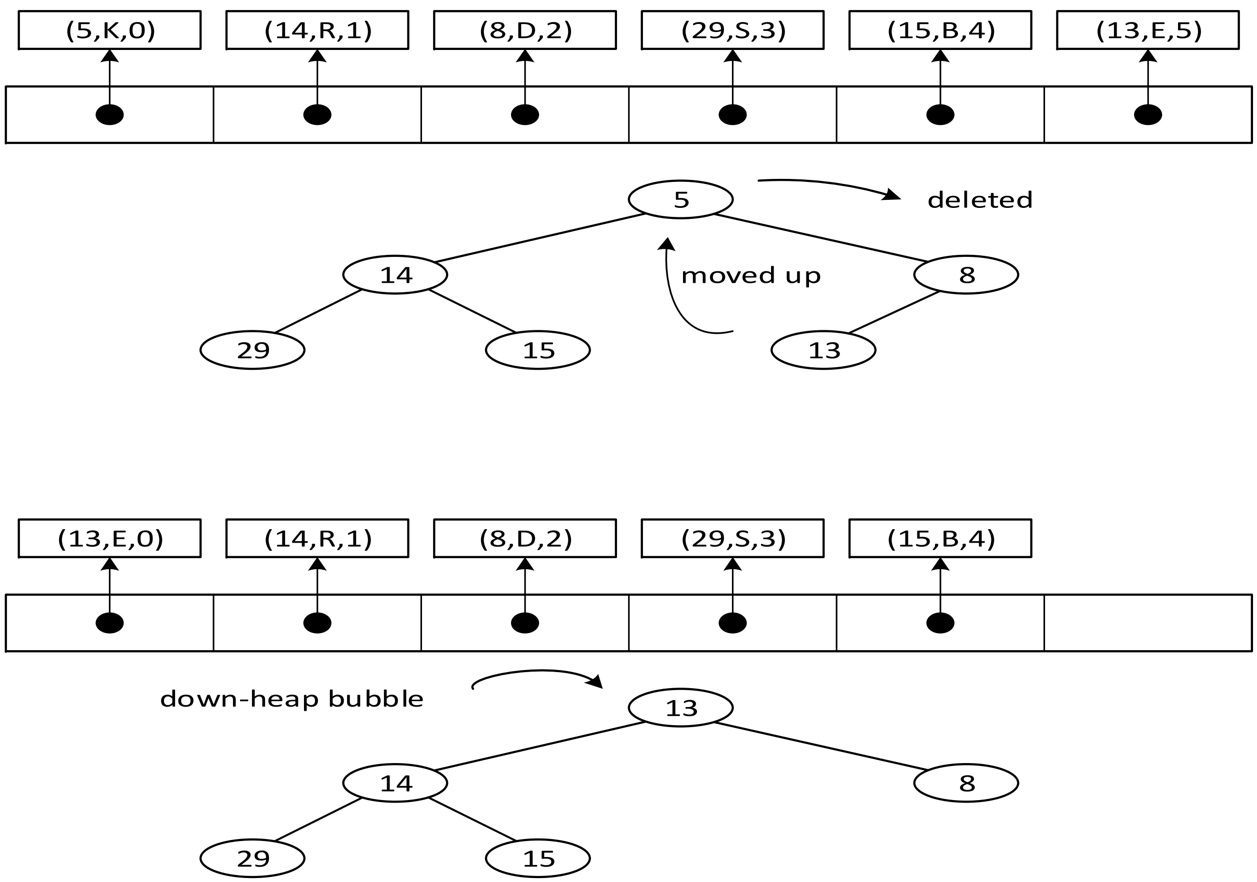
Priority Queues
Adaptable Priority Queue

- Can remove arbitrary entry (not just the root).
- Can replace the key of an entry.
- Can replace the value of an entry.

- Uses location-aware entities to find an entry in a priority queue efficiently.
- Location-aware entry keeps one more field, current index of the entry in an array-based heap.

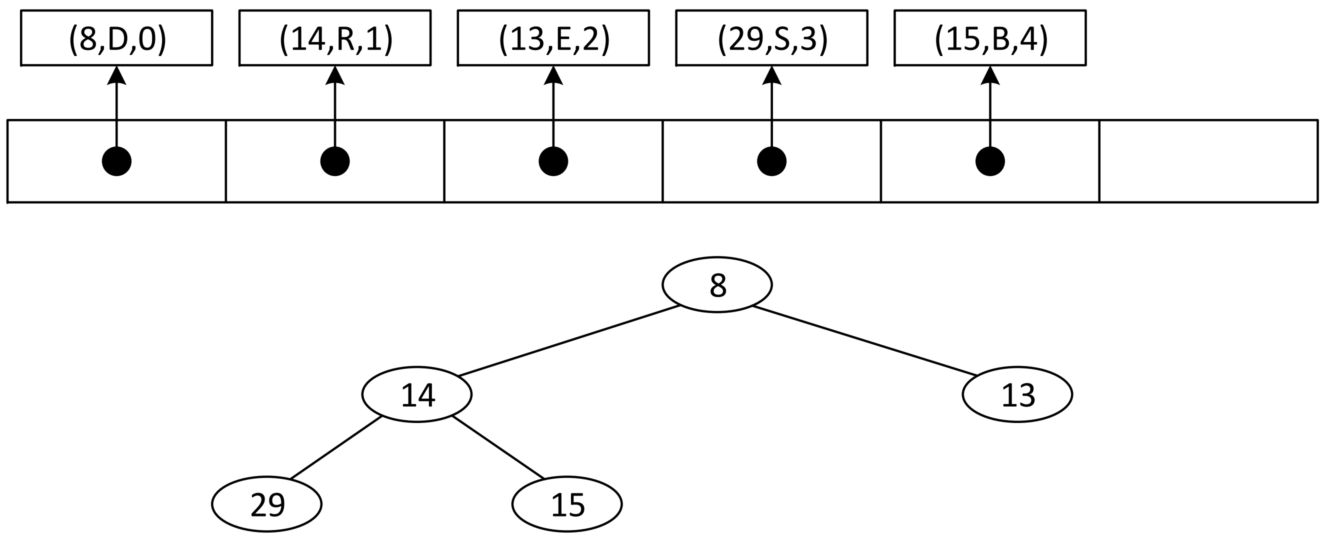
Priority Queues
Adaptable Priority Queue

- Illustration of *removeMin*



Priority Queues
Adaptable Priority Queue

- Illustration of *removeMin*



Priority Queues
HeapAdaptablePriorityQueue Class

- Extends *HeapPriorityQueue* class.
- An entry in the queue

```
protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
  private int index;          // entry's current index within the heap
  public AdaptablePQEntry(K key, V value, int j) {
      super(key, value);        // this sets the key and value
      index = j;                // this sets the new field
  }
  public int getIndex() { return index; }
  public void setIndex(int j) { index = j; }
}
```

Priority Queues
HeapAdaptablePriorityQueue Class

- *HeapAdaptablePriorityQueue.java* code.

Maps

- *Map* is a data structure to efficiently store and retrieve values based on *search keys*.
- Map stores (*key*, *value*) pairs.
- Each (*key*, *value*) pair is called an *entry*.
- Keys are unique.
- Maps are also known as *associative arrays*.

- Applications:
  - (movie title, movie information)
  - (part number, part information)
  - (reservation number, reservation information)
  - (student id, student information)

Maps
Map ADT

- size( ): Returns the number of entries in *M*.
- isEmpty( ): Returns true if *M* is empty. Returns false, otherwise.
- get(*k*): Returns the value *v* associated with the key *k*, if such entry exists. Returns null, otherwise.
- put(*k*, *v*): If there is no entry in *M* with a key equal to *k*, then adds the entry (*k*, *v*) to *M* and returns null. Otherwise, replaces the existing value associated with the key *k* with *v* and returns the old value.

Maps
Map ADT

- remove(*k*): Removes from *M* the entry with the key *k* and returns its value. If there is not entry in *M* with the key *k*, returns null.
- keySet( ): Returns an iterable collection containing all keys in *M*.
- values( ): Returns an iterable collection containing all values in *M*. If multiple keys map to the same value, then the value appears multiple times in the returned collection.
- entrySet( ): Returns an iterable collection containing all (*key*, *value*) entries in *M*.

Maps
Map ADT

- Map interface

```
public interface Map<K,V> {
  int size();
  boolean isEmpty();
  V get(K key);
  V put(K key, V value);
  V remove(K key);
  Iterable<K> keySet();
  Iterable<V> values();
  Iterable<Entry<K,V>> entrySet();
}
```

- Note: *java.util.Map* interface provides more extensive set of operations than those defined above.

Maps
Map ADT

- Simple application example: Word Frequency

  - Counts frequency of each word in a text.

  - Create an empty map.
  - In the map, an entry is (word, frequency) pair.
  - Read one word at a time.
  - If the word is not in the map, insert it and set frequency = 1
  - If the word is already in the map, increment the frequency of the word.

- *WordCount.java* code.

Maps
Hash Tables

- *Hash table* is an efficient implementation of a map.

- Consider a map that stores *n* entries.
- Assume keys are integers in the range [0, *N* – 1] and values are characters, usually *N* ≥ *n*.
- We can design a lookup table of length *N* as follows, where keys are used as indexes:

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Lookup table's capacity N = 11
Currently there are 4 entries: (1,D), (3,Z), (6,C), and (7,Q)

Maps
Hash Tables

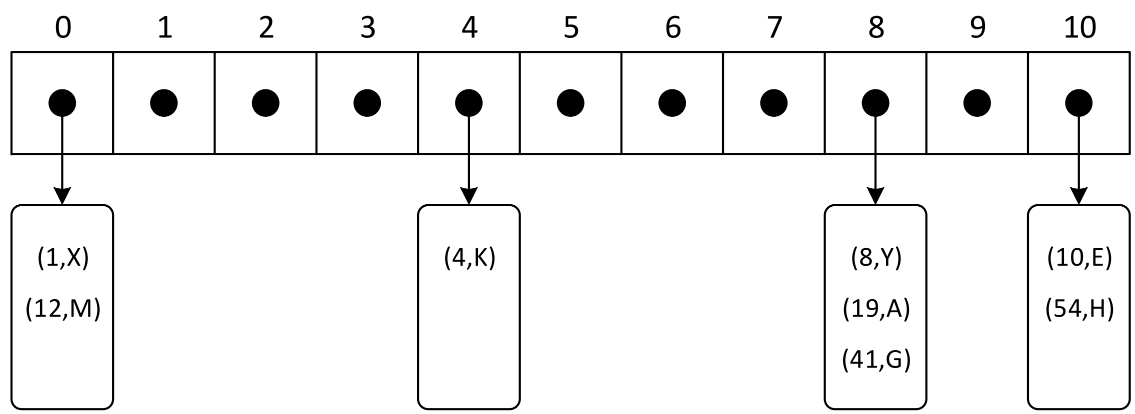
- Issues:
  - The domain of keys may be much larger than the actual number of elements to be stored in the table, i.e., $N >> n$. This is a waste of space.
  - Keys may not be integers. Then, they cannot be used as indexes in the table.

- Solution:
  - Use a *hash function* that maps keys to integers in the range $[0, N - 1]$, distributing keys relatively evenly.
  - N doesn't have to be very large (could be smaller).

Maps
Hash Tables

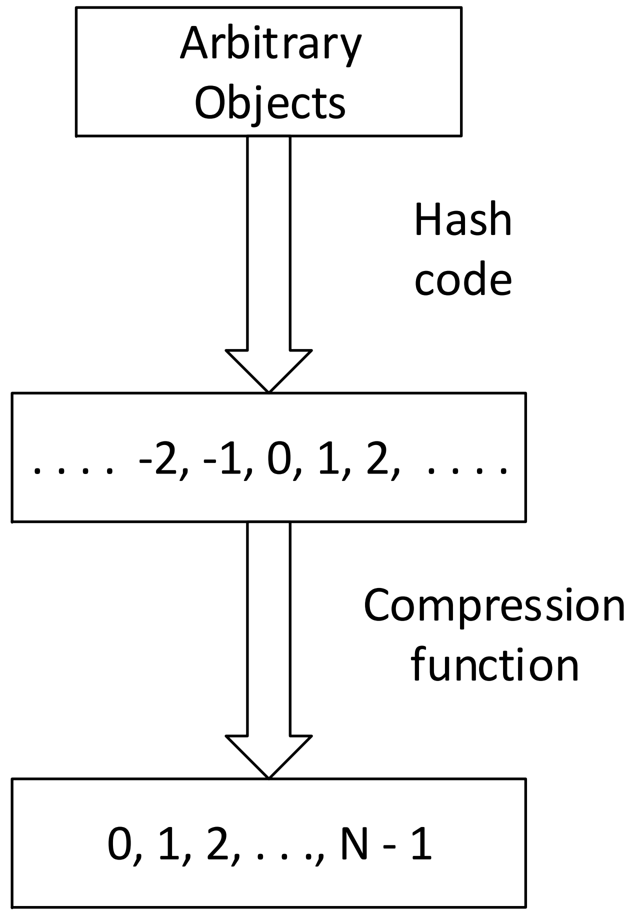
- Ideally a hash function distributes keys evenly across the table.
- In practice, some keys are mapped to the same location.

- One solution: each slot in the table keeps a *bucket* which stores a collection of entries. This table is called *bucket array*.



Maps
Hash Function

- Two step process:
  - *Hash code* maps keys of arbitrary object type to integers. The resulting integer is also called *hash code*.
  - *Compression function* maps the hash code to integers in the range [0, N – 1]



Maps
Hash Code

- Treat bit representation of base types as integers
- Polynomial hash code: used for strings or variable-length objects
- Cyclic-shift hash code: a variant of polynomial hash code

- Java has a default *hashCode*( ) function defined in the *Object* class, which returns a 32-bit integer of ***int*** type.
- When designing a hashCode() for a user-defined class, make sure: If *x*.*equals*(*y*), *x*.*hashCode*( ) = *y*.*hashCode*( )

Maps
Compression Function

- When two keys are mapped to the same hash table index, it is called *collision*.
- A good compression function must distribute hash codes (of keys) relatively uniformly across the hash table to minimize collisions.
- Will discuss two compression functions (compression functions are often called just *hash functions*):
  - *division* method
  - *MAD* (*multiply-add-and-divide*) method

Maps
Compression Function

- Division method: $i \bmod N$,

  where i is an integer (such as a hash code) and N is the hash table size.

- *MAD* method: $[(ai + b) \bmod p] \bmod N$,

  where N is hash table size, p is a prime number larger than N, and a and b are integers in $[0, p-1]$, $a > 0$.

```
private int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode( )*scale + shift)
                                        % prime) % capacity);
}
```

Maps
Compression Function

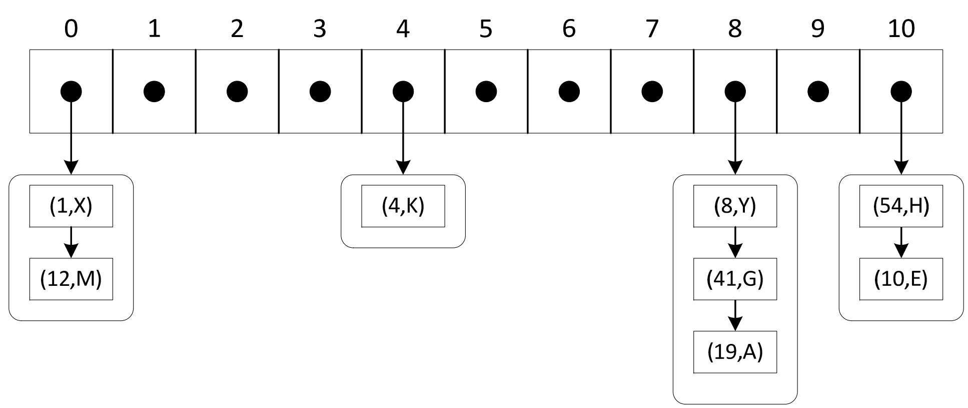
- *MAD* method is better (in terms of well distributing keys across the has table), but division method is more efficient.

Maps

Collision Handling

- When two keys are mapped to the same slot in the hash table, it is called *collision*.
- Will discuss two collision resolution approaches: *chaining* and *open addressing*.

- Chaining: Each slot in the table keeps an unsorted list and all keys that are mapped to the same slot are kept in the list.



Maps
Chaining Method

- Advantage: Easy to implement
- Drawback:
  - Additional storage
  - In the worst case, all keys are stored in the same list, which increases running time.
- Running time
  - Load factor $\lambda = n / N$, which is expected size of a list.
  - Map operations run in $O(\lceil n/N \rceil)$ or $O(\lambda)$
  - If keys are well distributed, $O(\lambda)$ = O(1) and running time is $O(1)$.
  - In the worst case, $O(n)$.

Maps
Open Addressing

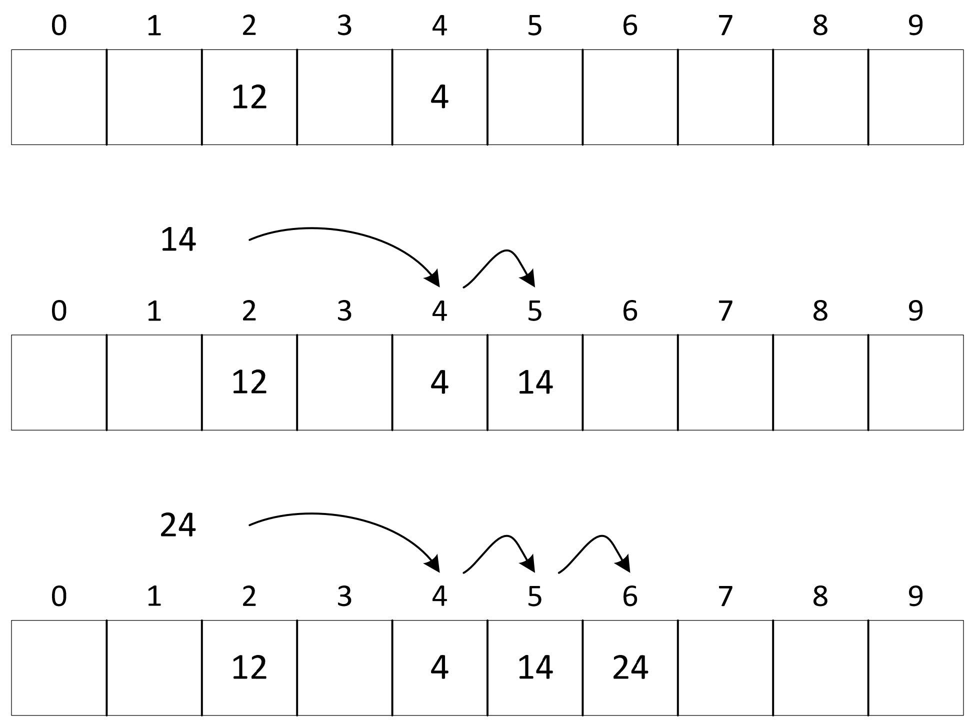
- All entries are stored in a hash table itself.
- No additional data structure and no additional storage space is needed.
- When adding a new key causes a collision, an alternative location in the table is found and the new element is stored in that location.
- Will briefly discuss three open addressing techniques – *linear probing*, *quadratic probing*, and *double hashing*.

Maps
Linear Probing

- Assume A is the array of a hash table.
- Inserting an entry (k, v).
  - Hash function h is applied to key k, i.e., $j \leftarrow h(k)$. We say k is mapped to j.
  - If $A[j]$ is empty, then the entry is stored there, i.e., $A[j] \leftarrow (k, v)$.
  - If that slot is occupied, the next bucket $A[j+1]$ is *probed* to see whether it is available.
  - If it is empty, the entry is stored there. Otherwise, the next bucket, $A[j+2]$, is probed, and so on, until an empty slot is found or all slots have been probed.
  - The sequence of slots probed , called *probe sequence*, is determined by $A[(j+i) \bmod N]$, for $i = 0, 1, 2, \ldots, N - 1$.
  - i is called *probe number*.

Maps
Linear Probing

- Illustration: N = 10, $h = k$ mod N, keys are added in the following order: 4, 12, 14, 24.



Maps
Linear Probing

- Searching an entry with key = *k*.
  - A key *k* is mapped to the array index *j,* i.e., $j \leftarrow h(k)$.
  - If $A[j]$ is empty, then conclude the entry is not in the hash table.
  - If that slot is occupied and it has the entry with *k*, then the entry is found.
  - If the slot is occupied and the key of the entry in the slot is not *k,* the next bucket, $A[j+2]$, is probed, and so on, until the entry is found or all slots have been probed.

Maps
Linear Probing

- Deleting an entry:
  - Assume initially all slots are empty.
  - Assume we want to remove an entry in *A*[*j*].
  - We cannot simply remove the entry in *A*[*j*].
  - Assume the current table is:

0	1	2	3	4	5	6	7	8	9
		12		4	14	24			

  - And, we delete an entry with key = 14.

Maps
Linear Probing

- After deleting entry with key = 14

0	1	2	3	4	5	6	7	8	9
		12		4		24			

- Search entry with key = 24

  24 is mapped to A[4]; occupied; A[5] is probed; empty; conclude entry with key = 24 is not in the table => this is wrong.

search 24

0	1	2	3	4	5	6	7	8	9
		12		4		24			

Maps
Linear Probing

- Solution: Put a “special object” or a “*defucnt*” object in the slot from which an entry is deleted.
- For example, place ϕ in the slot when an entry is removed.
- After removing entry with key = 14

0	1	2	3	4	5	6	7	8	9
		12		4	ϕ	24			

- When inserting, the slot with ϕ is considered empty.
- When searching and entry with key = *k*, the slot with ϕ is considered having an entry with a key ≠ *k*.

Maps
Linear Probing

- Linear probing tends to create *primary clustering*.
- A cluster is a contiguous occupied slots.
- Once a cluster is formed, it tends to grow, which is called *primary clustering*.

Maps
Quadratic Probing

- Uses a quadratic function to determine the next slot to probe.
- Example: Probe sequence is determined by

  $A[(h(k) + f(i)) \bmod N]$, for $i = 0, 1, 2, \ldots, N - 1$,

  where $f(i) = i^2$
- Assume that we are inserting a key 24 and it is mapped to $A[4]$, and that it is occupied. Then, the probe sequence is:

  $A[(4 + 1^2) \bmod 10] = A[5]$,

  $A[(4 + 2^2) \bmod 10] = A[8]$,

  $A[(4 + 3^2) \bmod 10] = A[3]$,

  . . . .

Maps
Quadratic Probing

- Quadratic hashing does not have primary clustering.
- But, it still has clustering problem, which is called secondary clustering.

- There are quadratic probing methods that use different quadratic functions.

Maps

Double Hashing

- Does not cause serious clustering problem.
- Uses two hash functions.
- Probe sequence is determined by

  $A[(h(k) + i \cdot h'(k)) \bmod N]$, for $i = 0, 1, 2, \ldots, N - 1$

- One common secondary hash function h' is:

  $h'(k) = q - (k \bmod q)$, for some prime number $q < N$, N is prime

- Another common h' is:

  $h'(k) = 1 + (k \bmod N')$, where N' is slightly smaller than N, N is prime

Maps
Double Hashing

- Example (of the second *h*')

  $h(k) = k \bmod 13$

  $h'(k) = 1 + (k \bmod 11)$

  $h(k, i) = (h(k) + i * h'(k)) \bmod m$

  - Inserting $k = 14$, $h(k) = 1$, $h'(k) = 4$
  - $h(14) = 1$, occupied
  - $i = 1$: $1 + 4 = 5$, occupied
  - $i = 2$: $1 + 8 = 9$, empty, store 14 here

0	
1	79
2	
3	
4	69
5	98
6	
7	59
8	
9	14
10	
11	37
12	

Maps

Load Factor and Efficiency

- Load factor is defined as $\lambda = n / N$
- A larger value of λ means there is higher probability of collisions.
- So, a smaller λ is better.
- With chaining method, λ could be greater than 1.
- With open addressing, $\lambda \leq 1$.

- Performance of chaining method:
  - A theoretical analysis shows that the average number of slots that need to be probed for a successful search is approximately $1+\frac{\lambda}{2}$.

Maps
Load Factor and Efficiency

- Performance of chaining method (continued):
  – Let *C* be the average number of elements that need to be probed for a successful search.

λ	C
0.5	1.25
0.7	1.35
1.0	1.5
2.0	2

  – Java uses chaining method and *λ* is set to 0.75 or less by default.

Maps

Load Factor and Efficiency

- Performance of double hashing:
  - The average number of slots that need to be probed for a successful search is approximately $\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$
  - Let *C* be the average number of slots that need to be probed for a successful search.

λ	C
0.3	1.19
0.5	1.39
0.7	1.72
0.9	2.56

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.