# CS171 Introduction to Computer Science II

# Priority Queues and Binary Heap

# Review

- Binary Search Trees (BST)
- Balanced search trees
- Hash tables

# ST implementations: summary

| implementation | guarantee | | | average case | | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | N/2 | N | N/2 | no | `equals()` |
| binary search (ordered array) | lg N | N | N | lg N | N/2 | N/2 | yes | `compareTo()` |
| BST | N | N | N | 1.38 lg N | 1.38 lg N | ? | yes | `compareTo()` |
| red-black tree | 2 lg N | 2 lg N | 2 lg N | 1.00 lg N | 1.00 lg N | 1.00 lg N | yes | `compareTo()` |
| separate chaining | lg N * | lg N * | lg N * | 3-5 * | 3-5 * | 3-5 * | no | `equals()` |
| linear probing | lg N * | lg N * | lg N * | 3-5 * | 3-5 * | 3-5 * | no | `equals()` |

* under uniform hashing assumption

# Hashing vs. balanced search trees

**Hashing.**

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

**Balanced search trees.**

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

**Java system includes both.**

- Red-black trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

# Priority Queues

- Need to process/search an item with largest (smallest) key, but not necessarily full sorted order
- Support two operations
  - Remove maximum (or minimum)
  - Insert
- Similar to
  - Stacks (remove newest)
  - Queues (remove oldest)

# Example

| *operation* | *argument* | *return value* |
|---|---|---|
| *insert* | P | |
| *insert* | Q | |
| *insert* | E | |
| *remove max* | | Q |
| *insert* | X | |
| *insert* | A | |
| *insert* | M | |
| *remove max* | | X |
| *insert* | P | |
| *insert* | L | |
| *insert* | E | |
| *remove max* | | P |

# Applications

- Job scheduling
  - Keys corresponds to priorities of the tasks
- Sorting algorithm
  - Heapsort
- Graph algorithms
  - Shortest path
- Statistics
  - Maintain largest M values in a sequence

# Priority queue API

**Requirement.** Generic items are `Comparable`.

| public class MaxPQ<Key extends Comparable<Key>> | | |
|---|---|---|
| | MaxPQ() | *create a priority queue* |
| | MaxPQ(maxN) | *create a priority queue of initial capacity maxN* |
| void | insert(Key v) | *insert a key into the priority queue* |
| Key | max() | *return the largest key* |
| Key | delMax() | *return and remove the largest key* |
| boolean | isEmpty() | *is the priority queue empty?* |
| int | size() | *number of entries in the priority queue* |

**API for a generic priority queue**

## Priority queue client example

**Challenge.** Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

**Constraint.** Not enough memory to store $N$ items.

```
% more tinyBatch.txt
Turing        6/17/1990   644.08
vonNeumann    3/26/2002  4121.85
Dijkstra      8/22/2007  2678.40
vonNeumann    1/11/1999  4409.74
Dijkstra     11/18/1995   837.42
Hoare         5/10/1993  3229.27
vonNeumann    2/12/1994  4732.35
Hoare         8/18/1992  4381.21
Turing        1/11/2002    66.10
Thompson      2/27/2000  4747.08
Turing        2/11/1991  2156.86
Hoare         8/12/2003  1025.70
vonNeumann   10/13/1993  2520.97
Dijkstra      9/10/2000   708.95
Turing       10/12/1993  3532.36
Hoare         2/10/2005  4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson      2/27/2000  4747.08
vonNeumann    2/12/1994  4732.35
vonNeumann    1/11/1999  4409.74
Hoare         8/18/1992  4381.21
vonNeumann    3/26/2002  4121.85
```

sort key

# Possible implementations

- Sorting N items
  - Time: NlogN
  - Space: N
- Elementary PQ - Compare each new key against M largest seen so far
  - Time: NM
  - Space: M
- Using an efficient MaxPQ Implementation

# Priority queue client example

**Challenge.** Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();

while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   Transaction item = new Transaction(line);
   pq.insert(item);
   if (pq.size() > M)
      pq.delMin();
}
```

use a min-oriented pq

Transaction data type is Comparable

pq contains largest M items

**order of growth of finding the largest M in a stream of N items**

| implementation | time | space |
|---|---|---|
| sort | N log N | N |
| elementary PQ | M N | M |
| binary heap | N log M | M |
| best in theory | N | M |

# Implementations

- Elementary representations
  - Unordered array (lazy approach)
  - ordered array (eager approach)
- Efficient implementation
  - Binary heap structure

- Can we implement priority queue using Binary Search Trees?

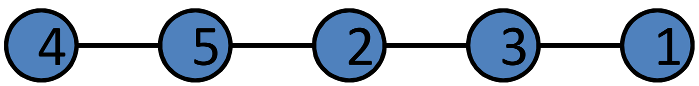
# Priority queue: unordered and ordered array implementation

| *operation* | *argument* | *return value* | *size* | *contents (unordered)* | | | | | | | *contents (ordered)* | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *insert* | P | | 1 | P | | | | | | | P | | | | | | |
| *insert* | Q | | 2 | P | Q | | | | | | P | Q | | | | | |
| *insert* | E | | 3 | P | Q | E | | | | | E | P | Q | | | | |
| *remove max* | | Q | 2 | P | E | | | | | | E | P | | | | | |
| *insert* | X | | 3 | P | E | X | | | | | E | P | X | | | | |
| *insert* | A | | 4 | P | E | X | A | | | | A | E | P | X | | | |
| *insert* | M | | 5 | P | E | X | A | M | | | A | E | M | P | X | | |
| *remove max* | | X | 4 | P | E | M | A | | | | A | E | M | P | | | |
| *insert* | P | | 5 | P | E | M | A | P | | | A | E | M | P | P | | |
| *insert* | L | | 6 | P | E | M | A | P | L | | A | E | L | M | P | P | |
| *insert* | E | | 7 | P | E | M | A | P | L | E | A | E | E | L | M | P | P |
| *remove max* | | P | 6 | E | M | A | P | L | E | | A | E | E | L | M | P | |

**A sequence of operations on a priority queue**

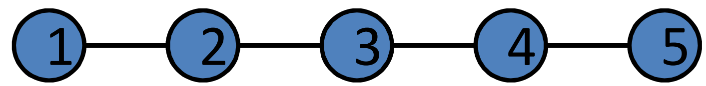
# Sequence-based Priority Queue

- 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

# Priority queue: unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;    // pq[i] = ith element on pq
   private int N;       // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

   public boolean isEmpty()
   {  return N == 0; }

   public void insert(Key x)
   {  pq[N++] = x;  }

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic array creation

`less()` and `exch()` as for sorting

# Priority queue elementary implementations

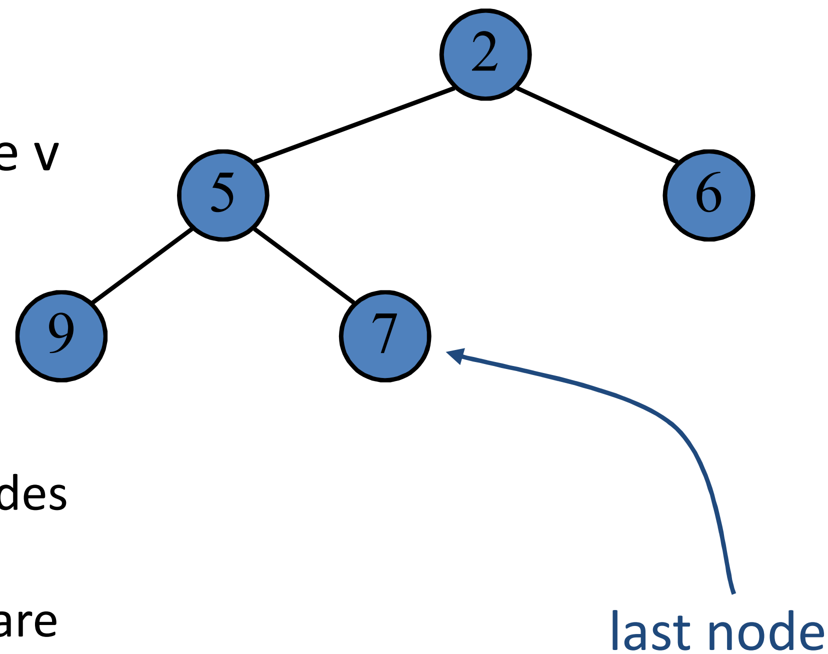
Challenge. Implement all operations efficiently.

**order-of-growth of running time for priority queue with N items**

| implementation | insert | del max | max |
|---|---|---|---|
| unordered array | 1 | N | N |
| ordered array | N | 1 | 1 |
| goal | log N | log N | log N |

# Binary Heap Tree

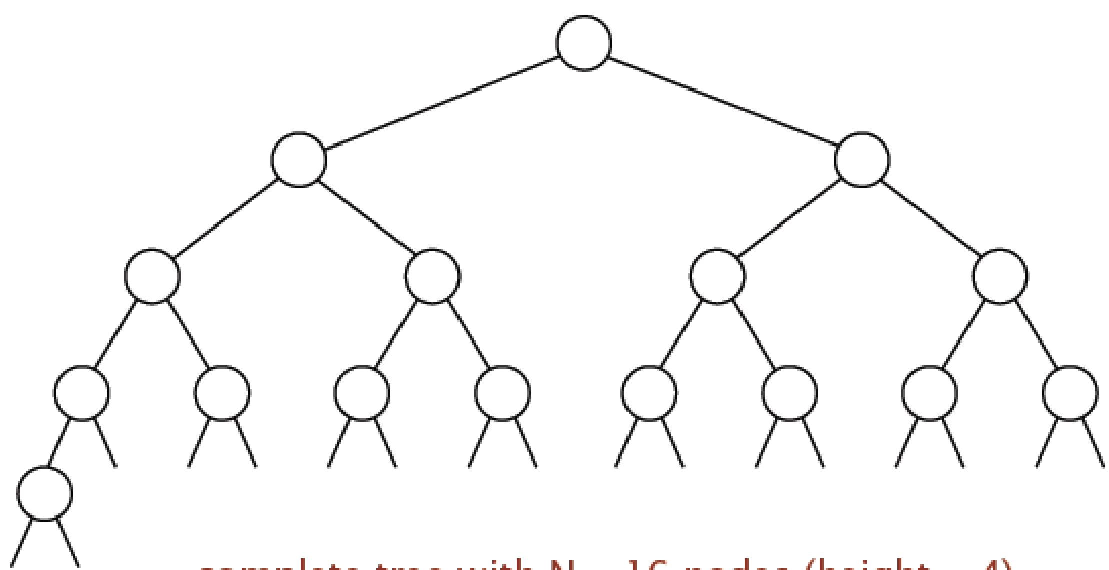
- A heap is a binary tree storing keys at its nodes and satisfying two properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let $h$ be the height of the heap
    - for $i = 0, \ldots, h-1$, there are $2^i$ nodes of depth $i$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



# Binary tree

**Binary tree.** Empty or node with links to left and right binary trees.

**Complete tree.** Perfectly balanced, except for bottom level.



complete tree with N = 16 nodes (height = 4)

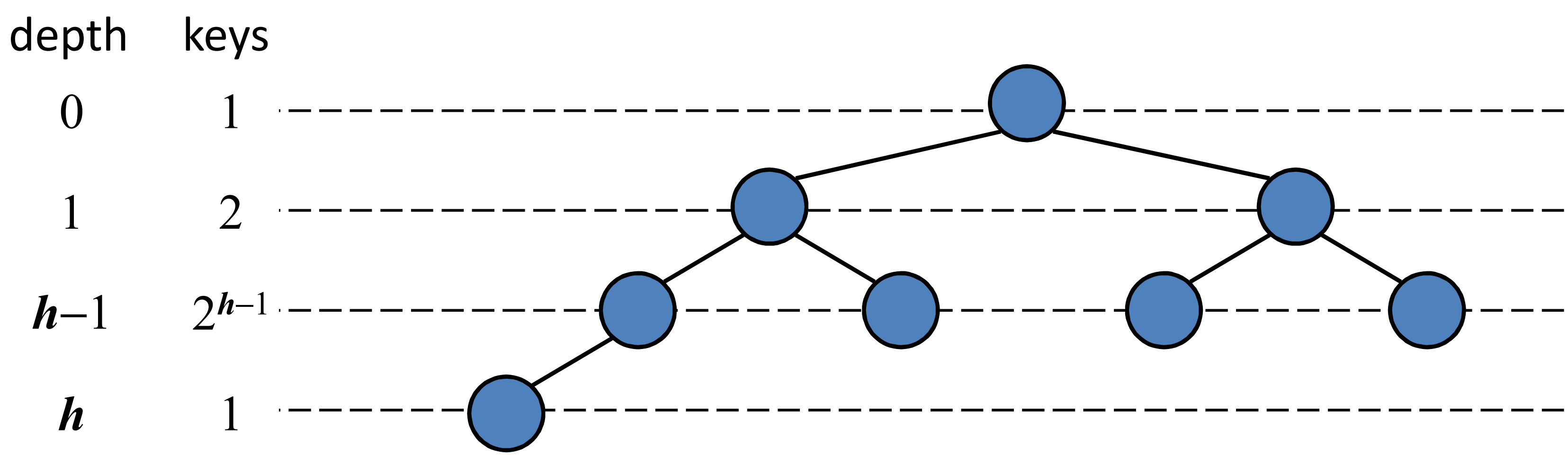
**Property.** Height of complete tree with $N$ nodes is $\lfloor \lg N \rfloor$.

**Pf.** Height only increases when $N$ is a power of 2.

# Height of a Heap

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

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



# A complete binary tree in nature

# Binary heap representations

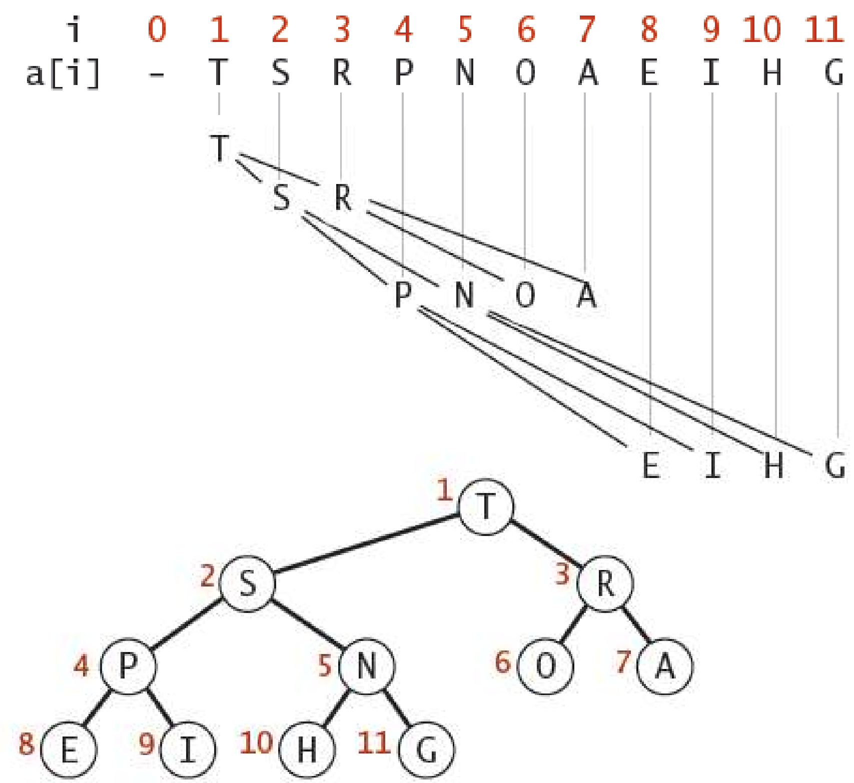
Binary heap. Array representation of a heap-ordered complete binary tree.

## Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

## Array representation.

- Take nodes in level order.
- No explicit links needed!



Heap representations

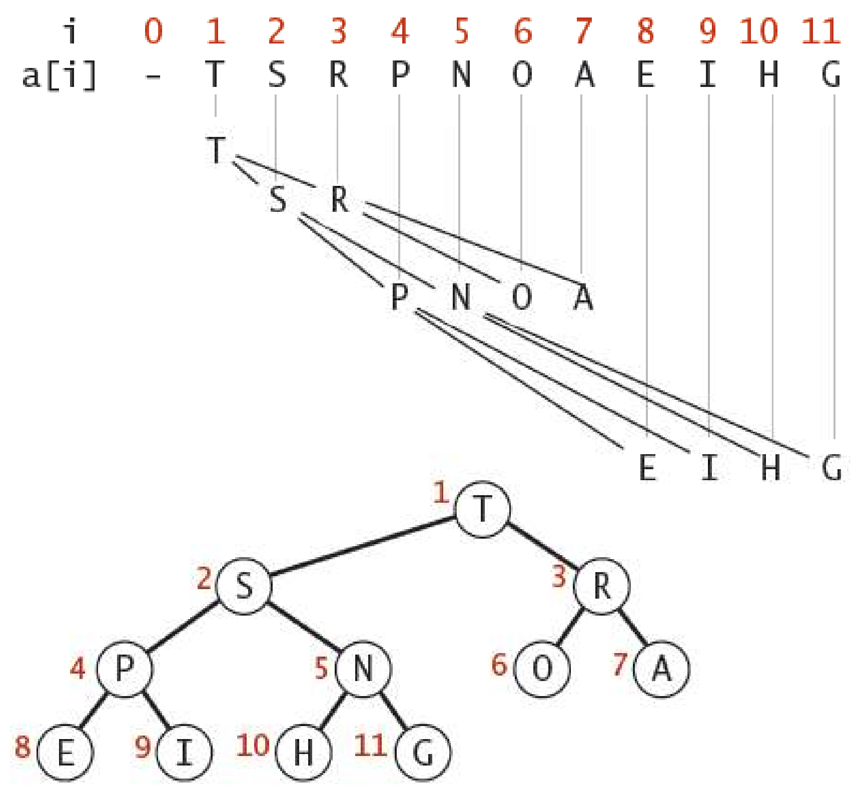
# Binary heap properties

Proposition. Largest key is `a[1]`, which is root of binary tree.

indices start at 1

Proposition. Can use array indices to move through tree.

- Parent of node at `k` is at `k/2`.
- Children of node at `k` are at `2k` and `2k+1`.




Heap representations

# Insert/Remove and Maintaining Heap order

- When a node’s key is larger than its parent key
  - Upheap (promote, swim)
- When a node’s key becomes smaller than its children’s keys
  - Downheap (demote, sink)

# Promotion in a heap

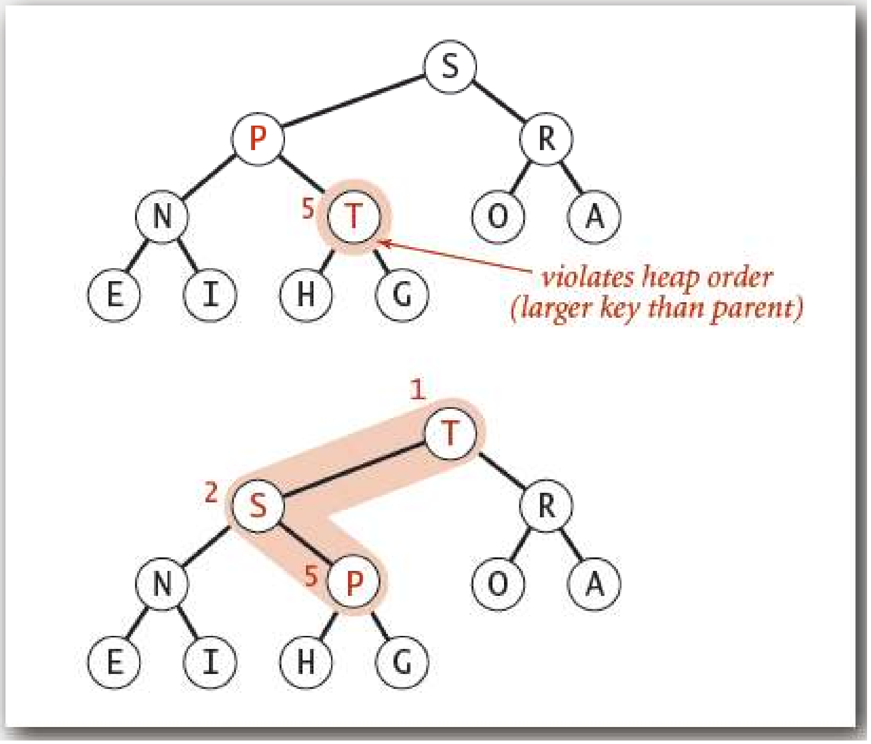
Scenario. Node's key becomes larger key than its parent's key.

To eliminate the violation:

- Exchange key in node with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2

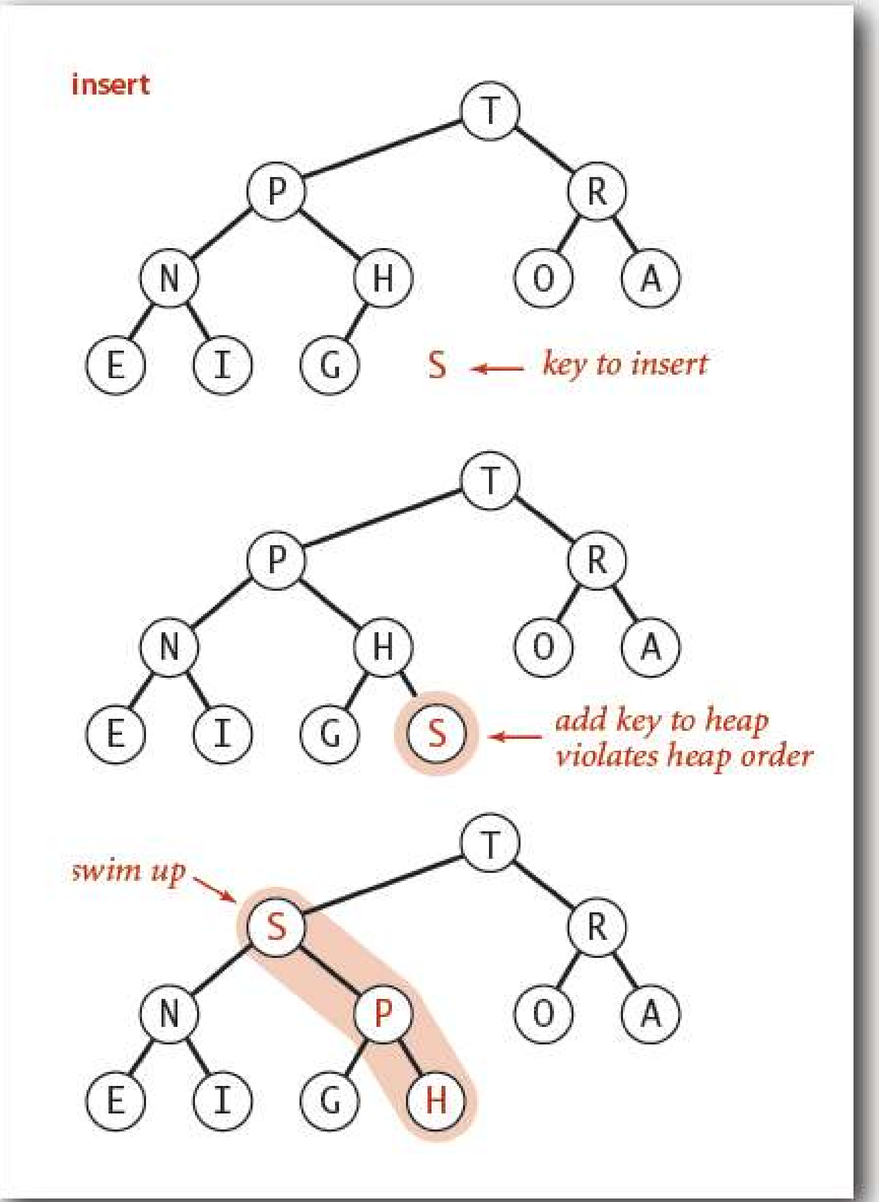


# Insertion in a heap

**Insert.** Add node at end, then swim it up.

**Cost.** At most $1 + \lg N$ compares.

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```



# Demotion in a heap

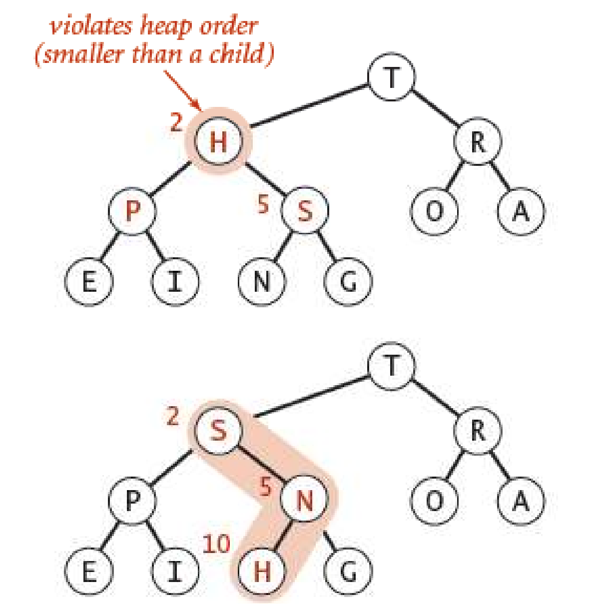
**Scenario.** Node's key becomes smaller than one (or both) of its children's keys.

**To eliminate the violation:**

- Exchange key in node with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node at k are 2k and 2k+1



Top-down reheapify (sink)

**Power struggle.** Better subordinate promoted.

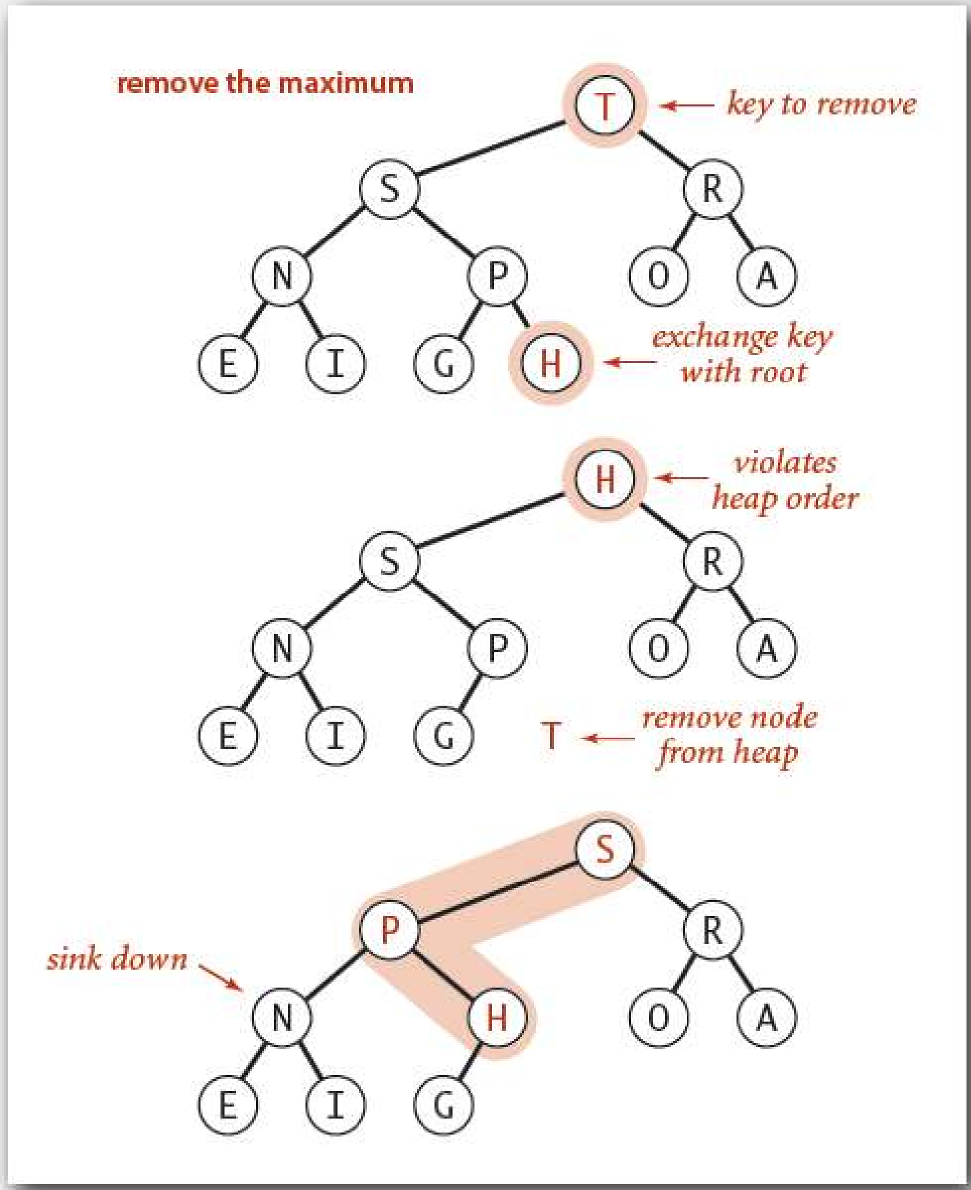
# Delete the maximum in a heap

**Delete max.** Exchange root with node at end, then sink it down.

**Cost.** At most $2 \lg N$ compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

prevent loitering



# Demo

# Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;
   private int N;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {   return N == 0;   }
   public void insert(Key key)
   {   /* see previous code */   }
   public Key delMax()
   {   /* see previous code */   }

   private void swim(int k)
   {   /* see previous code */   }
   private void sink(int k)
   {   /* see previous code */   }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j] < 0;   }
   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;   }
}
```

PQ ops

heap helper functions

array helper functions

# Priority queues implementation cost summary

**order-of-growth of running time for priority queue with N items**

| implementation | insert | del max | max |
|---|---|---|---|
| unordered array | 1 | N | N |
| ordered array | N | 1 | 1 |
| binary heap | log N | log N | 1 |
| d-ary heap | $\log_d N$ | $d \log_d N$ | 1 |
| Fibonacci | 1 | log N † | 1 |
| impossible | 1 | 1 | 1 |

† amortized