heap

- *complete binary tree (review)*
- *heap and priority queues (Chapter 9)*
- *binary heap and min-heap*
- *max-heap demo*
- *max-heap coding*
- *heap sort (Chapter 7)*

heap coding: heap.h

Heap ADT: A **one - based** and **one dimensional array** is used to simplify parent and child calculations.

```
struct Heap {
  int *nodes;          // an array of nodes
  int capacity;        // array size of node or key, item
  int N;               // the number of nodes in the heap
  bool (*comp)(Heap*, int, int);
  Heap(int capa = 2) {
    capacity = capa;
    nodes = new int[capacity];
    N = 0;
    comp = nullptr;
  };
  ~Heap() {};
};
using heap = Heap*;
```

heap coding: heap.h

```
void clear(heap hp);                    // deallocate heap
int size(heap hp);                      // return nodes in heap currently
int level(int n);                       // return level based on num of nodes
int capacity(heap hp);                   // return its capacity (array size)
int reserve(heap hp, int capa);         // reserve the array size (= capacity)
int full(heap hp);                      // return true/false
int empty(heap hp);                     // return true/false
void grow(heap hp, int key);            // add a new key
void trim(heap hp);                     // delete a queue
int heapify(heap hp);                   // convert a complete BT into a heap

// helper functions to support grow/trim functions
int less(heap hp, int i, int j);        // used in max heap
int more(heap hp, int i, int j);        // used in min heap
void swim(heap hp, int k);              // bubble up
void sink(heap hp, int k);              // tickle down
// helper functions to check heap invariant
int heapOrdered(heap hp);               // is heap[1..N] a heap?
```

Promotion in a heap: swim

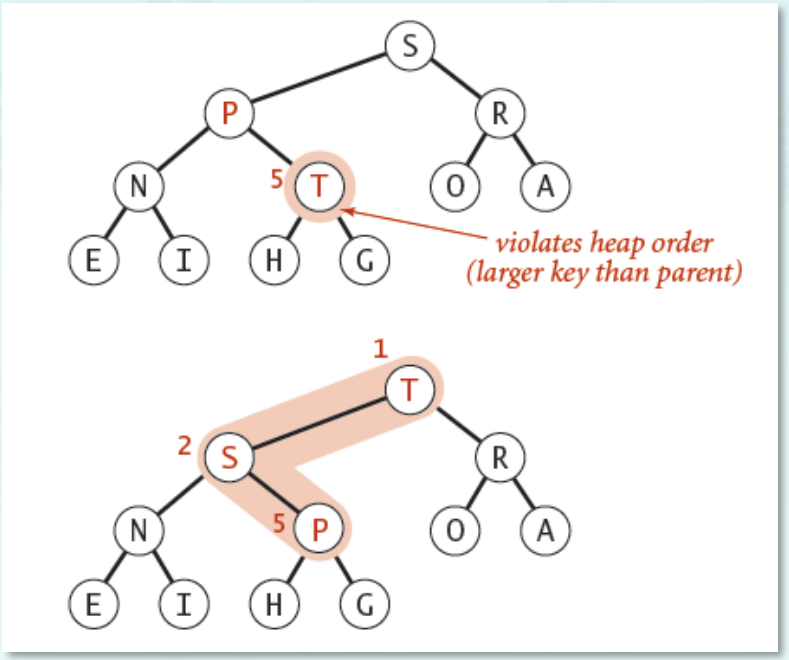
- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until **heap order** restored.

swim up

or

sink down

This is a maxheap example.

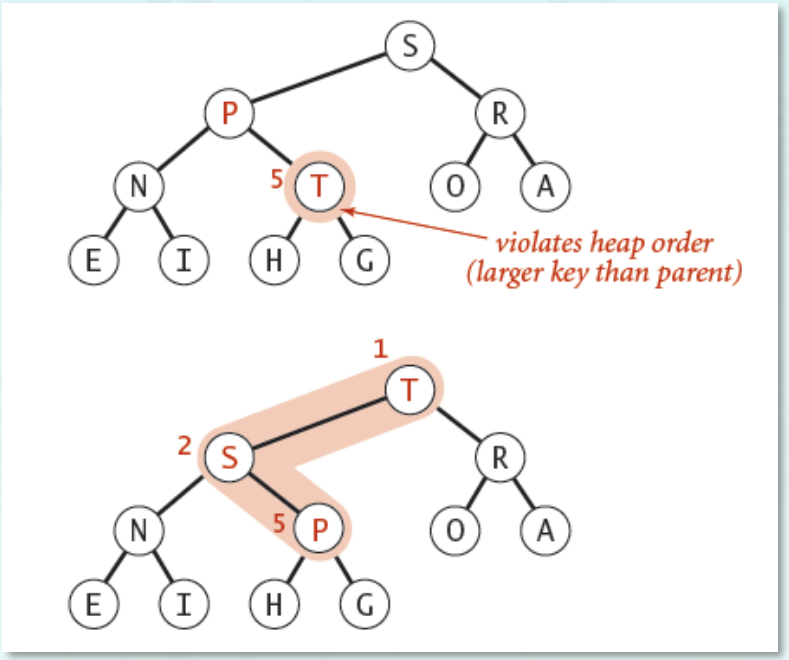


Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

guess a name?

```
bool ____(heap h, int p, int c) {
    return h->nodes[p] < h->nodes[c];
}
```



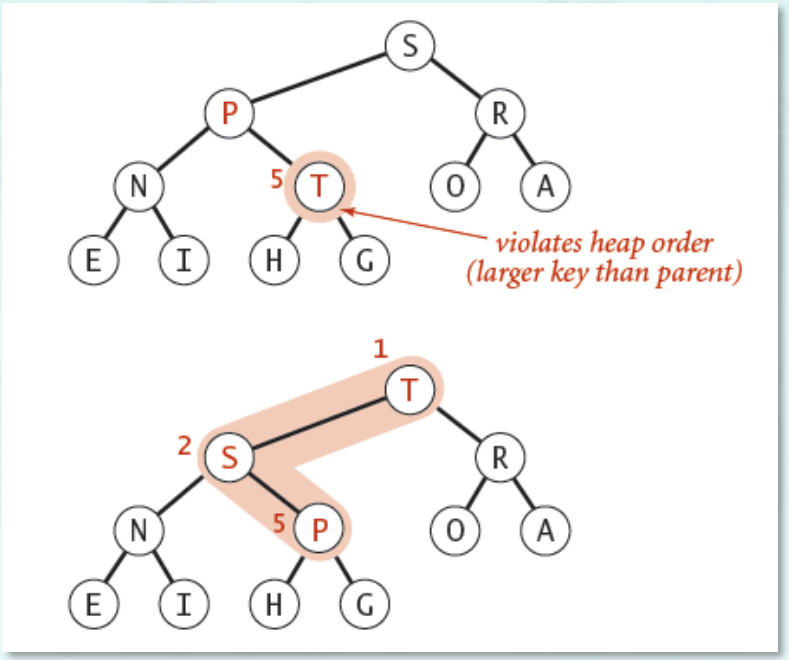
Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

guess a name?

```
bool ____(heap h, int p, int c) {
    return h->nodes[p] < h->nodes[c];
}
```

```
void ____(heap h, int p, int c) {
    Key item = h->nodes[p];
    h->nodes[p] = h->nodes[c];
    h->nodes[c] = item;
}
```

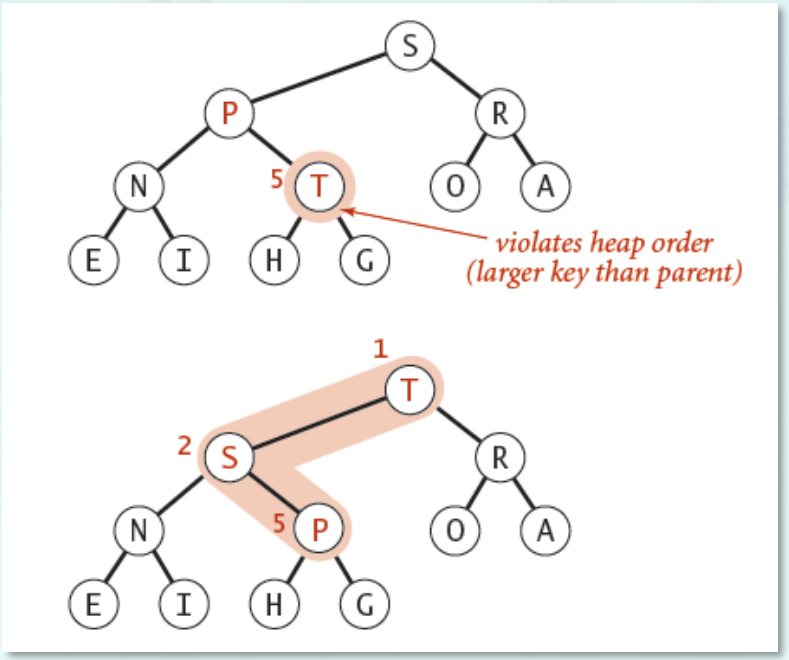


Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

```
bool less(heap h, int p, int c) {
    return h->nodes[p] < h->nodes[c];
}
```

```
void swap(heap h, int p, int c) {
    Key item = h->nodes[p];
    h->nodes[p] = h->nodes[c];
    h->nodes[c] = item;
}
```

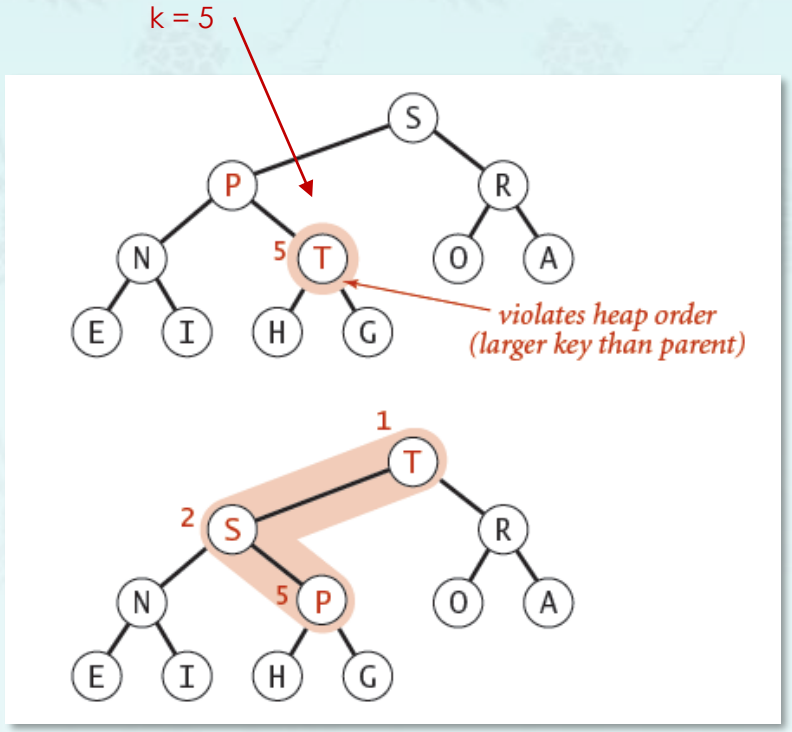


Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

guess a name?

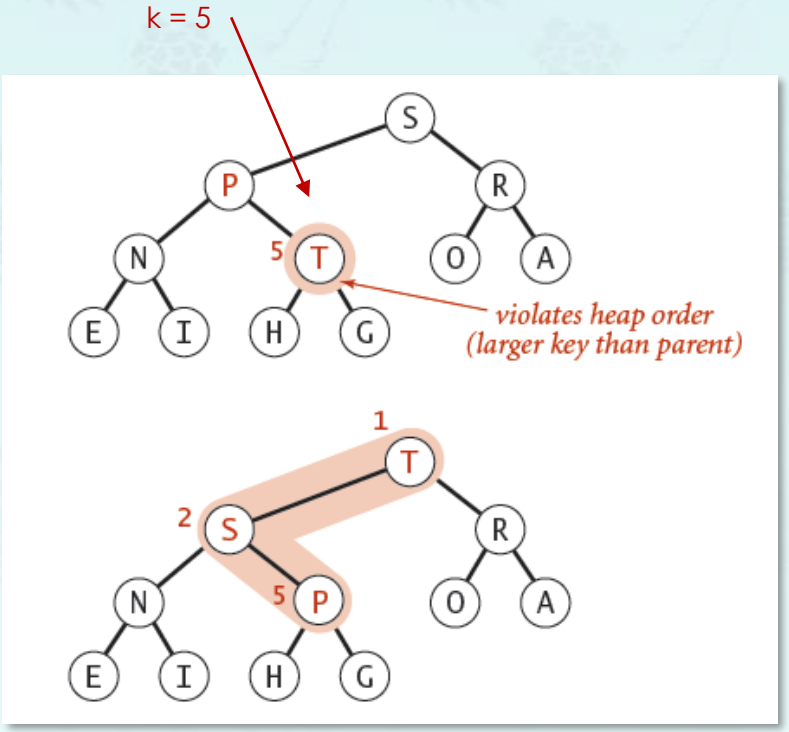
```
void      (heap h, int k)
{
  while (not reached at root &&
          k's parent key < k's key)
  {
                                    
  }
}
```



Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

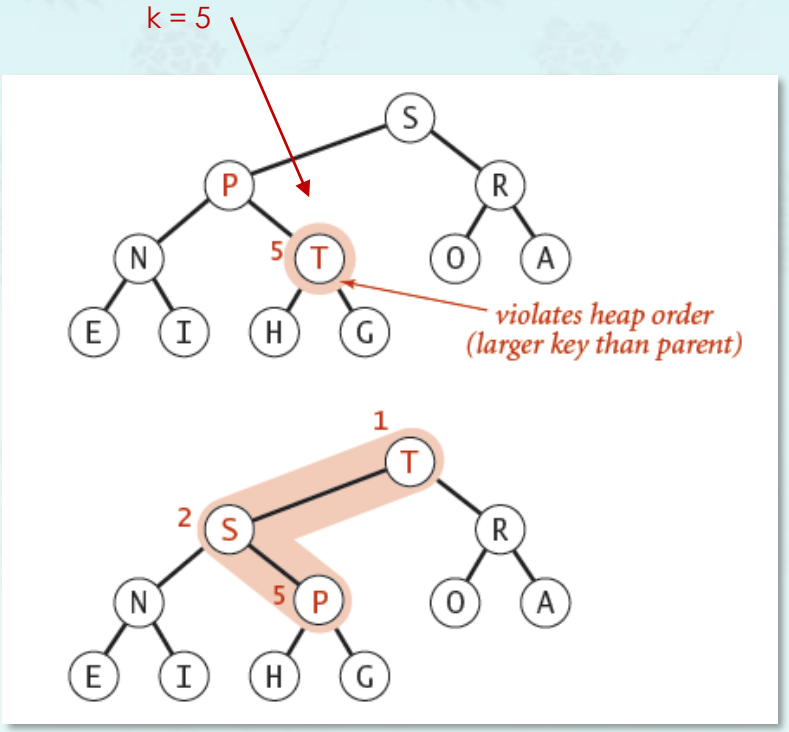
```
void swim(heap h, int k)
{
  while (not reached at root &&
          k's parent key < k's key)
  {
          [                    ]

  }
}
```



Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

```
void swim(heap h, int k)
{
  while (not reached at root &&
          k's parent key < k's key)
  {
        swap k and its parent
        go for the next
  }
}
```



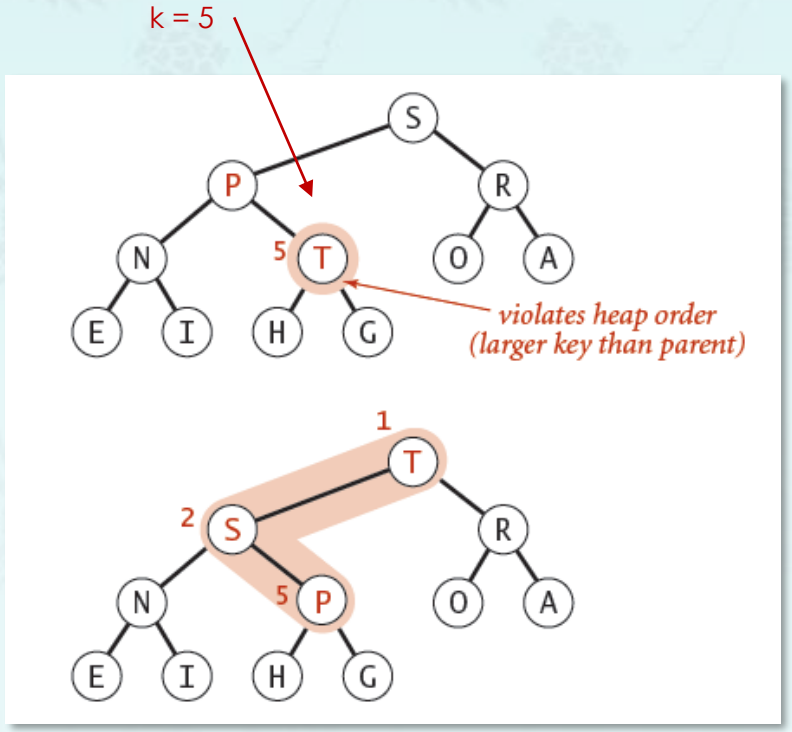
Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

not reached at root

parent of k

```
void swim(h ap h, int k)
{
  while (
  {


  }
}
```



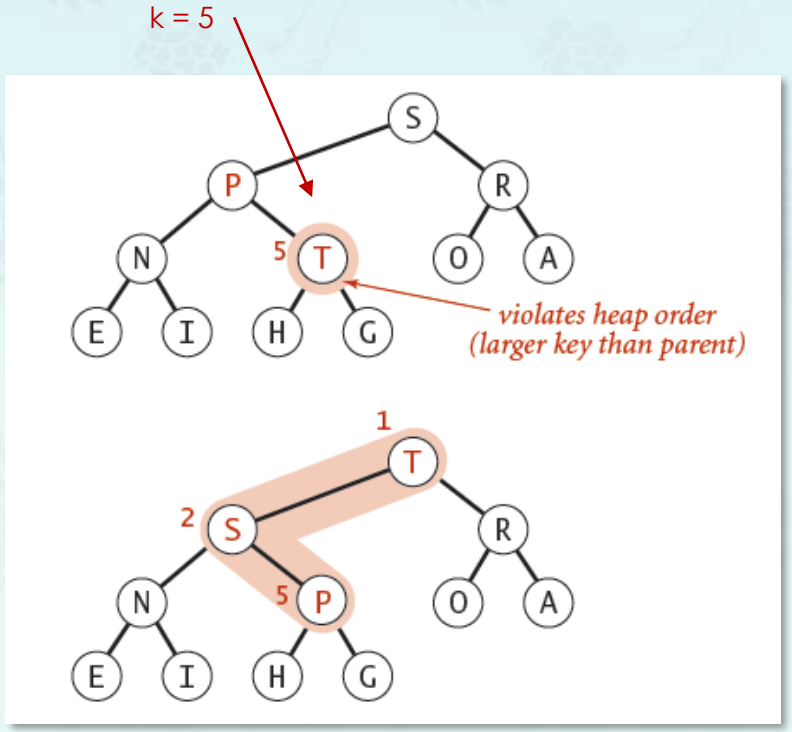
Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

not reached at root

parent(k/2) is less its child(k)

```
void swim(h ap h, int k)
{
  while (k > 1 &&
  {


  }
}
```



Promotion in a heap: swim

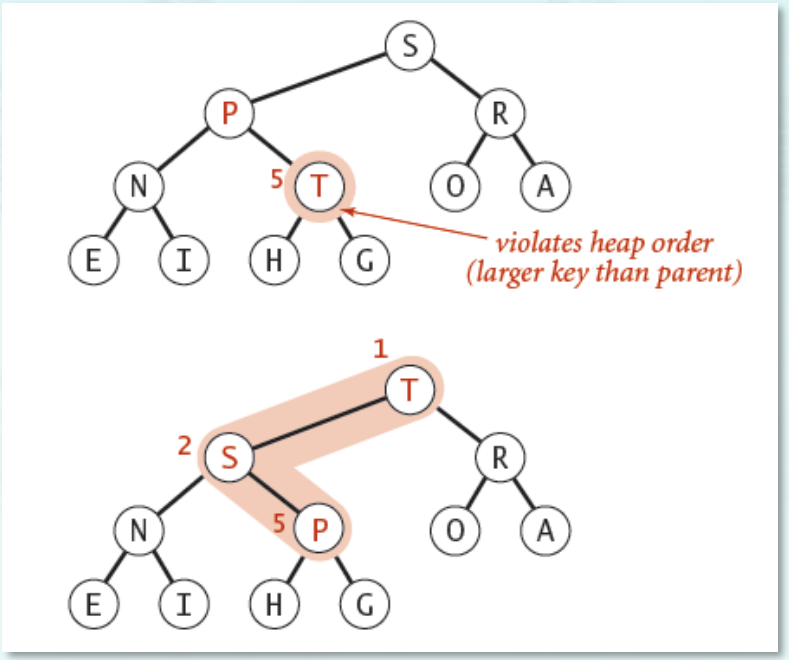
- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

not reached at root

parent(k/2) is less its child(k)

```
void swim(h ap h, int k)
{
  while (k > 1 && less(h, k / 2, k))
  {
    
  }
}
```

swap parent(k/2) and its child(k)



Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

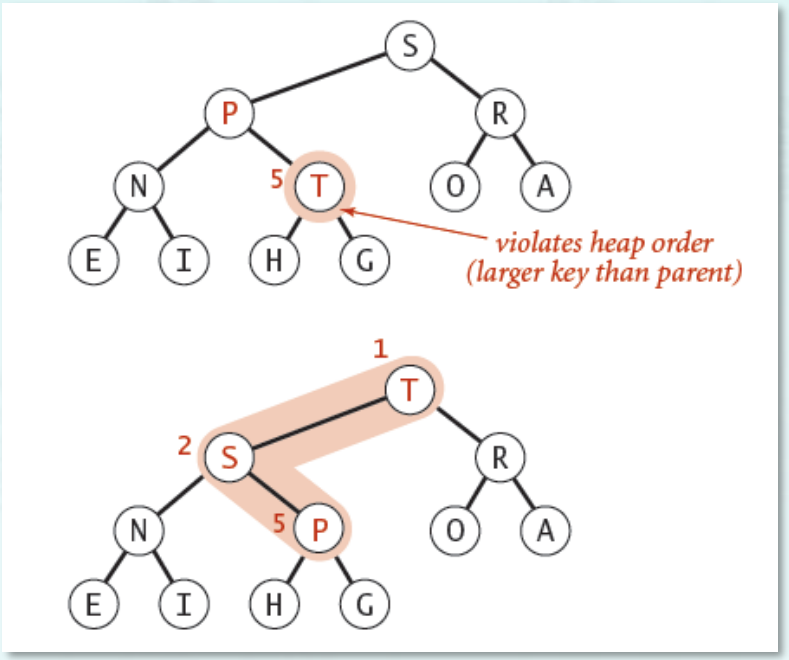
```
void swim(h  ap h, int k)
{
  while (k > 1 && less(h, k / 2, k))
  {
    swap(h, k / 2, k);

  }
}
```

not reached at root

parent(k/2) is less its child(k)

swap parent(k/2) and its child(k)

move up one level



나중에 comp써서 less, more자유롭게 쓸수있게 할꺼

Promotion in a heap: swim

- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

```
void swim(h ap h, int k)
{
  while (k > 1 && less(h, k / 2, k))
  {
    swap(h, k / 2, k);
    k = k / 2;
  }
}
```

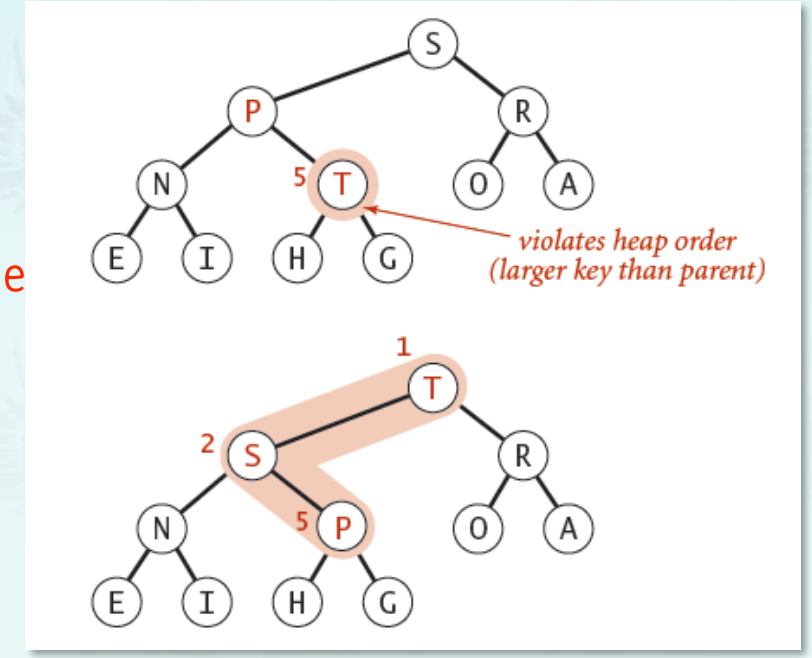
not reached at root

parent(k/2) is less its child(k)

root, parent node, curr node

swap parent(k/2) and its child(k)

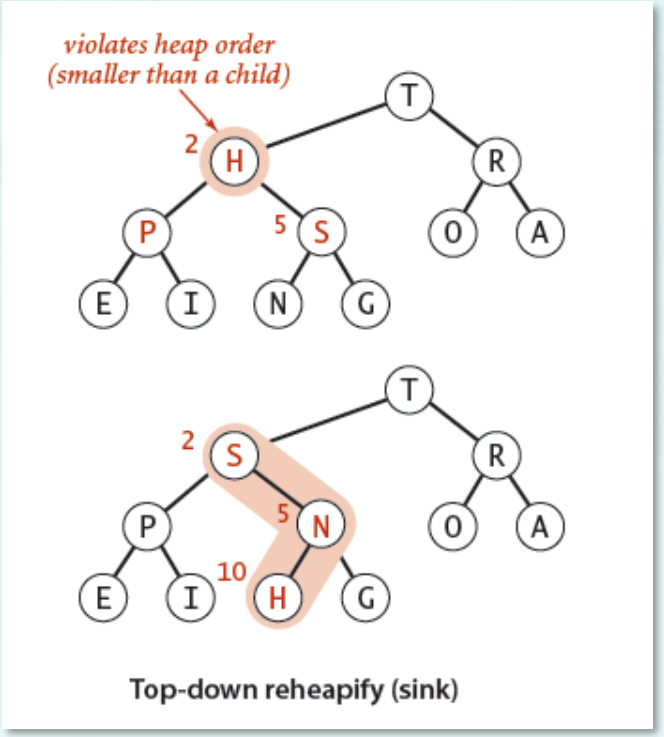
move up one level



swim up

or

sink down

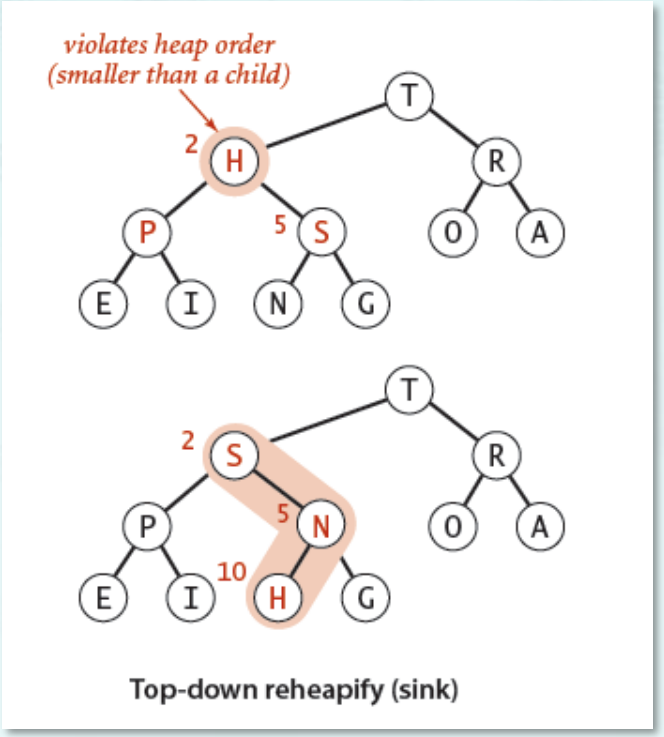


Top-down reheapify (sink)

Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.

Why not smaller child?



Top-down reheapify (sink)

This is a maxheap example.

두개의 child를 비교해서 큰거를 자신이랑 비교하기!

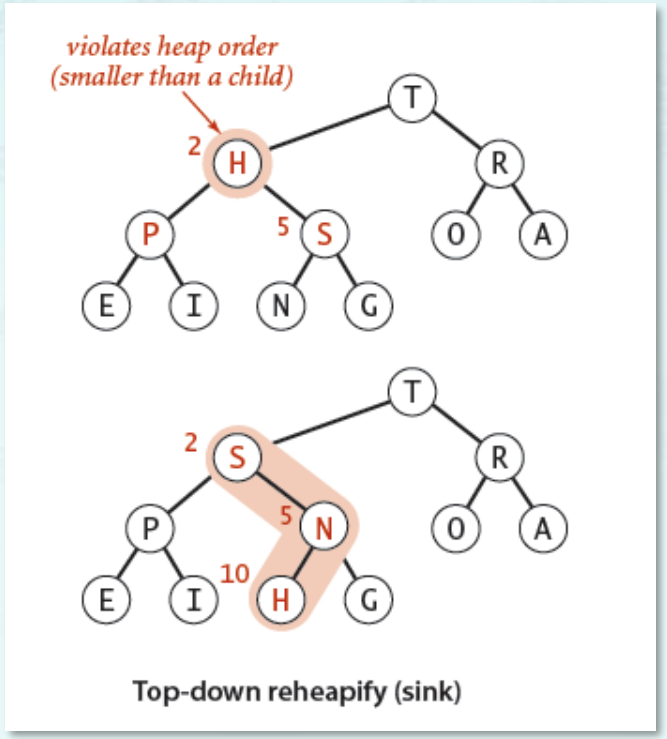
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (k's child not reached the last)
   {
      find the larger child of k, let it be j. (j = 5)

      if k's key is not less than j's key, break;
      swap k and j since k's key > j's key
      set k to the next node wh
   }
}
```

k = 2



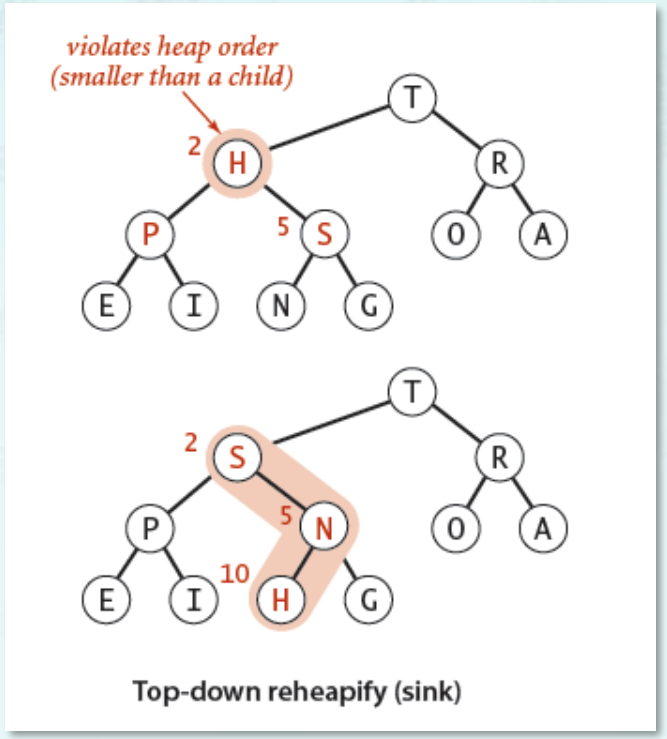
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (k's child not reached the last)
   {
      find the larger child of k, let it be j. (j = 5)

      if k's key is not less than j's key, break;
      swap k and j since k's key > j's key
      set k to the next node which is j.
   }
}
```

k = 2



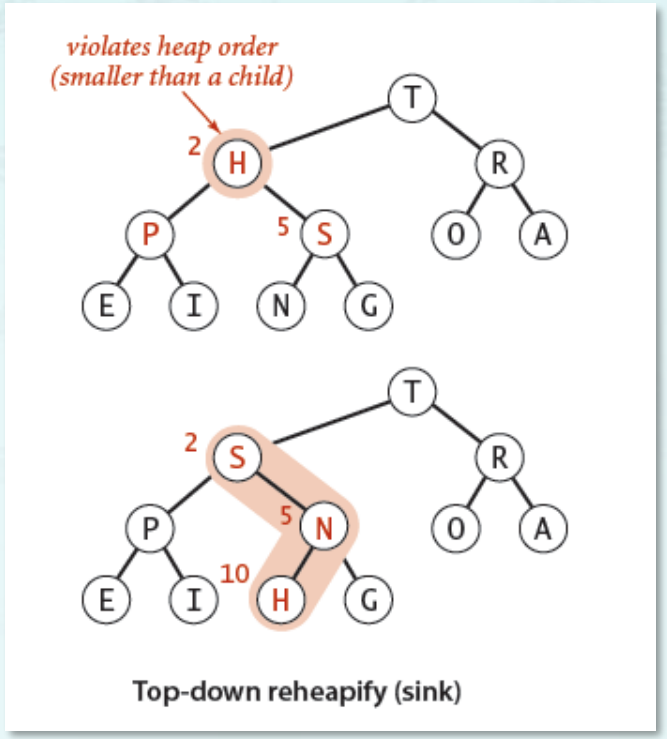
Top-down reheapify (sink)

Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (k's child not reached the last)
    {
    find the larger child of k, let it be j. (j = 5)

   }
}
```



Top-down reheapify (sink)

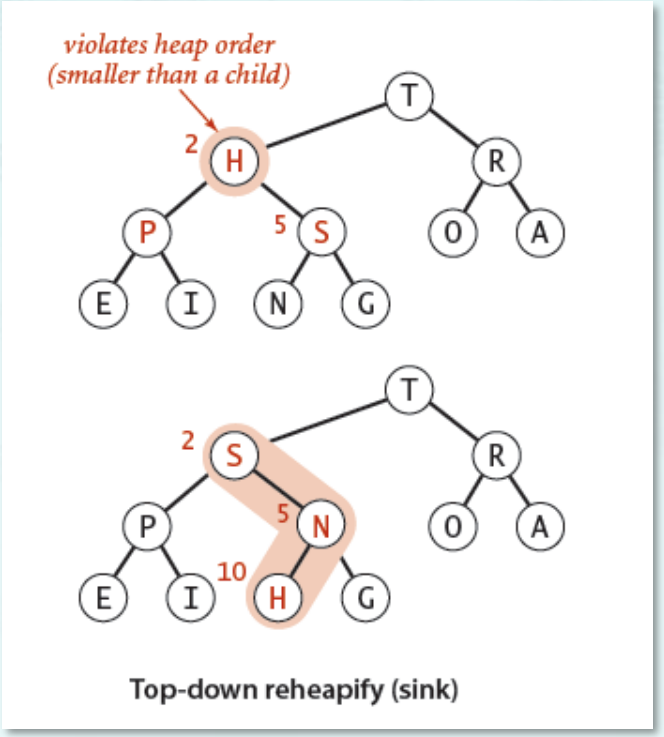
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      find the larger child of k, let it be j. (j = 5)

   }
}
```

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



Top-down reheapify (sink)

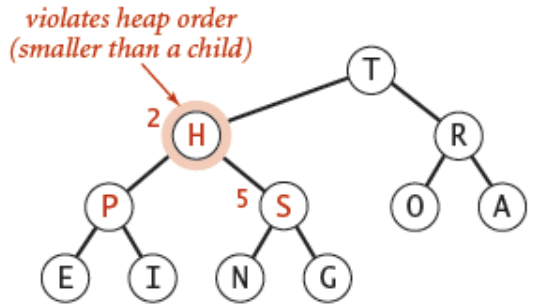
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

   find the larger child of k, let it be j. (j = 5)

   }
}
```

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



Top-down reheapify (sink)

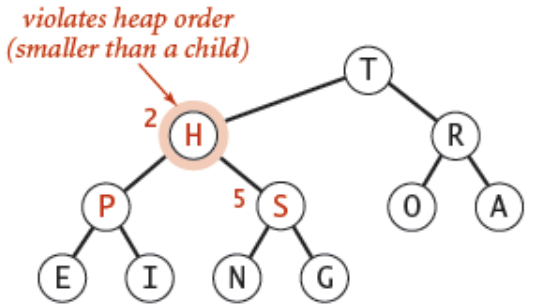
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

      if (j < h->N && less(h, j, j + 1)) j++;

   }
}
```

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



Top-down reheapify (sink)

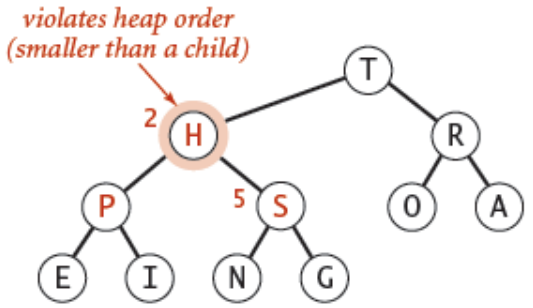
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

      if (j < h->N && less(h, j, j + 1)) j++;

      if k's key is not less than j's key, break;
      swap k and j since k's key > j's key
      set k to the next node (which is j.)
   }
}
```

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



Top-down reheapify (sink)

Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

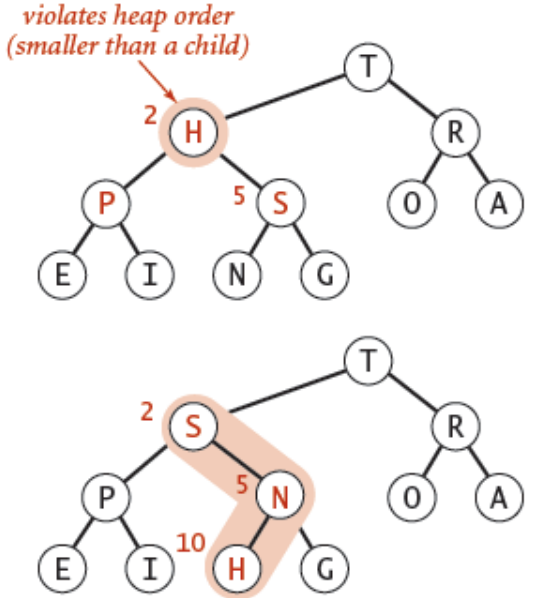
Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
```

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

swap k and j since k's key > j's key
set k to the next node (which is j.)

```
}
```



Top-down reheapify (sink)

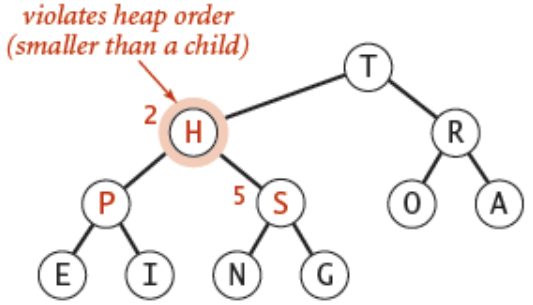
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      set k to the next node (which is j.)
   }
}
```

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



Top-down reheapify (sink)

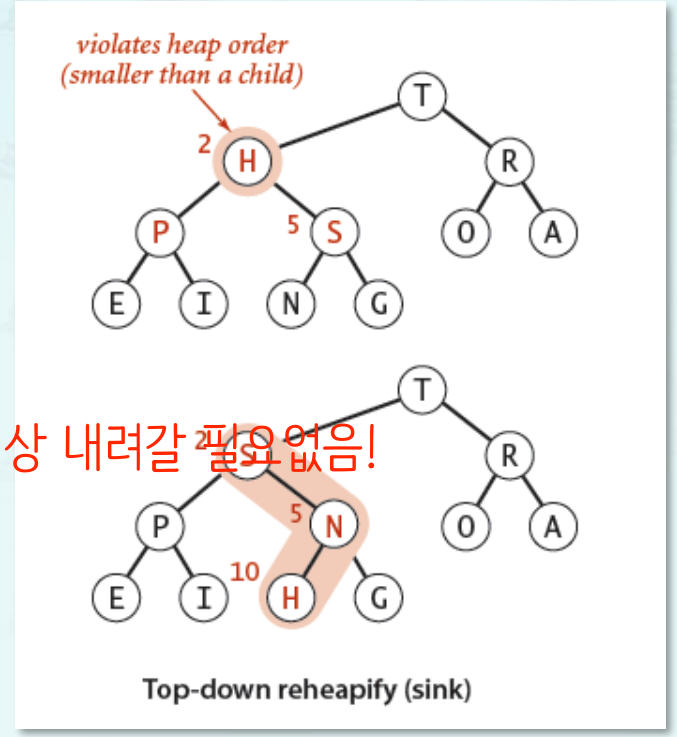
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

Why not smaller child?

```
void sink(heap h, int k)
{
   while (2 * k <= h->N)
   {
      int j = 2 * k;

      if (j < h->N && less(h, j, j + 1)) j++;
      if (!less(h, k, j)) break;
      swap(h, k, j);
      k = j;
   }
}
```

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



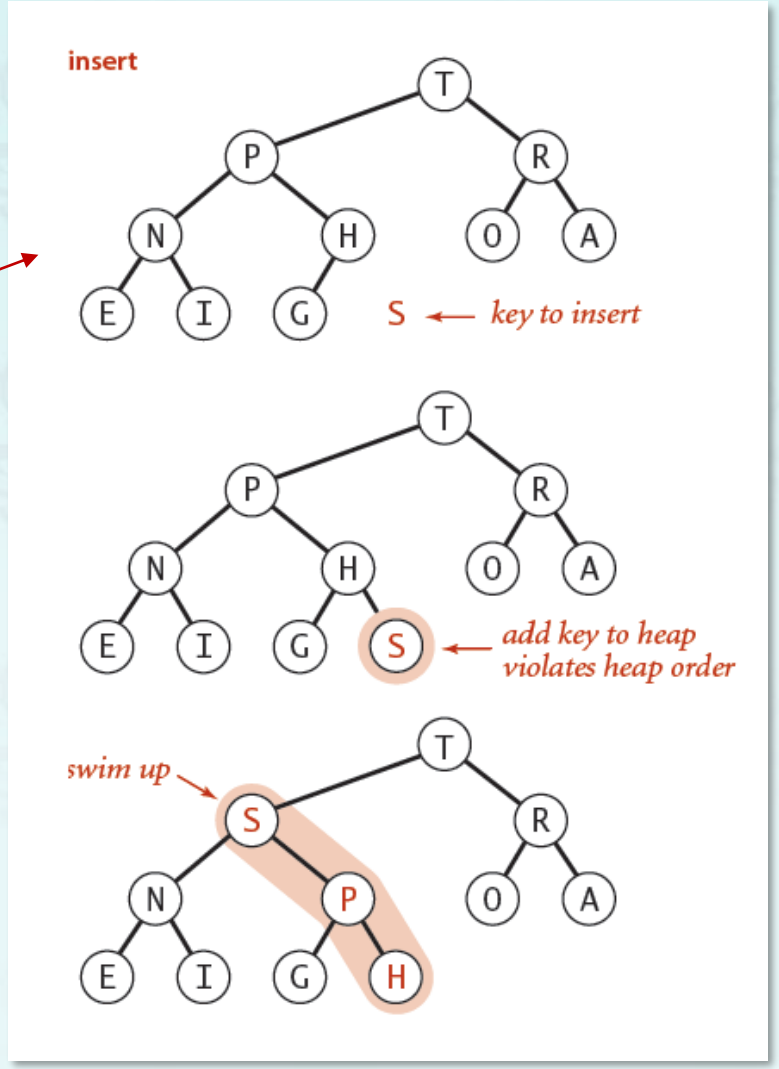
Top-down reheapify (sink)

heap coding

- Insert: Add node at end, then **swim** it up.
  - Cost: At most 1 + log N compares.

What is N now?

Insert



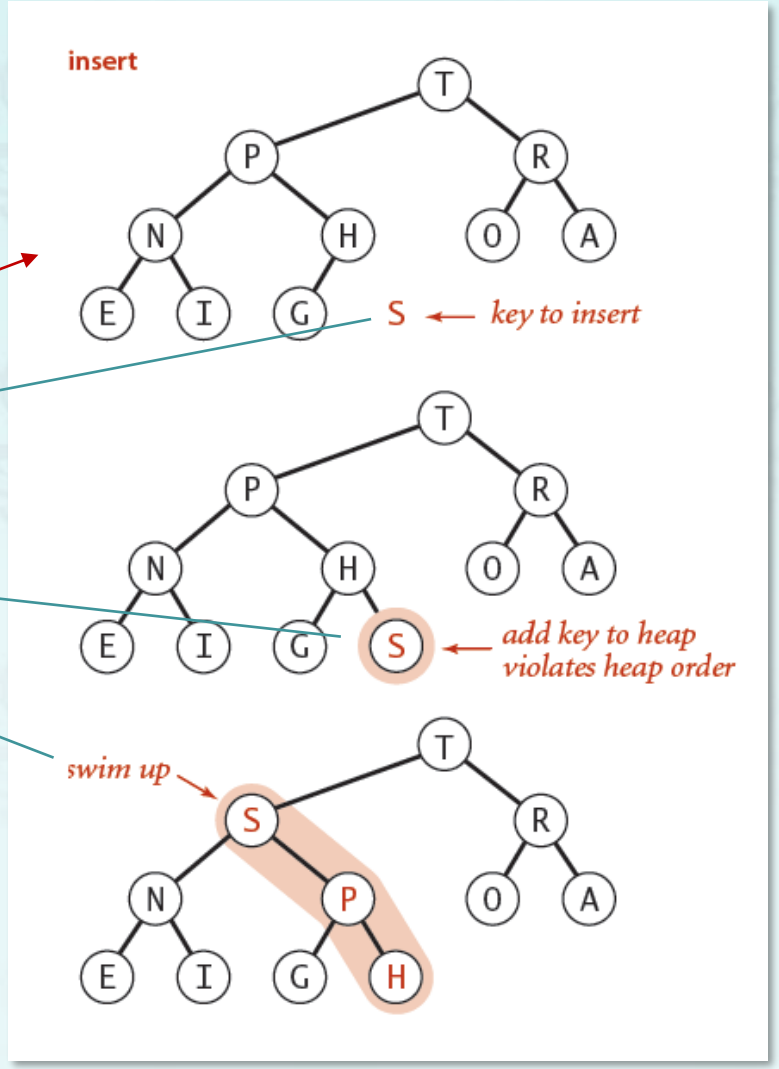
heap coding

Insertion in a heap:

- Insert: Add node at end, then **swim** it up.
  - Cost: At most 1 + log N compares.

What is N now?

```
void insert(heap h, Key key)
{
   
}
```



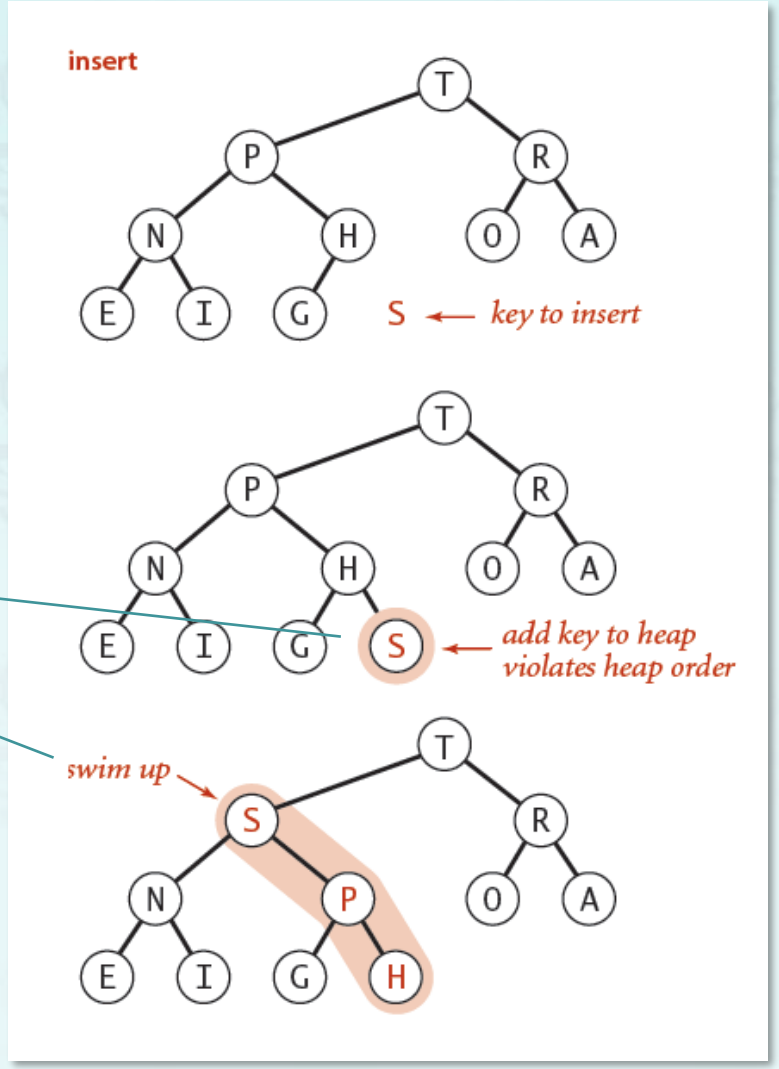

```
typedef struct heap {
    Key   *nodes;    // an array of node
    int   capacity; // array size of node
    int   N;        // the number of nodes
} heap;
```

heap coding

Insertion in a heap:

- Insert: Add node at end, then **swim** it up.
  - Cost: At most 1 + log N compares.

```
void insert(heap h, Key key)
{
   h->nodes[++h->N] = key;
   
}
```




```
typedef struct heap {
    Key   *nodes;   // an array of node
    int   capacity; // array size of node
    int   N;        // the number of nodes
} heap;
```

```
void swim(heap h, int k)
void sink(heap h, int k)
```

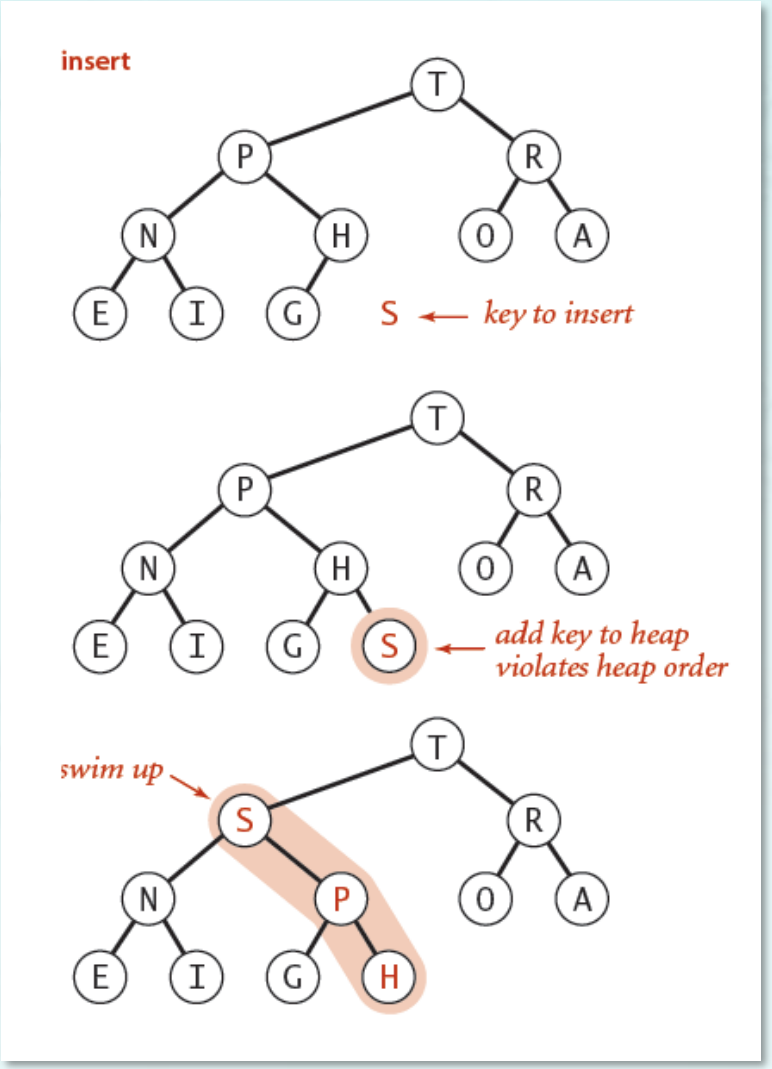
heap coding

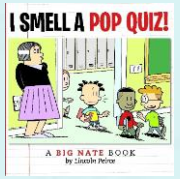
Insertion in a heap:

- Insert: Add node at end, then **swim** it up.
  - Cost: At most 1 + log N compares.

```
void insert(heap h, Key key)
{
   h->nodes[++h->N] = key;
   swim(h, h->N);
}
```

```
typedef struct heap {
    Key   *nodes;   // an array of node
    int   capacity; // array size of node
    int   N;        // the number of nodes
} heap;
```

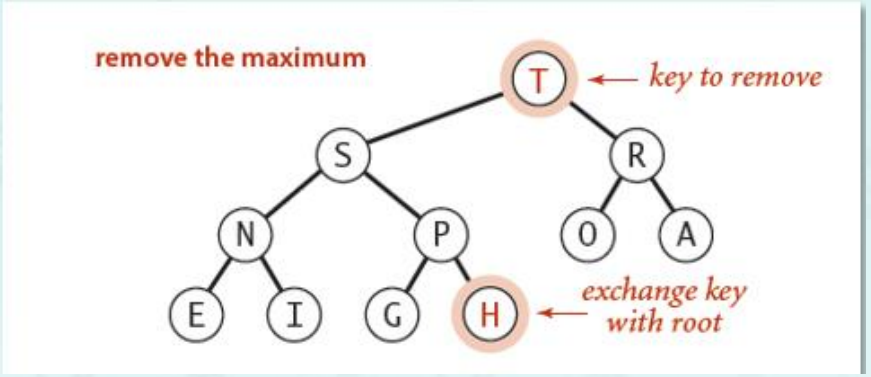




heap coding

(1) Delete the root (max or min) in a heap:

(2) How many times do comparisons occur for n nodes ? (select one):
n, 2n, n^2, 2 log n, n log n, log n




```
void delete(heap h) {


}
```

← **2 or 3 lines of code**

```
void swim(heap h, int k)
void sink(heap h, int k)
```

```
bool less(heap h, int p, int c)
```

```
void swap(heap h, int p, int c)
```

```
typedef   int    Key;
typedef struct struct_heap *heap;
typedef struct struct_heap {
    Key   *nodes;
    int   capacity;
    int   N;
} struct_heap;
```

heap coding

Delete the root (max or min) in a heap:

- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

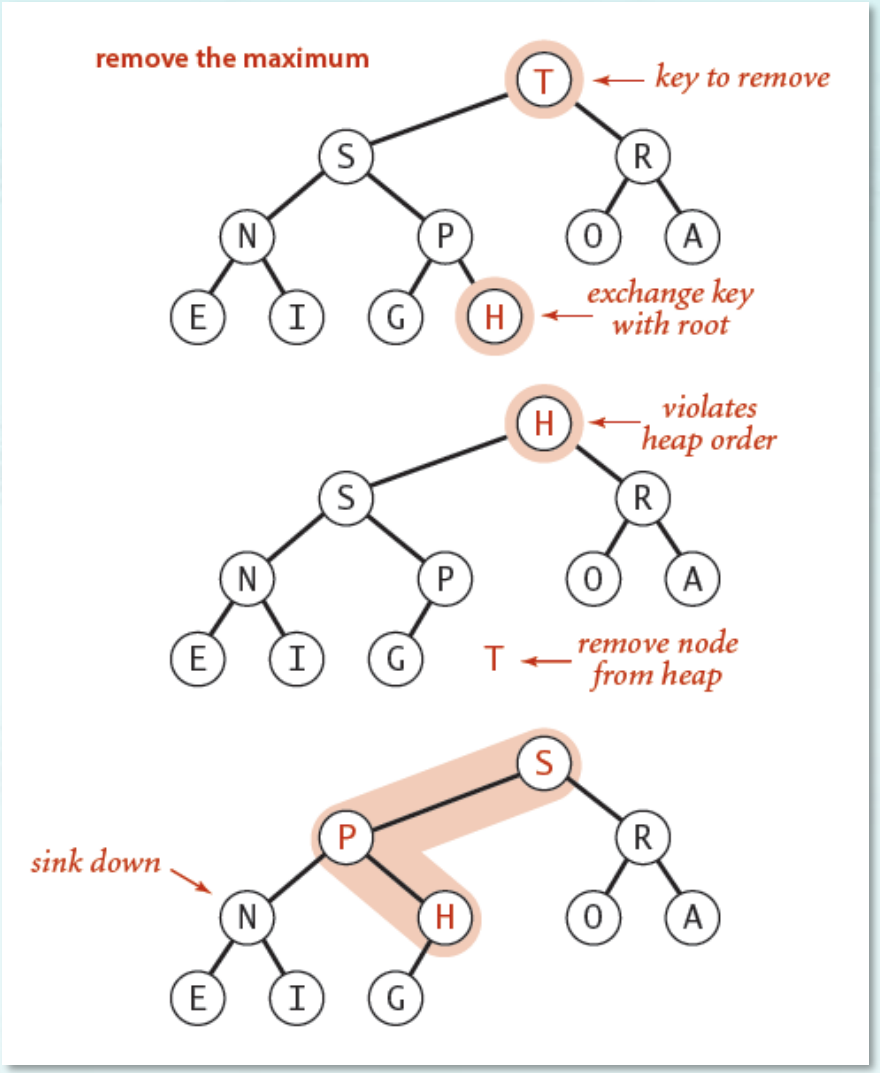
```
void delete(heap h) {



}
```

```
    swap(h, ..., ... );
    sink(h, ...);
}
```

```
void swim(heap h, int k)
void sink(heap h, int k)
```

```
bool less(heap h, int p, int c)
```

```
void swap(heap h, int p, int c)
```



heap coding

Delete the root (max or min) in a heap:

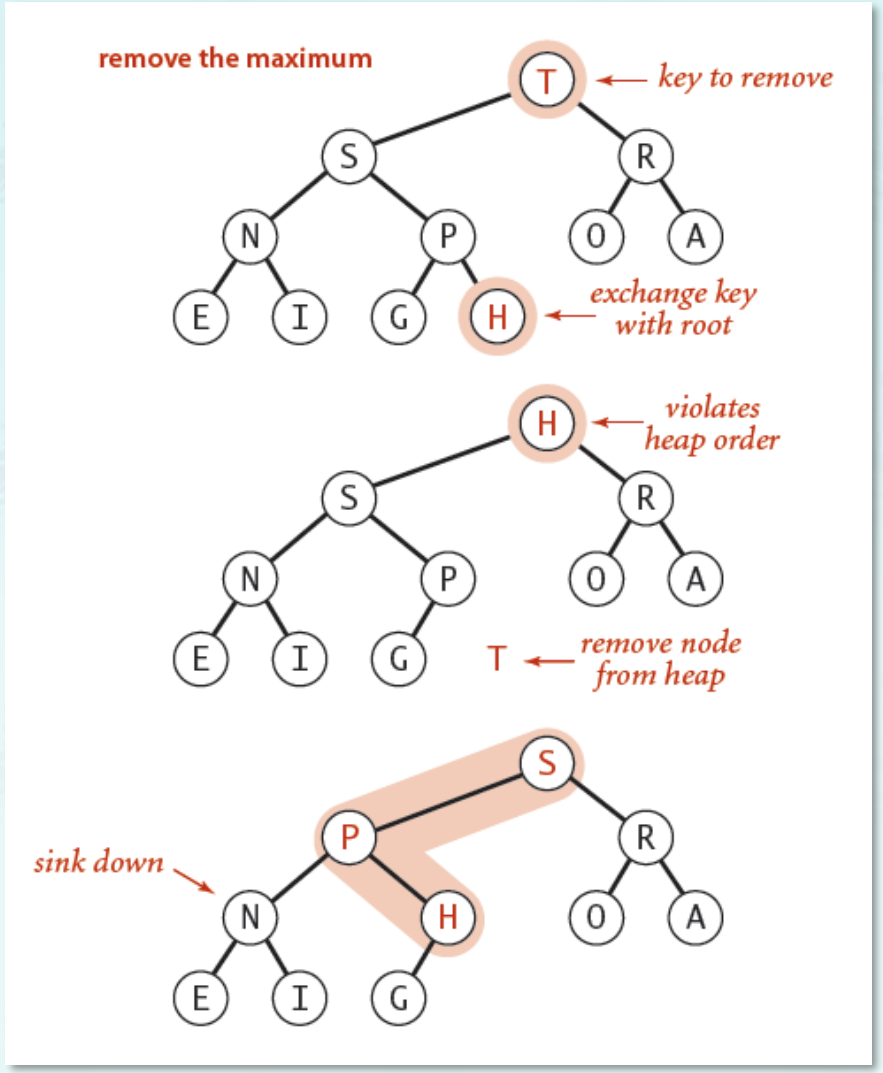
- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

```
void delete(heap h) {

}
```

```
void swim(heap h, int k)
void sink(heap h, int k)
```

```
bool less(heap h, int p, int c)
```

```
void swap(heap h, int p, int c)
```



heap coding

Delete the root (max or min) in a heap:

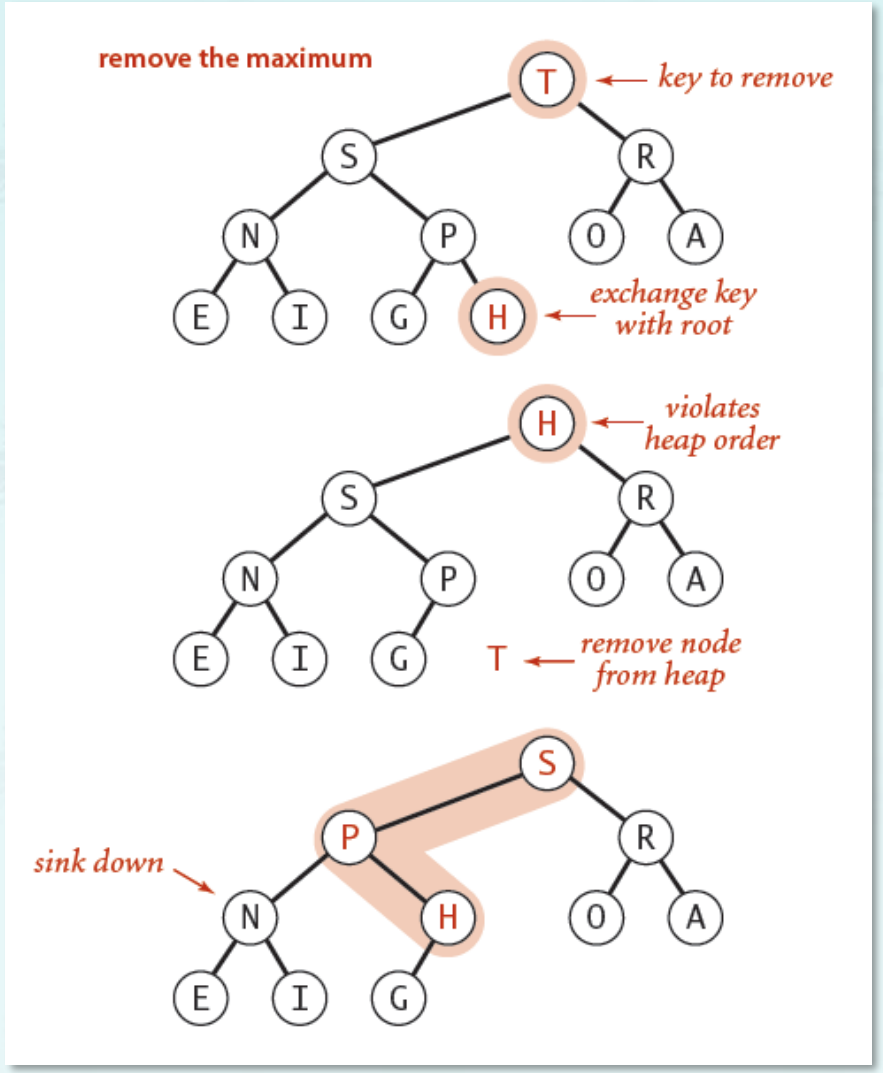
- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

```
void delete(heap h) {
    swap(h, 1, h->N--);

}
```

```
void swim(heap h, int k)
void sink(heap h, int k)
```

```
bool less(heap h, int p, int c)
```

```
void swap(heap h, int p, int c)
```



heap coding

Delete the root (max or min) in a heap:

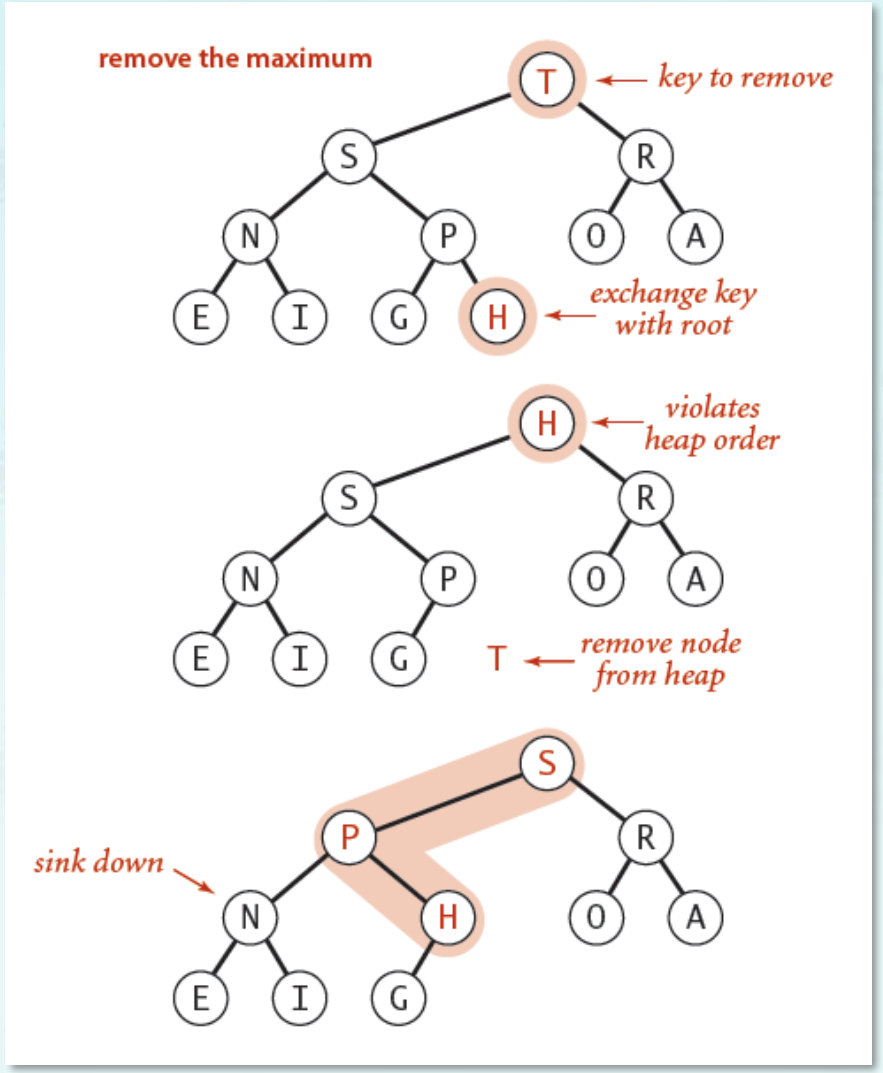
- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

```
void delete(heap h) {
    swap(h, 1, h->N--);
    sink(h, 1);
}
```

```
void swim(heap h, int k)
void sink(heap h, int k)
```

```
bool less(heap h, int p, int c)
```

```
void swap(heap h, int p, int c)
```



heap coding: heap.h

```
void clear(heap hp);                    // deallocate heap
int size(heap hp);                      // return nodes in heap currently
int level(int n);                       // return level based on num of nodes
int capacity(heap hp);                   // return its capacity (array size)
int reserve(heap hp, int capa);         // reserve the array size (= capacity)
int full(heap hp);                      // return true/false
int empty(heap hp);                     // return true/false
void grow(heap hp, int key);            // add a new key
void trim(heap hp);                     // delete a queue
int heapify(heap hp);                   // convert a complete BT into a heap

// helper functions to support grow/trim functions
int less(heap hp, int i, int j);        // used in max heap
int more(heap hp, int i, int j);        // used in min heap
void swim(heap hp, int k);              // bubble up
void sink(heap hp, int k);              // tickle down
// helper functions to check heap invariant
int heapOrdered(heap hp);              // is heap[1..N] a heap?
```

heap coding: heap.cpp

```
// return the number of items in heap
int size(heap hp) {
    return heap->N;
}
```

```
// Is this heap empty?
int empty(heap hp) {
    return (heap->N == 0) ? true : false;
}
```

```
// Is this heap full?
int full(heap hp) {
    return (heap->N == heap->capacity - 1) ? true : false;
}
```

heap coding: heap.cpp

```
int less(heap hp, int i, int j) {
    return heap->nodes[i] < heap->nodes[j];
}
```

```
void swap(heap hp, int i, int j) {
    Key t = heap->nodes[i];
    heap->nodes[i] = heap->nodes[j];
    heap->nodes[j] = t;
}
```

```
void swim(heap hp, int k) {

}
```

```
void sink(heap hp, int k) {

}
```

heap coding: heap.cpp

```
void grow(heap hp, int key) {

    cout << "YOUR CODE HERE\n";

    // add key @ ++heap->N

    // swim up @ heap->N

}
```

```
void trim(heap hp) {
    if (empty(heap)) return;

    cout << "YOUR CODE HERE\n";

}
```

heap coding: heap.cpp

newCBT()	with a given array, instantiate a new complete binary tree its result is neither maxheap nor minheap.
heapify()	make a complete binary tree into a (max) heap
heapsort()	use max/min-heap to sort elements in heap

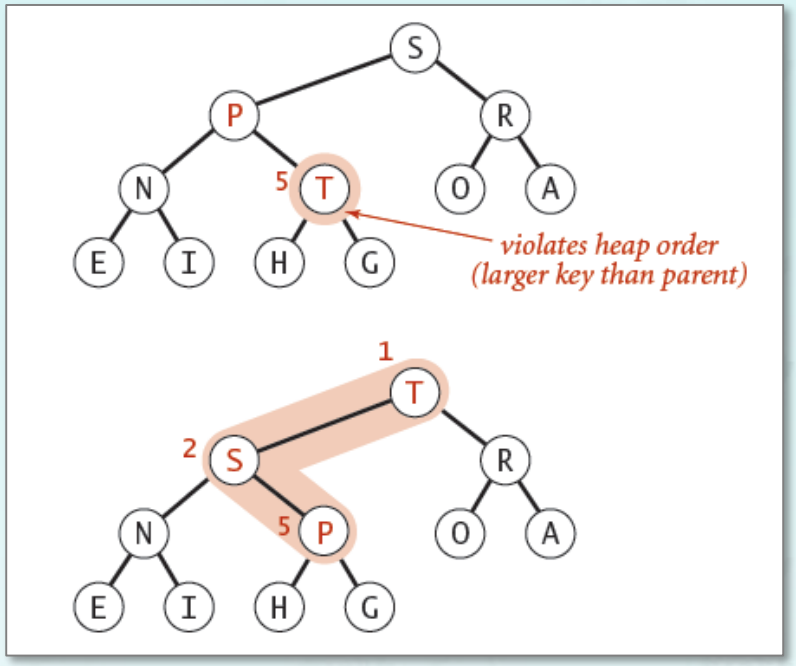
heap coding: heap.cpp

```
// instantiates a CBT with given data and its size.
heap newCBT(int *a, int n) {
    int capacity = ?

    heap p = new Heap{ capacity };

    p->N = n;
    for (int i = 0; i < n; i++)
        p->nodes[i + 1] = a[i];
    return p;
}
```

heap

- *complete binary tree (review)*
- *heap and priority queues (Chapter 9)*
- *binary heap and min-heap*
- *max-heap demo*
- *max-heap coding*
- *heap sort (Chapter 7)*

Heap ADT: heap implementation:

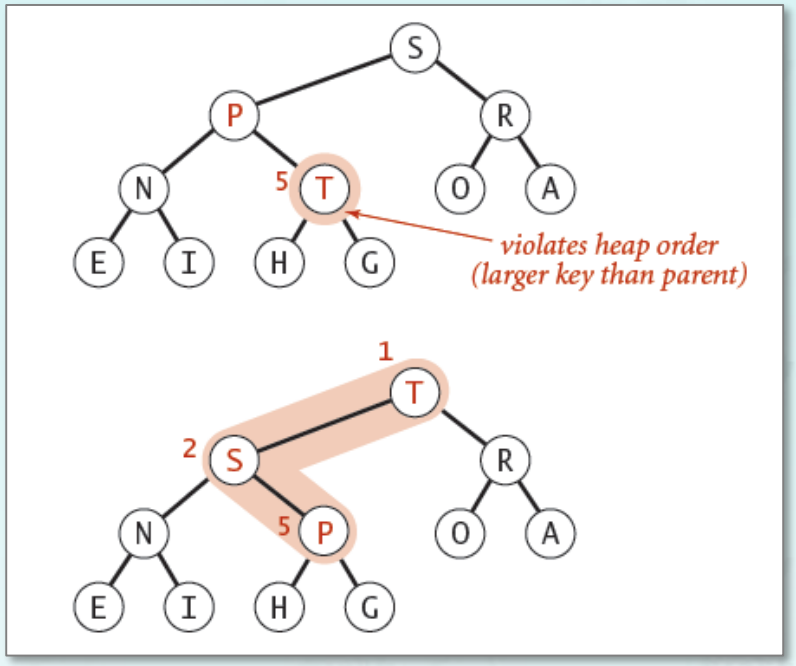



```
void swim(heap hp, int k) {
    while (k > 1 &&                    ) {
    }
}
```

parent of k

while (parent is smaller)

Heap ADT: heap implementation:

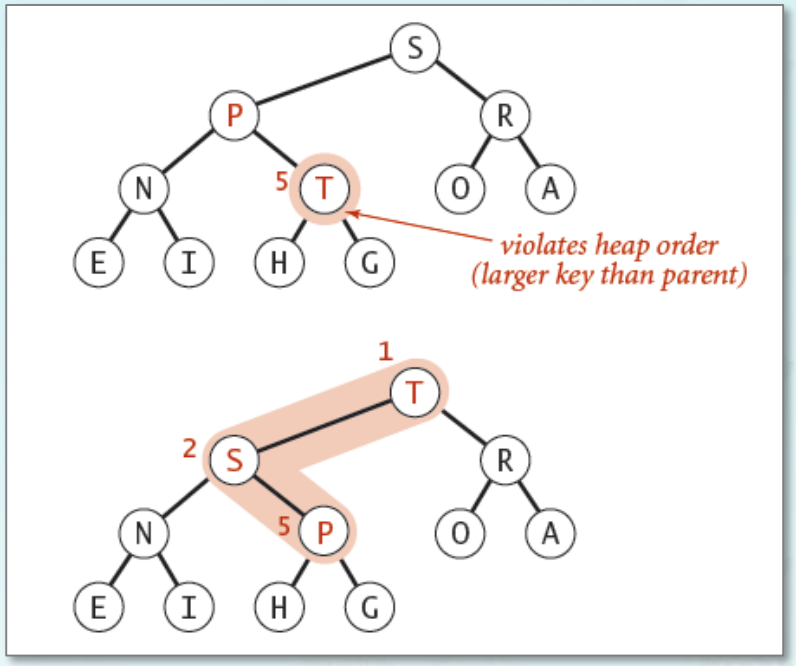



```
void swim(heap hp, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        
    }
}
```

while (parent is smaller)

parent of k

Heap ADT: heap implementation:

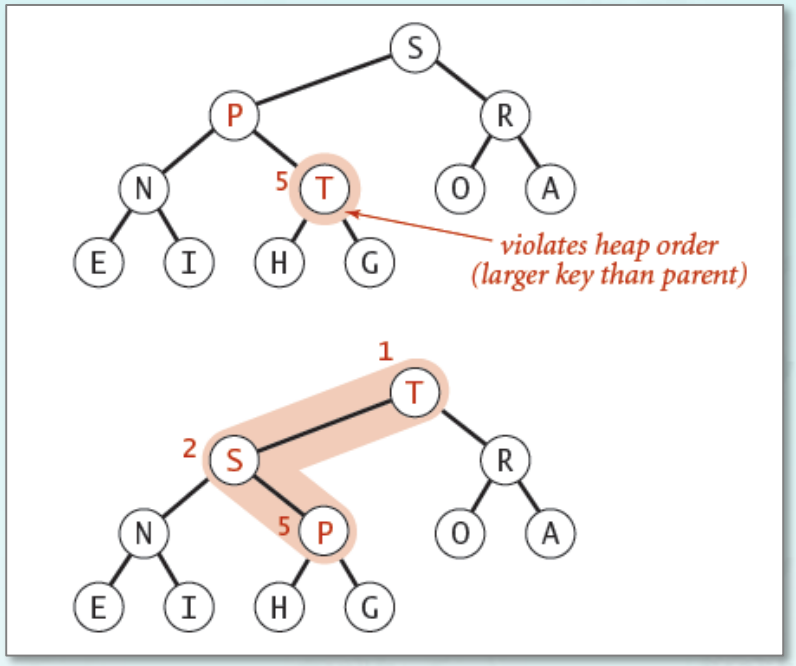



```
void swim(heap hp, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        swap(heap, k/2, k);
        [                          ]
    }
}
```

while (parent is smaller)

parent of k

Heap ADT: heap implementation:




```
void swim(heap hp, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        swap(heap, k/2, k);
        k = k/2;
    }
}
```

while (parent is smaller)

parent of k

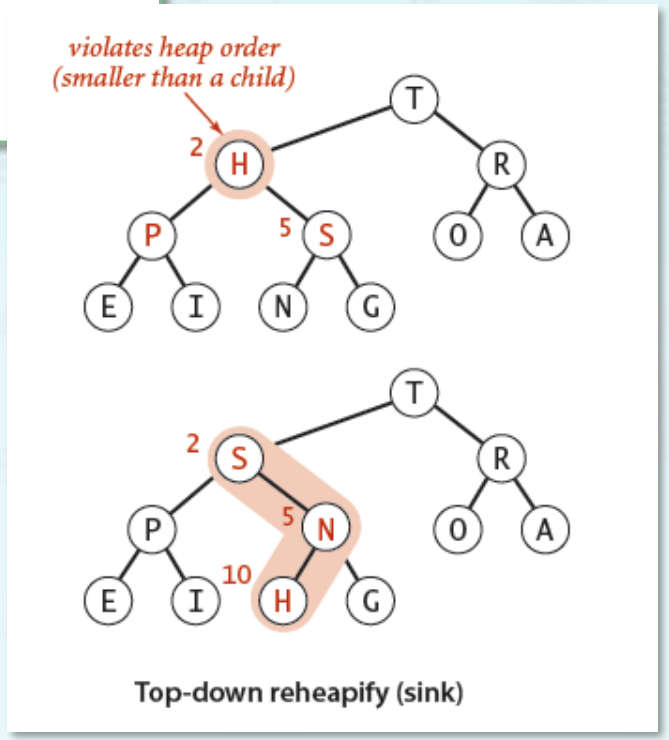
Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (
        int
        if
        if
    }
}
```

compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
while (2 * k <= N)
while (2 * k < N)
while (k <= N)
while (k < N)
```



Top-down reheapify (sink)

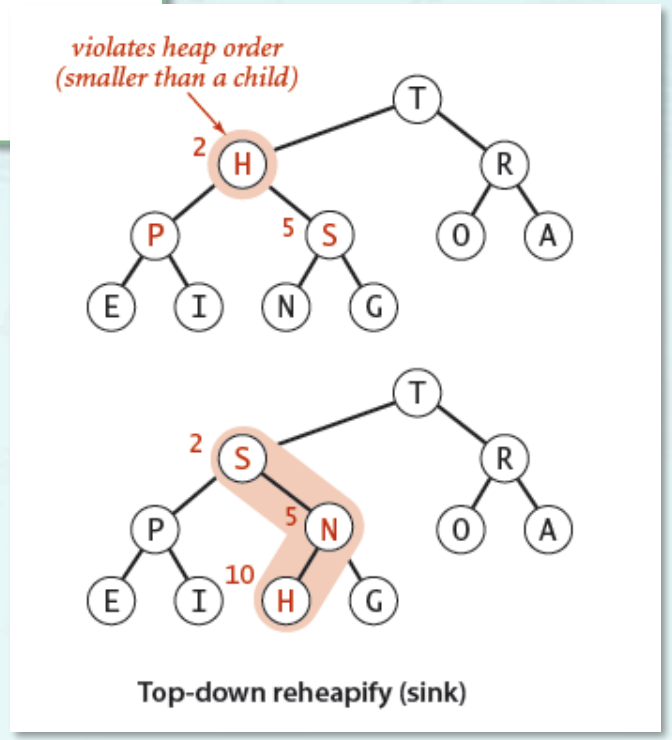
Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int
        if
        if
    }
}
```

compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
int j = 2k + 1;
```



Top-down reheapify (sink)

Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if
        if
    }
}
```

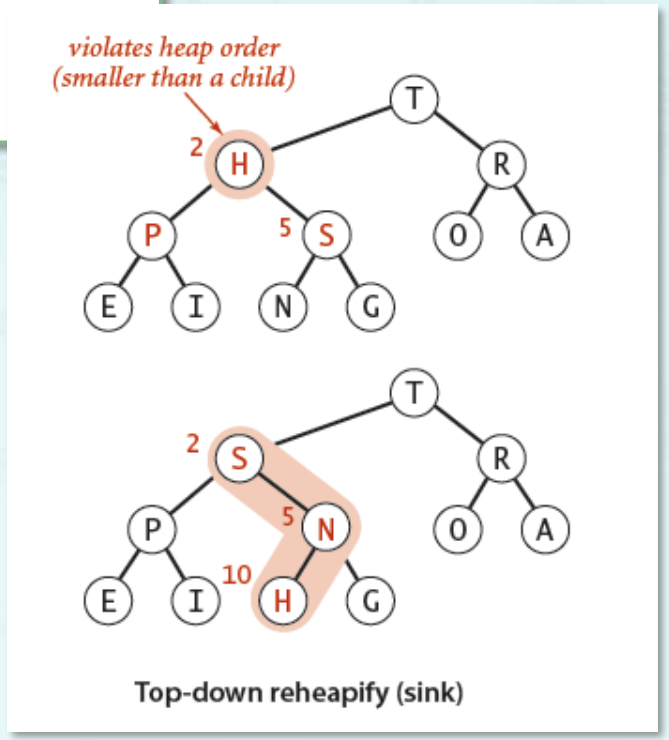
compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
```

Now let j points to the larger child;

```
if (less( ?? )) ??;
```



Top-down reheapify (sink)

Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if
        if
    }
}
```

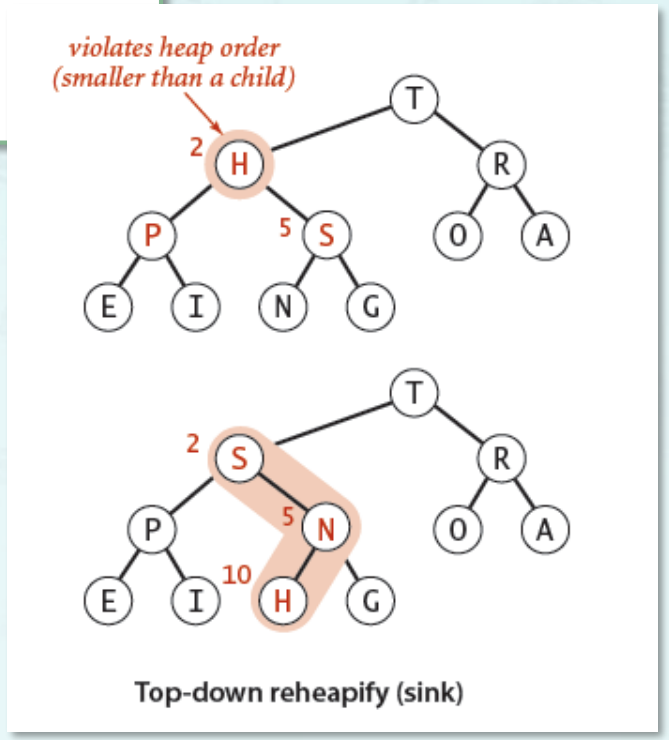
compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
```

Now let j points to the larger child;

```
less(heap, j, j + 1) j++;
```



Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if
        if
    }
}
```

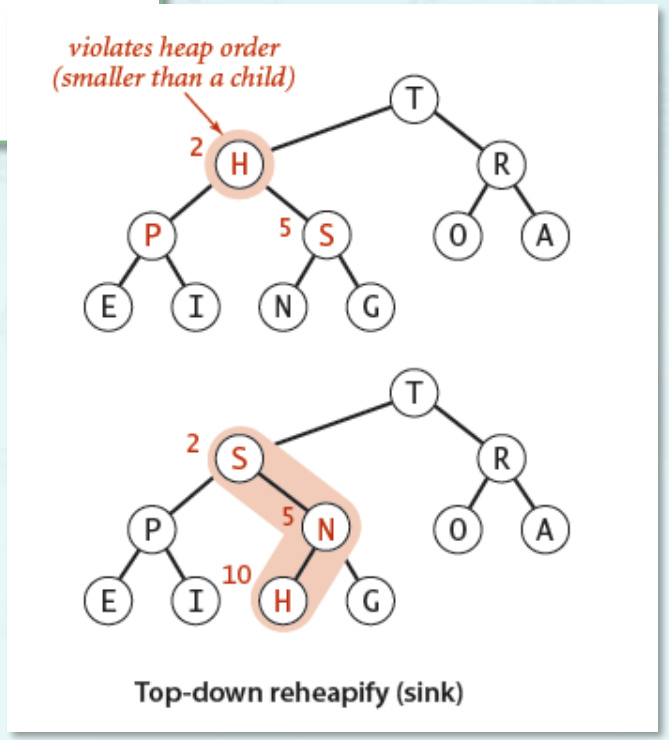
compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
```

Now let j points to the larger child;
- if there is one child, keep j as it is. – skip this.
- if there are two children,
  compare two children and increment j if necessary

```
if (j < ?? && less( ?? )) ??;
```



Top-down reheapify (sink)

Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if
        if
    }
}
```

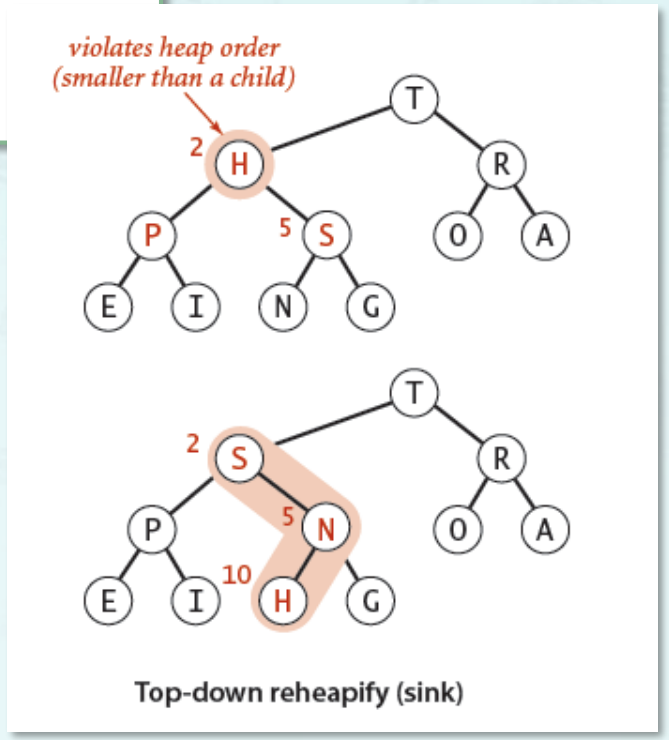
compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
```

Now let j points to the larger child;
- if there is one child, keep j as it is. – skip this.
- if there are two children,
  compare two children and increment j if necessary

```
if (j < N && less(heap, j, j + 1)) j++;
```



Top-down reheapify (sink)

Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if (j <N && less(heap, j, j + 1)) j++;
        if
    }
}
```

compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
if (j < N && less(heap, j, j + 1)) j++;
```

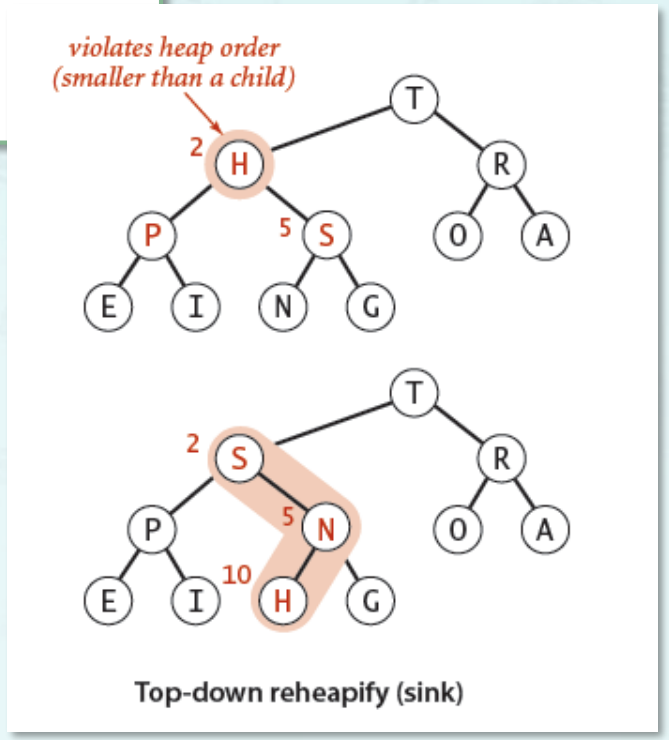
Now compare k and j (or k's child – larger one);

- if k is larger, there is nothing to do! break

```
if (!less( ?? )) break;
```

- if k is smaller, swap k and j (…k is sinking down)

```
swap( ?? );
```



Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if (j <N && less(heap, j, j + 1)) j++;
        if (!less(heap, k, j)) break;
        swap(heap, k, j);
    }
}
```

compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
int j = 2k;
if (j < N && less(heap, j, j + 1)) j++;
```

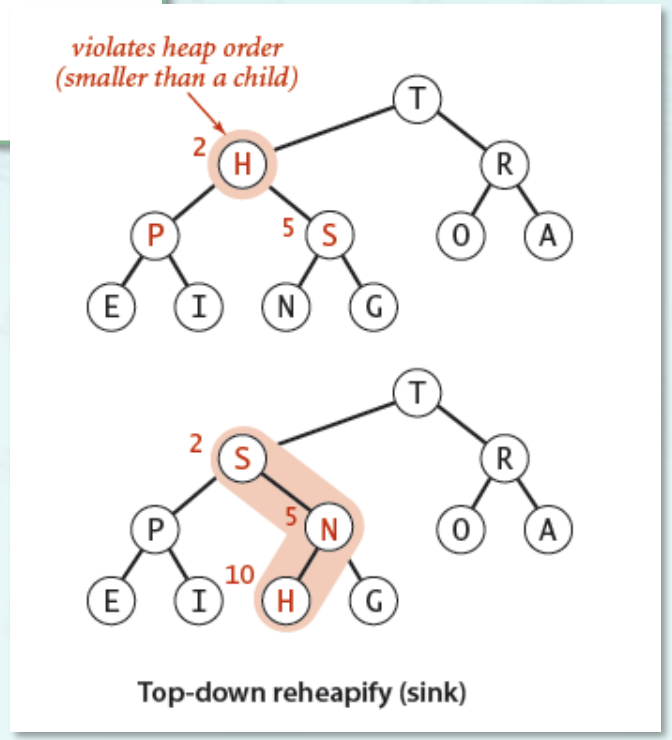
Now compare k and j (or k's child – larger one);

- if k is larger, there is nothing to do! break

```
if (!less(heap, k, j)) break;
```

- if k is smaller, swap k and j (…k is sinking down)

```
swap(heap, k, j);
```



Top-down reheapify (sink)

Heap ADT: heap implementation:

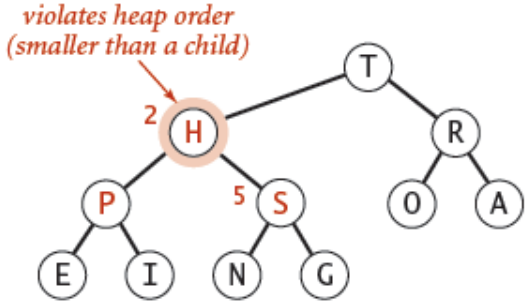
```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if (j <N && less(heap, j, j + 1)) j++;
        if (!less(heap, k, j)) break;
        swap(heap, k, j);
    }
}
```

compare two children and …

- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

What's next? Increment k?

swap … sinking down….

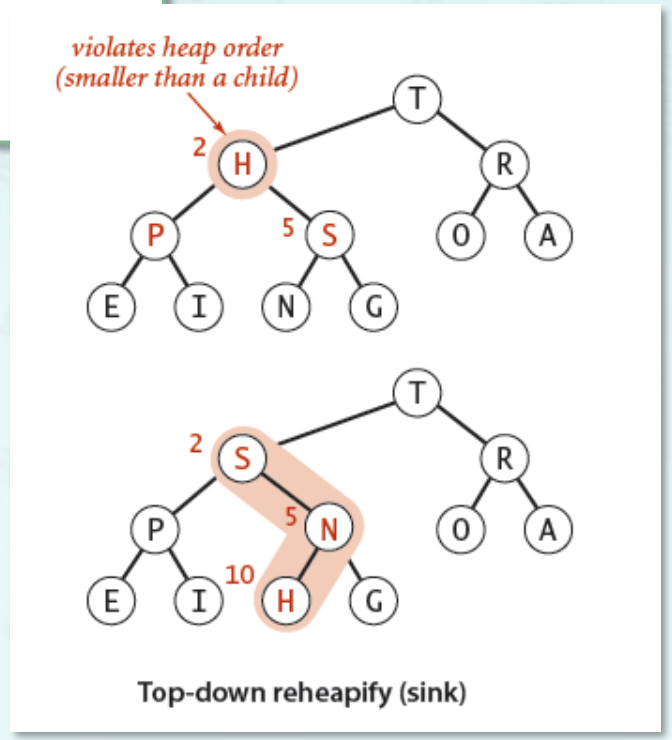


Top-down reheapify (sink)

Heap ADT: heap implementation:

```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if (j <N && less(heap, j, j + 1)) j++;
        if (!less(heap, k, j)) break;
        swap(heap, k, j);
        k = j;
    }
}
```

swap … sinking down….



Top-down reheapify (sink)