ITP20001/ECE20010 Data Structures

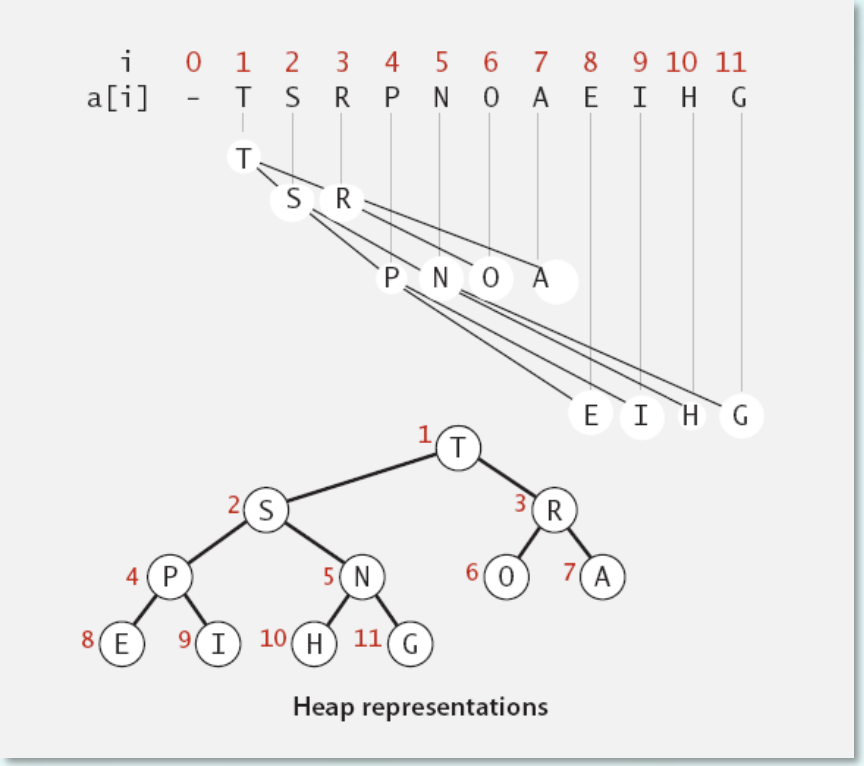
Chapter 5

- *binary search tree*
  - *Implementation*
- *HSet11*
  - *Implement Complete Binary Tree, Heap and Priority Queue (Chapter 09).*

Binary heap: array representation of a **heap-ordered** complete binary tree

- **Properties:**
  - **Heap-ordered:**
    Parent's key no smaller than children's keys. [maxheap]
  - **Heap-structure:**
    A complete binary tree

- Array representation
  - Indices start at 1.
  - Take nodes in level order.
    - Parent at k is at k/2.
    - Children at k are at 2k and 2k+1.
  - No explicit links needed!



Heap representations

Heap ADT: heap structure:

heap.h

```
typedef int    Key;
typedef struct heap_struct *heap;
typedef struct heap_struct {
    Key      *nodes;    // an array of nodes
    int      capacity;  // array size of node or key, item
    int      N;         // the number of nodes in the heap
} heap_struct;
```

Heap ADT: heaps structure:

```
heap newHeap(int capacity);                 // heap is created with capacity(or array size)
void freeHeap(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 resize(heap hp, int size);              // resize the array size (= capacity)
int isFull(heap hp);                        // return true/false
int isEmpty(heap hp);                       // return true/false
void insertMax(heap hp, Key key);           // insert in max queue
void deleteMax(heap hp);                    // delete in max queue
int heapify(heap hp);                       // convert a complete BT into a maxheap
// helper functions to support insert/delete 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 swap(heap hp, int i, int j);           // exchange two node
void swim(heap hp, int k);                  // bubble up
void sink(heap hp, int k);                  // tickle down
// helper functions to check heap invariant
int   isMaxHeap(heap hp);                   // is heap[1..N] a maxheap?
```

Heap ADT: heap implementation:

heap.h

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

```
// instantiates a new heap and return the new heap pointer.
heap newHeap(int capacity) {
  heap hp = (heap)malloc(sizeof(heap_struct));
  assert(heap != NULL);

  heap->capacity = capacity < 2 ? 2 : capacity;
  heap->nodes = (Key *)malloc(sizeof(Key)* heap->capacity);
  assert(heap->nodes != NULL);
  heap->N = 0;
  return heap;
}
```

Heap ADT: heap implementation:

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

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

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

Heap ADT: heap implementation:

```
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;
}
```

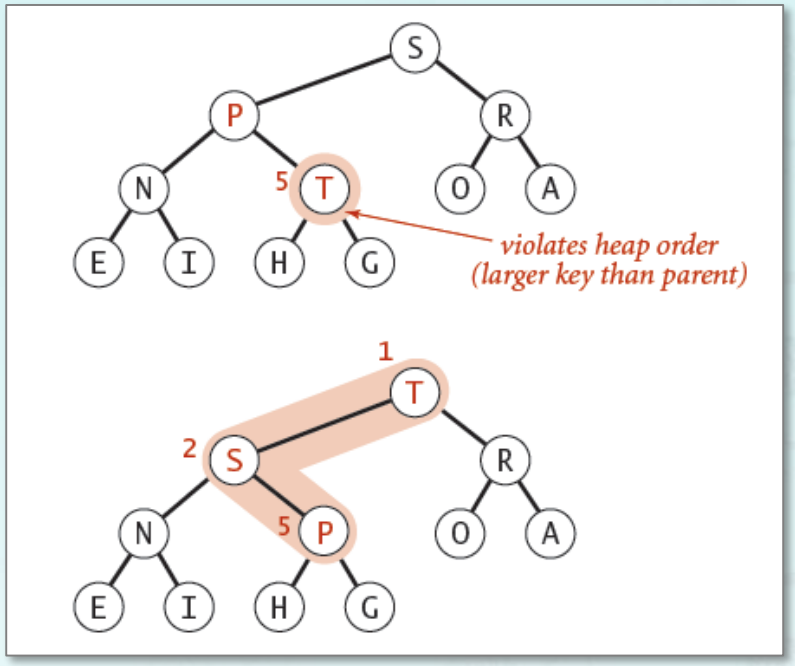
```
void swim(heap hp, int k) {

}
```

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

}
```

Heap ADT: heap implementation:

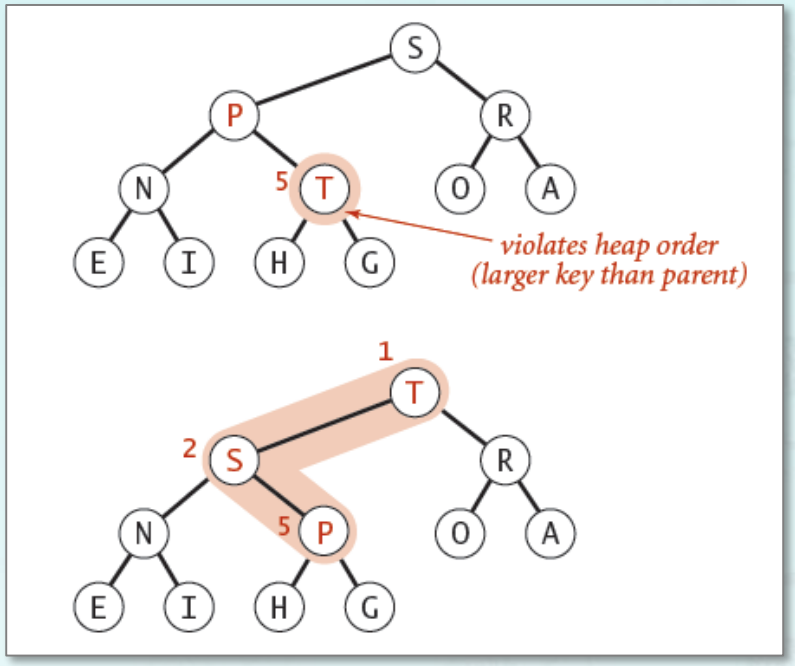



```
void swim(heap hp, int k) {
    while (k > 1 &&                          ) {
        
        
    }
}
```

while (parent is smaller)

parent of k

Heap ADT: heap implementation:

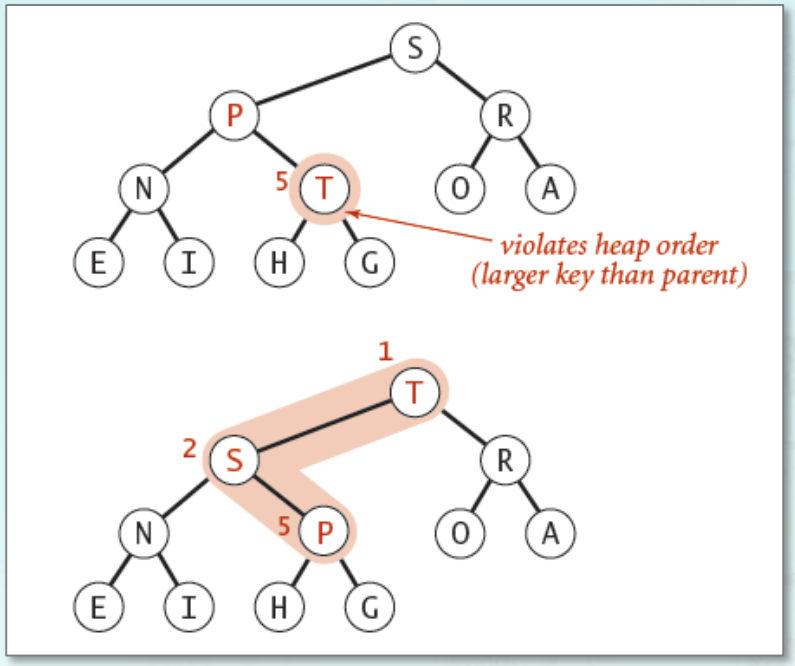



```
void swim(heap hp, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        
    }
}
```

while (parent is smaller)

parent of k

Heap ADT: heap implementation:



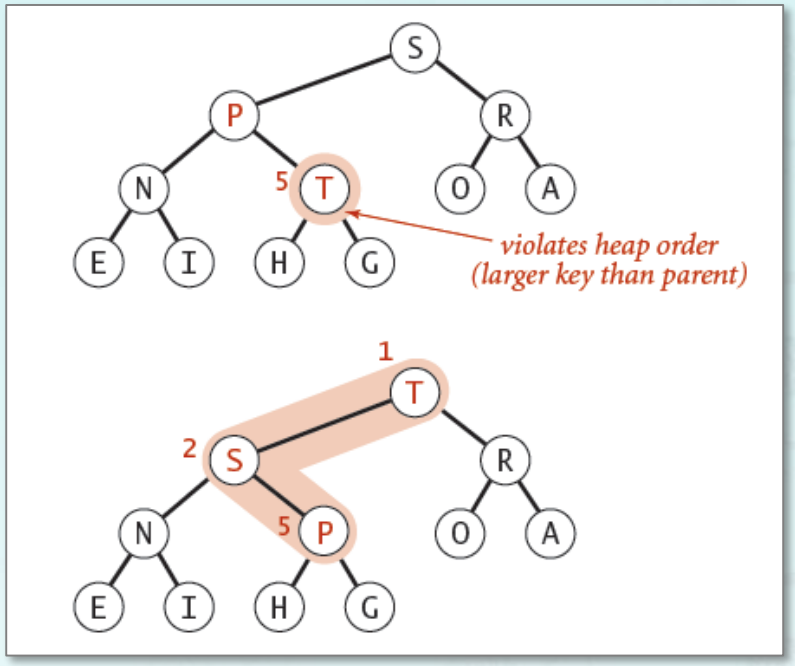

```
void swim(heap hp, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        swap(heap, k/2, k);

    }
}
```

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

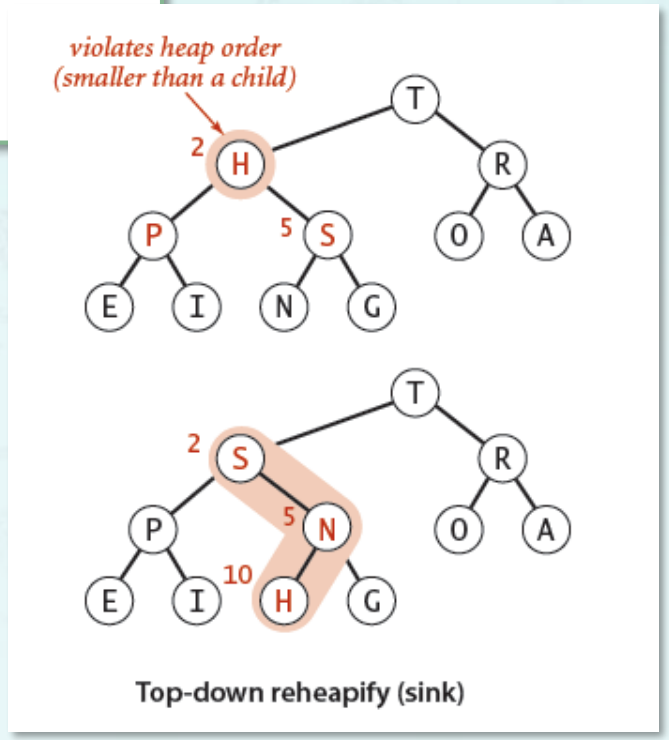
Heap ADT: heap implementation:

```
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)
```



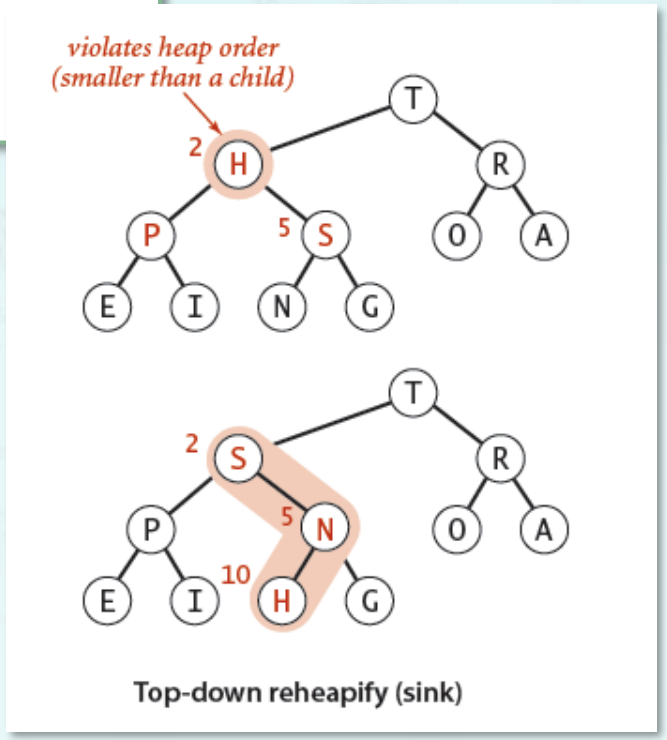
Heap ADT: heap implementation:

```
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
    }
}
```

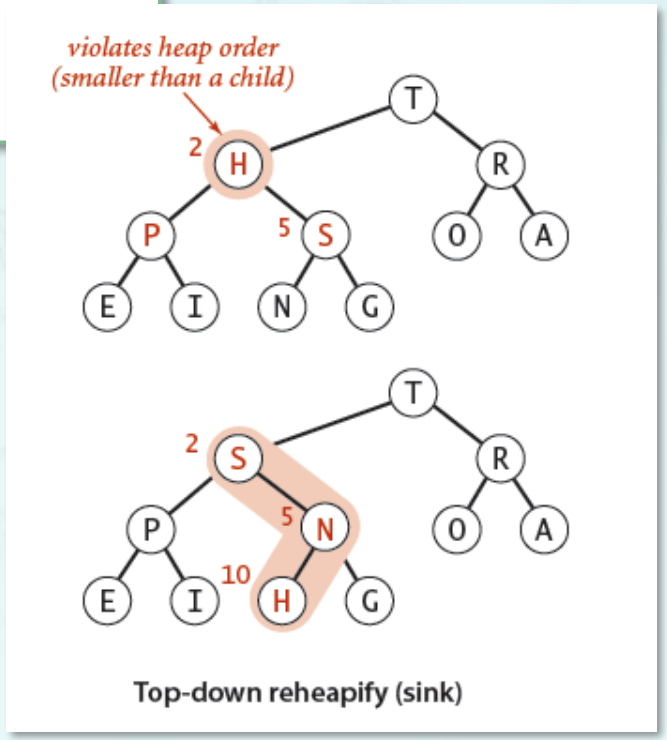
compare two children and …

- 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
    }
}
```

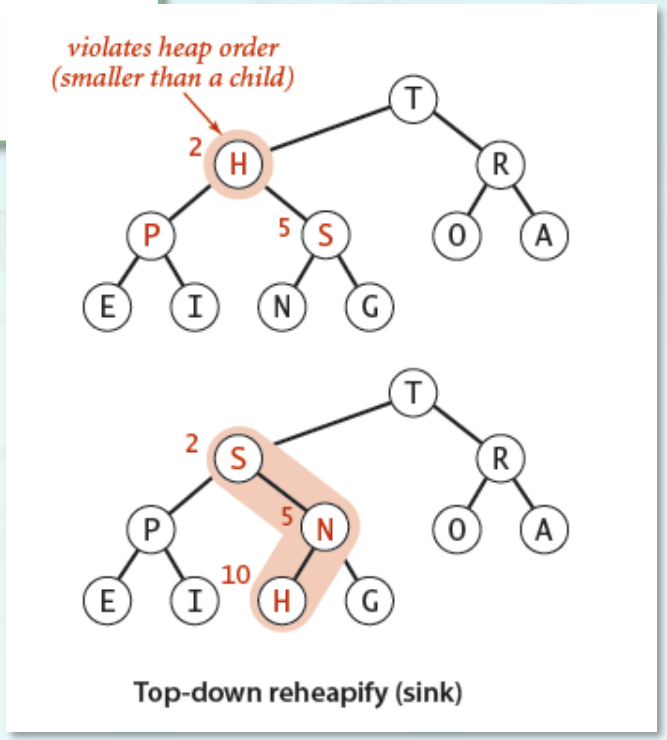
compare two children and …

- 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++;`



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 …

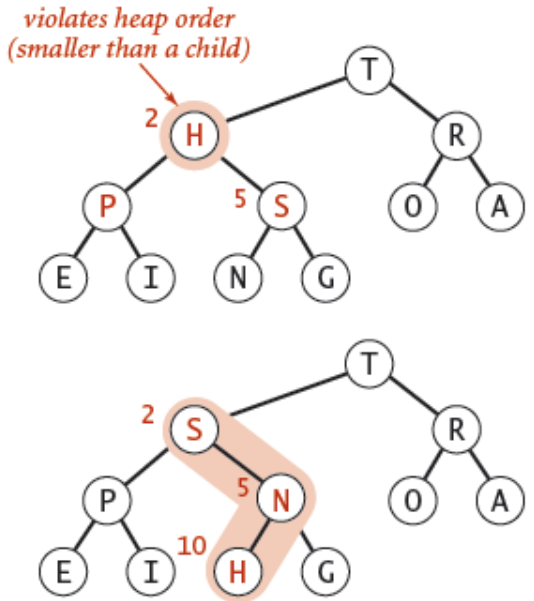
- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
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 < ?? && 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 …

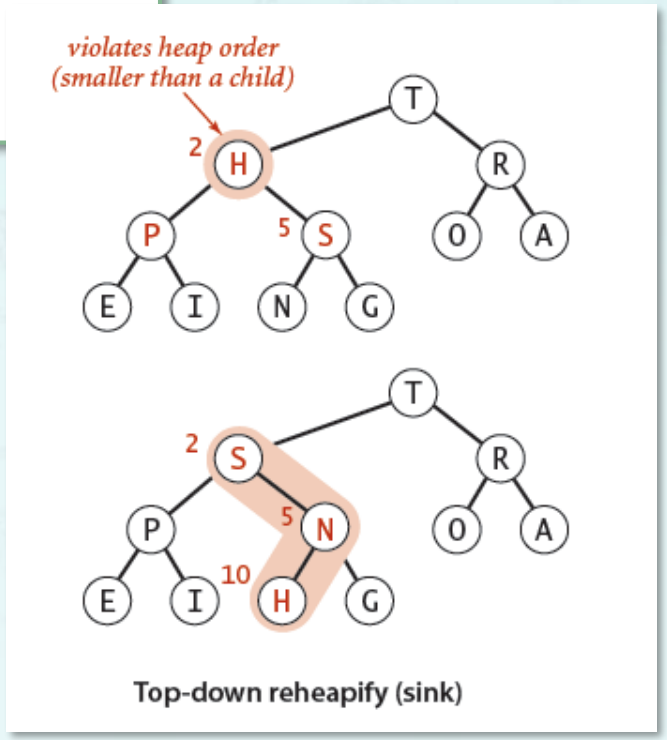
- until it reaches bottom
- let j points to left child
- select the larger child
- quit if k is larger

```
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++;
```

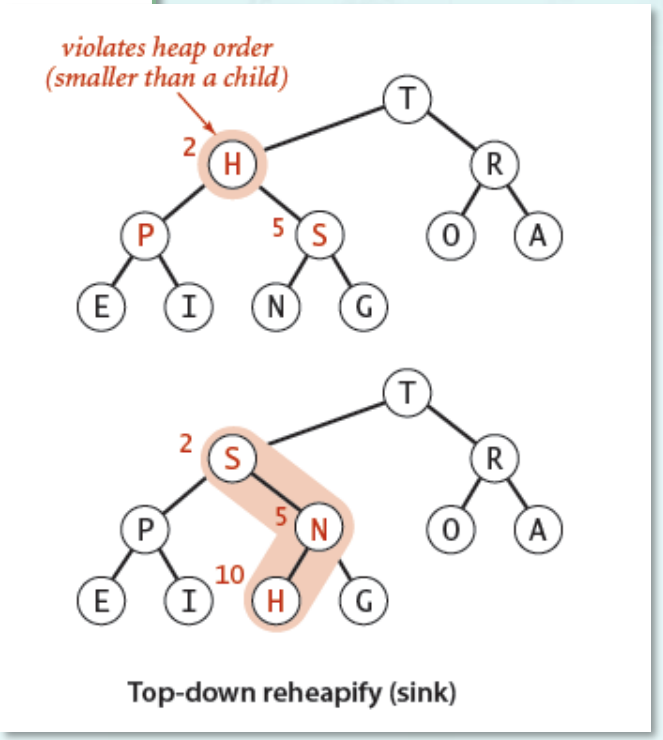
Now compare k and j (or k's child – larger one);

- 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++;
```

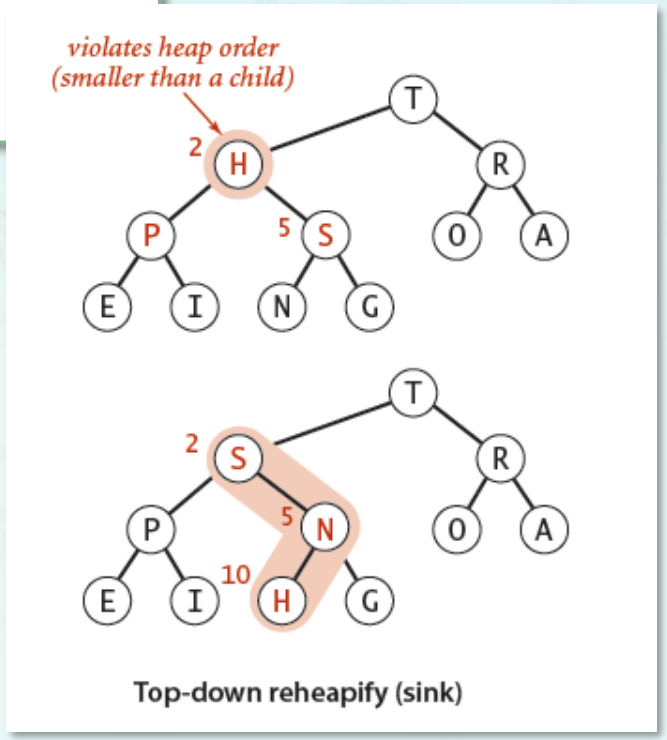
Now compare k and j (or k's child – larger one);

- 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:

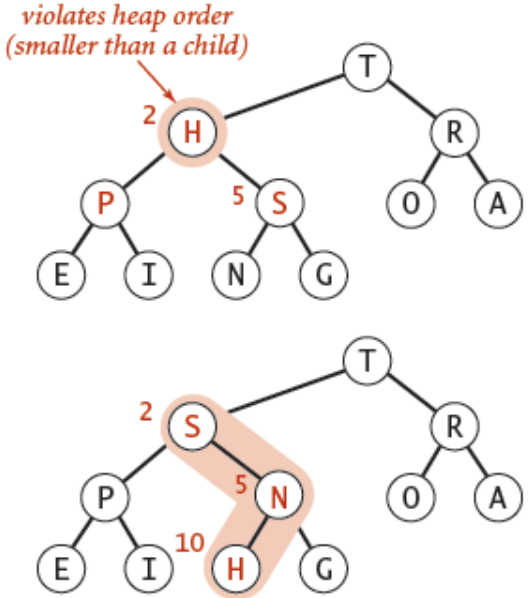
```
void sink(heap hp, int k) {
    int N = heap->N;
    while (2 * k <= N) {
        int j = 2 * k;
        if (j <N && less(heap, j, j + 1)) j++;
        if (!less(heap, k, j)) break;
        swap(heap, k, j);
    }
}
```

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….

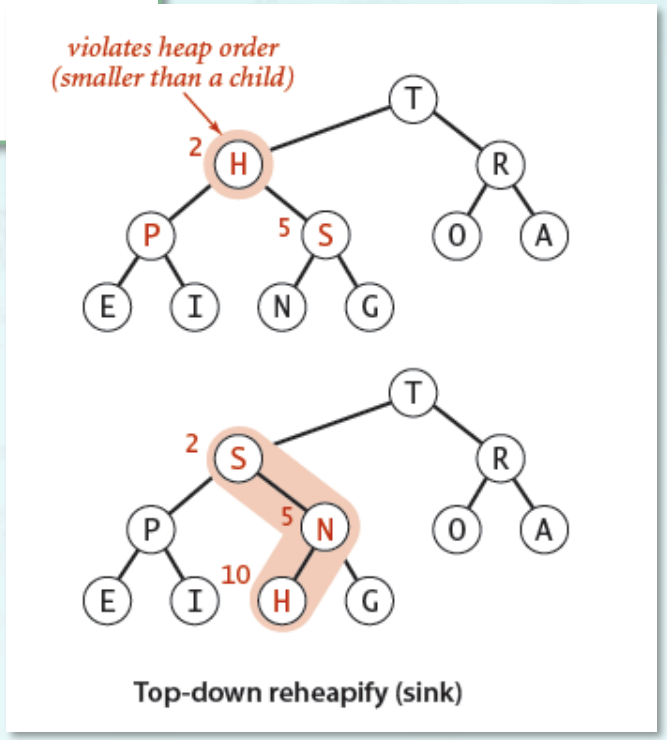


Top-down reheapify (sink)

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)

Heap ADT: heap implementation:

```
void insertHeap(heap hp, Key key) {
    if (isFull(heap))
        printf("insertMAX: YOUR CODE HERE\n");

    // add key @ ++heap->N

    // swim up @ heap->N

}
```

```
void deleteHeap(heap hp) {
    if (isEmpty(heap)) return;

    printf("deleteMax: YOUR CODE HERE\n");

    if ((heap->N > 0) && (heap->N == (heap->capacity - 1) / 4))
        printf("deleteMax: YOUR CODE HERE\n");
}
```

- **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**

Q: What is the difference between **newCBT()** and **newHeap()**?
heap **newHeap**(int capacity)
heap **newCBT**(Key *a, int aSize)

Heap ADT: heap implementation:

```
// instantiates a CBT with given data and its size.
heap newCBT(Key *a, int aSize) {
    heap hp = newHeap(aSize + 1);
    heap->N = aSize;
    for (i = 0; i < aSize; i++)
        heap->nodes[i + 1] = a[i];
    return heap;
}
```

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

```
// instantiates a new heap and returns the new heap pointer.
heap newHeap(int capacity) {
  heap hp = (heap)malloc(sizeof(heap_struct));
  assert(heap != NULL);

  heap->capacity = capacity < 2 ? 2 : capacity;
  heap->nodes = (Key *)malloc(sizeof(Key)* heap->capacity);
  assert(heap->nodes != NULL);

  heap->N = 0;
  return heap;
}
```