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

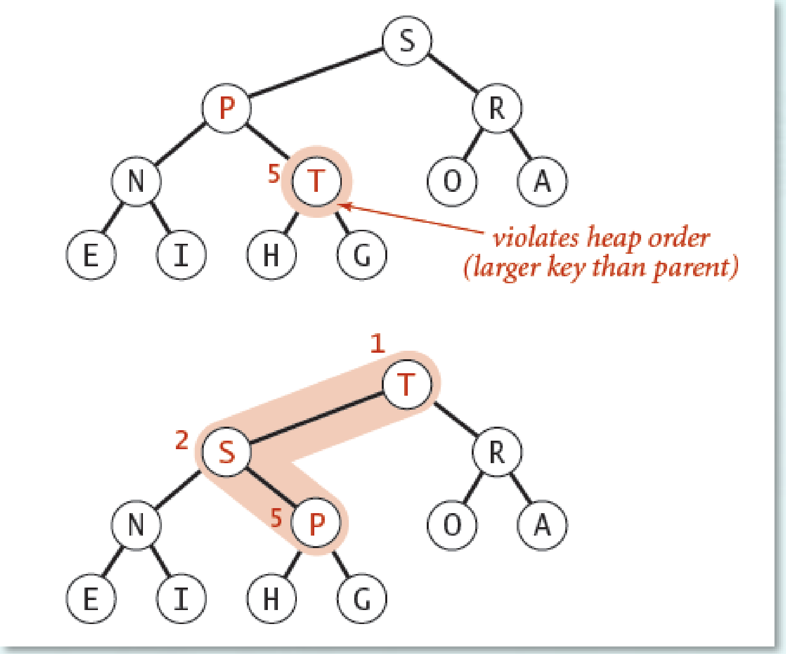
```
void clear(heap hp);                    // deallocate heap
int size(heap hp);                      // return nodes in heap currently
int level(int n);                       // return level based on num of nodes
int capacity(heap hp);                   // return its capacity (array size)
int reserve(heap 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.

This is a maxheap example.

swim up
or
sink down

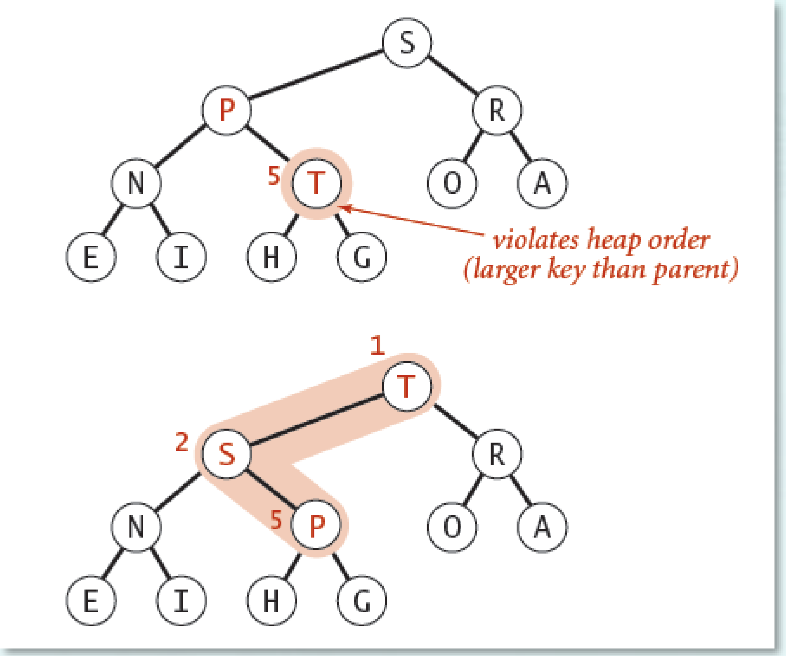


Promotion in a heap: swim

- 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];
}
```



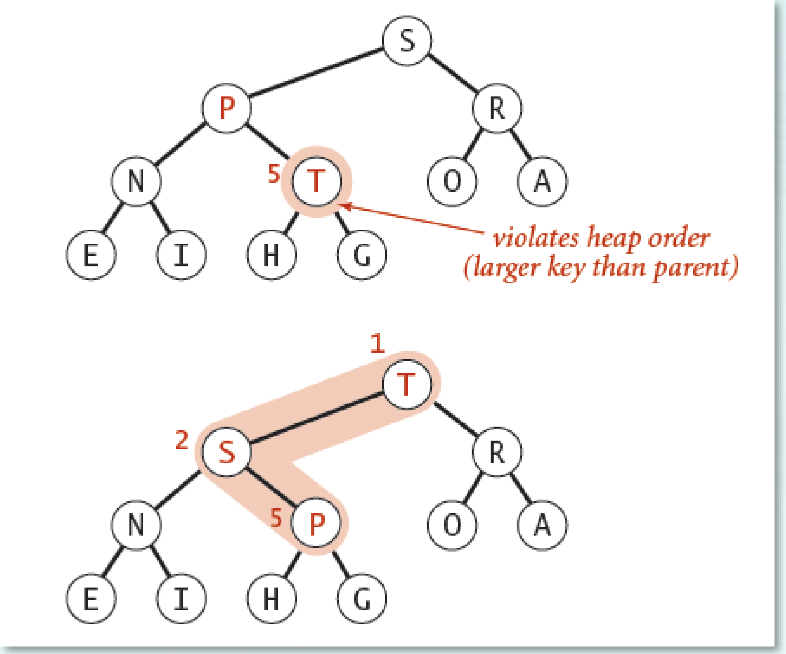
Promotion in a heap: swim

- 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) {
    int item = h->nodes[p];
    h->nodes[p] = h->nodes[c];
    h->nodes[c] = item;
}
```

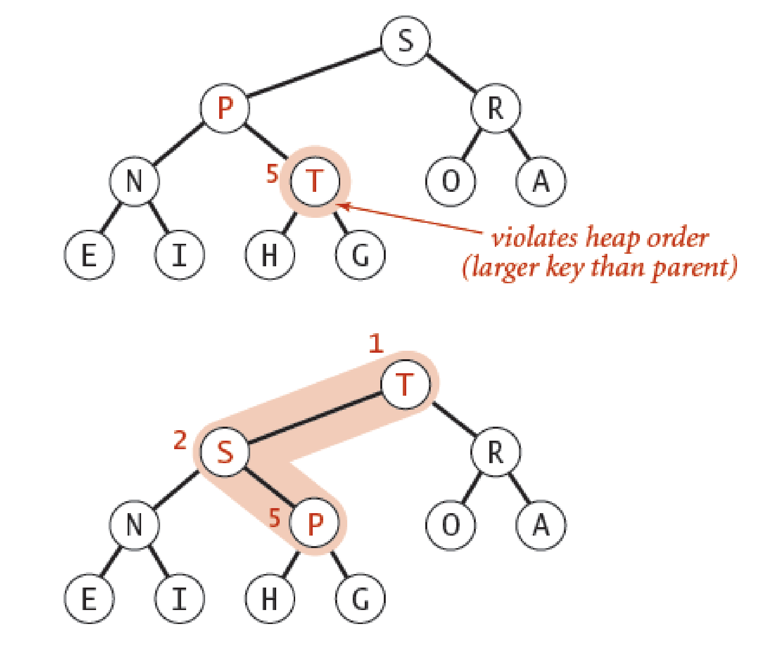


Promotion in a heap: swim

- 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) {
    int item = h->nodes[p];
    h->nodes[p] = h->nodes[c];
    h->nodes[c] = item;
}
```



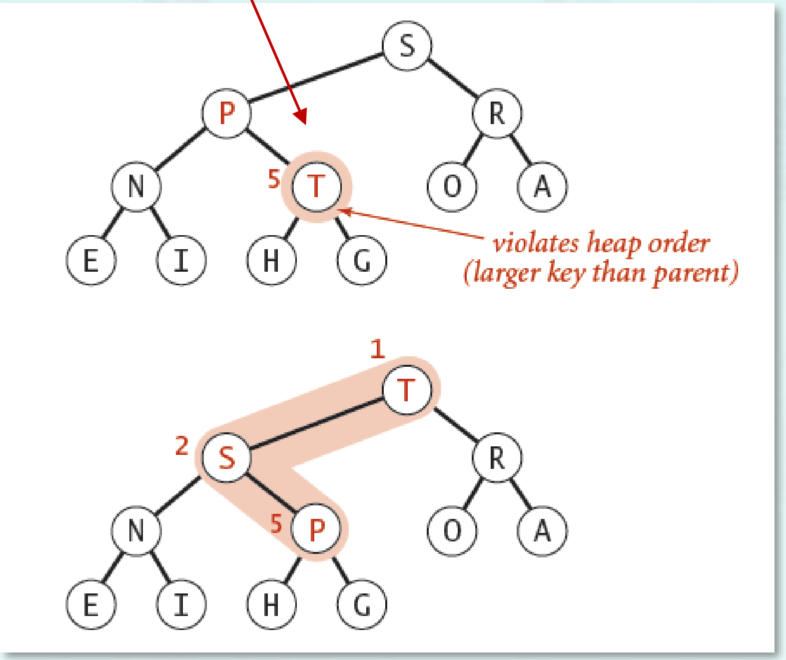
Promotion in a heap: swim

- 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)
  {

  }
}
```

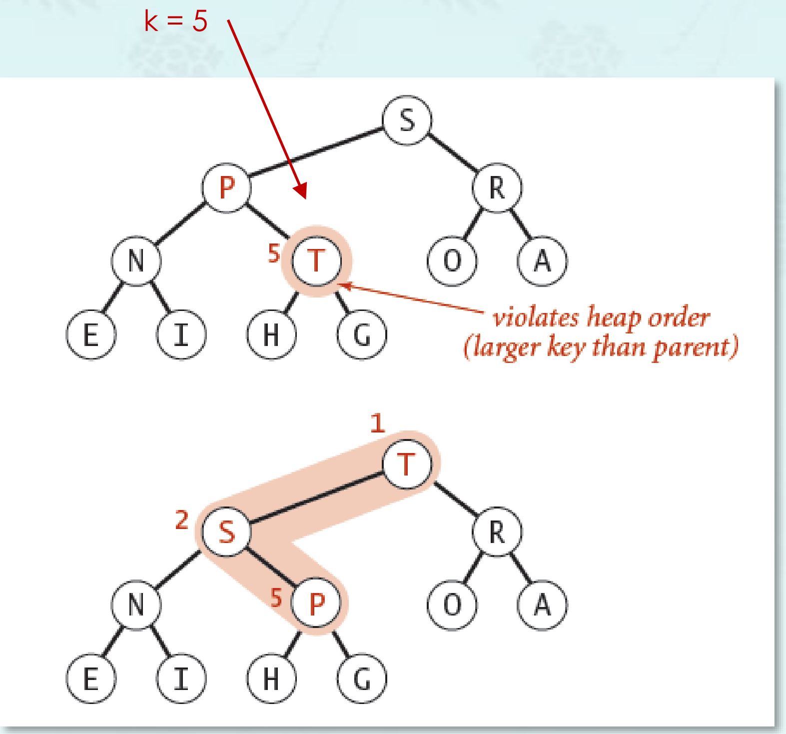
k = 5



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)
  {

  }
}
```

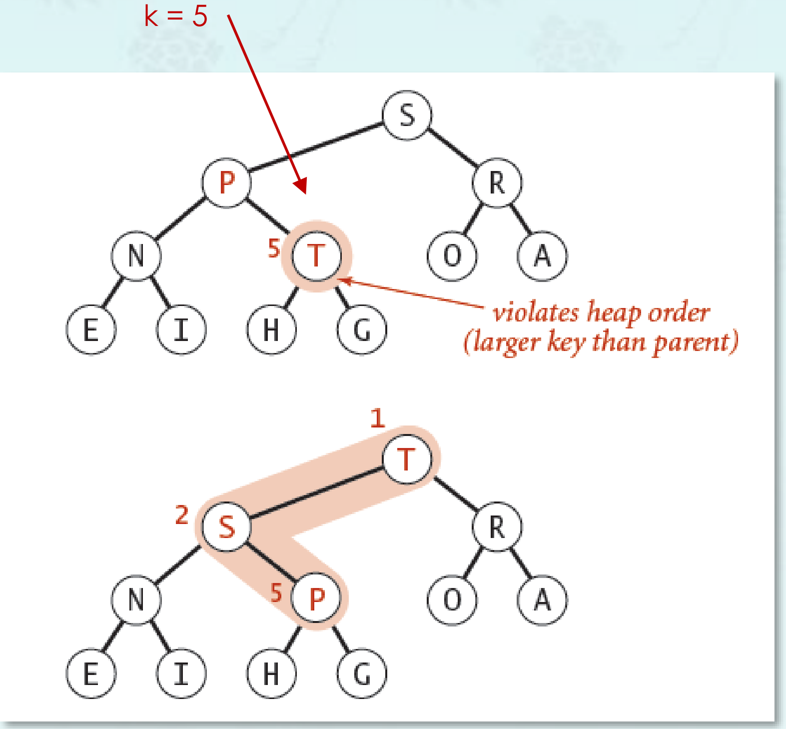


heap coding

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
  }
}
```



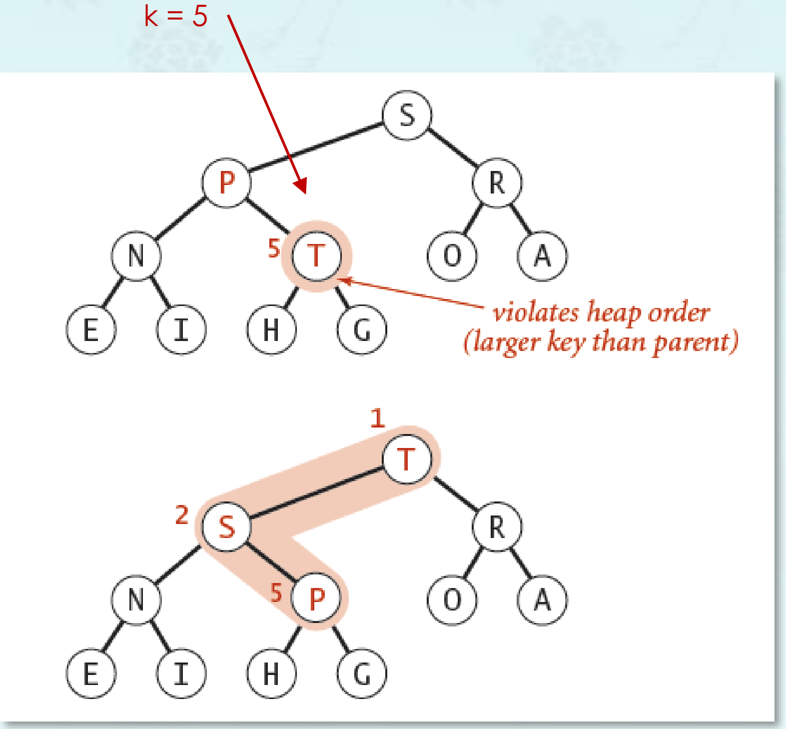
Promotion in a heap: swim

- 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  p h, int k)
{
  while (
  {

  }
}
```



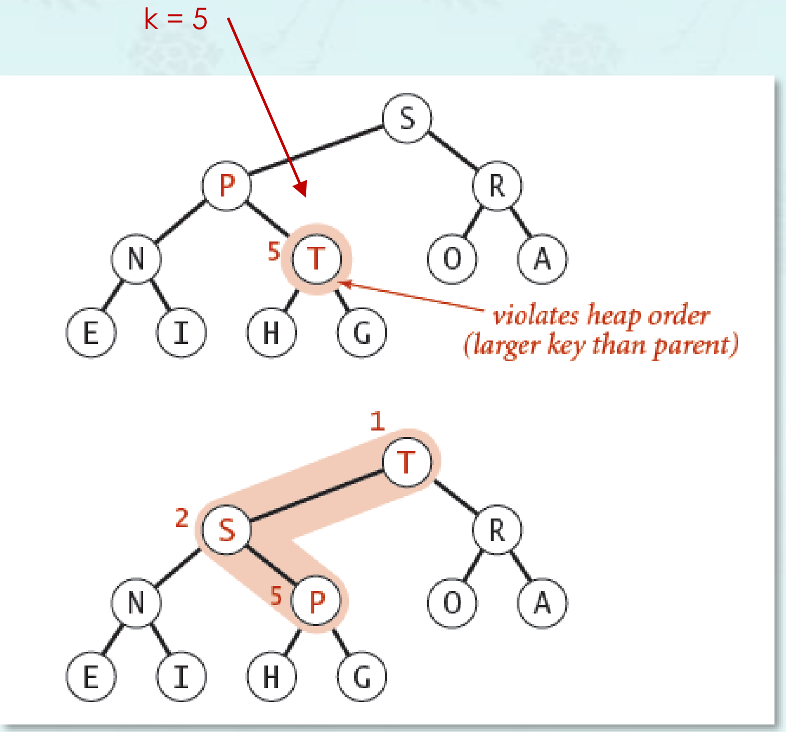
Promotion in a heap: swim

- 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(heap h, int k)
{
  while (k > 1 &&                    )
  {

  }
}
```



Promotion in a heap: swim

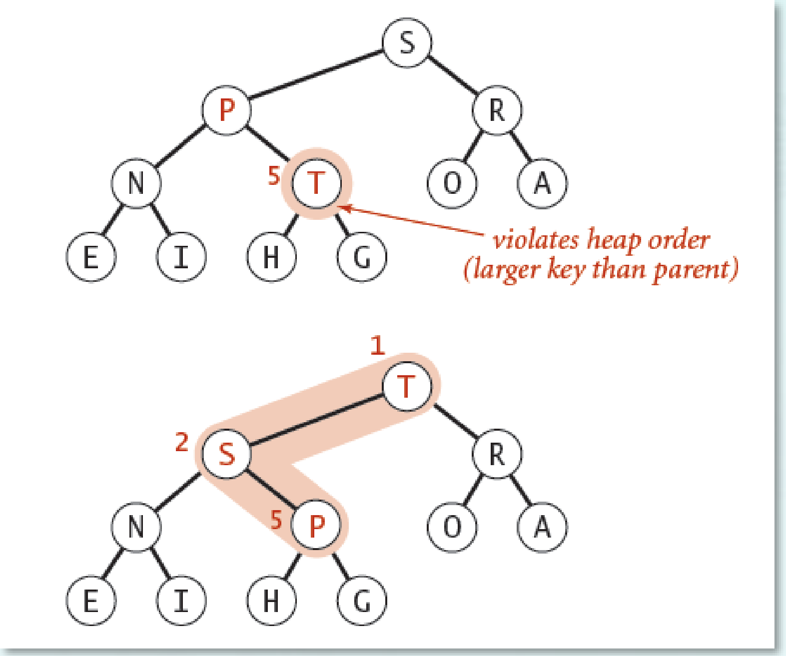
- To eliminate the violation:
  - Swap key in child with key in parent.
  - Repeat until heap order restored.

not reached at root

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

```
void swim(heap 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.

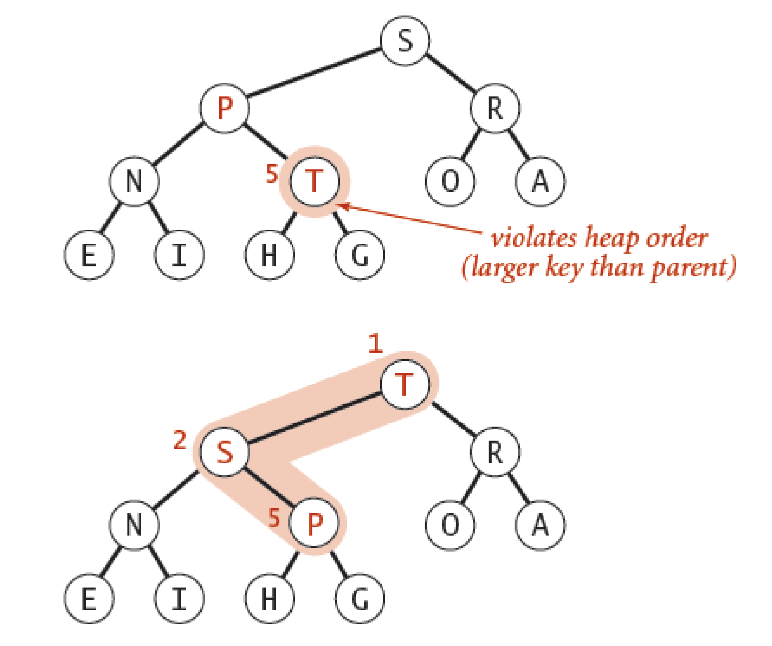
not reached at root

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

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

  }
}
```

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

move up one level



Promotion in a heap: swim

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

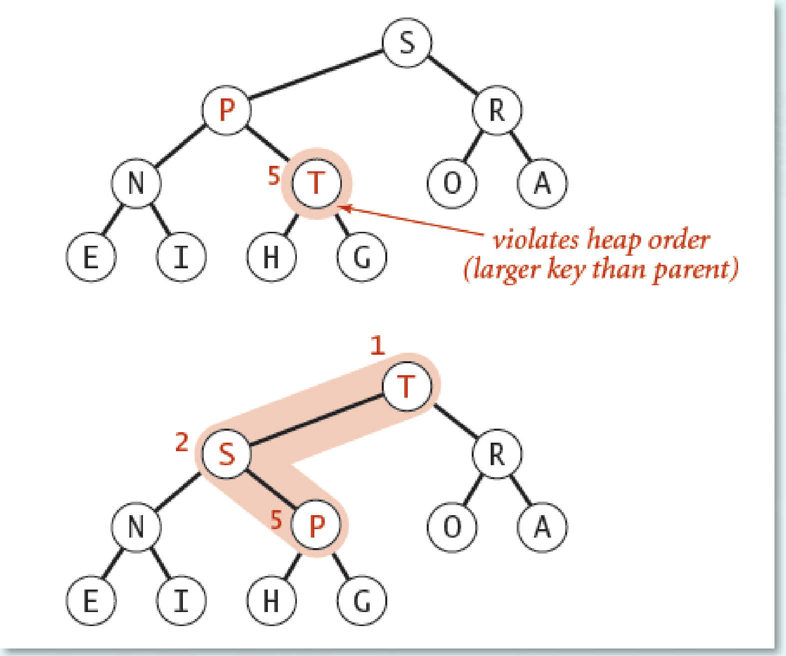
```
void swim(heap h, int k)
{
  while (k > 1 && less(h, k / 2, k))
  {
    swap(h, k / 2, k);
    k = k / 2;
  }
}
```

not reached at root

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

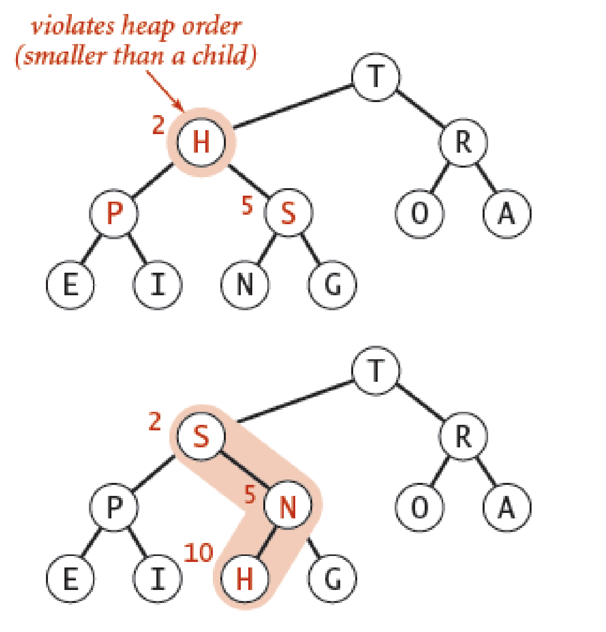
swap parent(k/2) and its child(k)

move up one level



heap coding

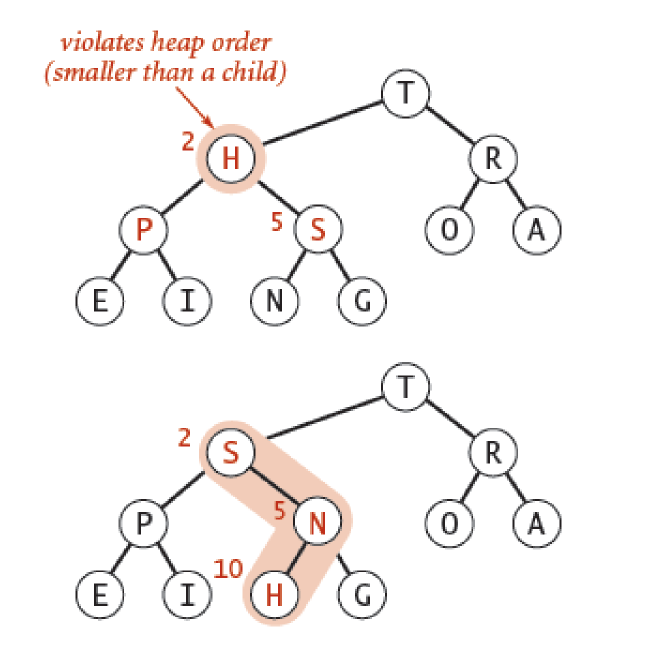
swim up
or
sink down



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: Why not smaller child?
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.



Top-down reheapify (sink)

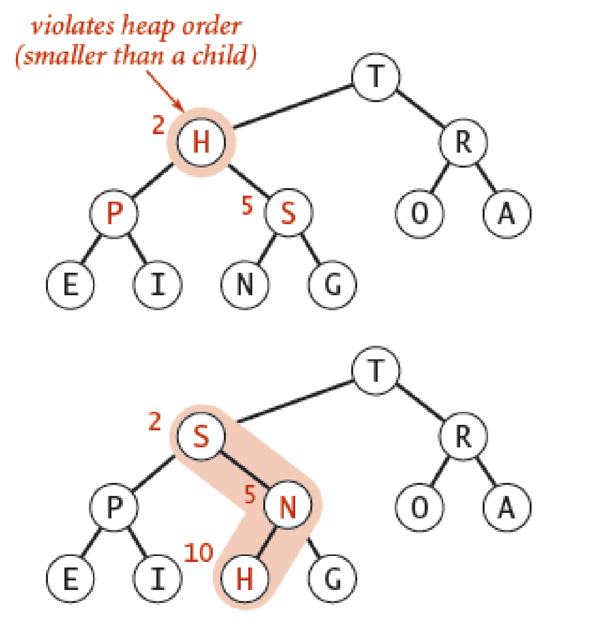
This is a maxheap example.

Demotion in a heap: sink

- Parent’s key becomes **smaller** than one (or both) of its children’s.
- To eliminate the violation: Why not smaller child?
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.

```
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



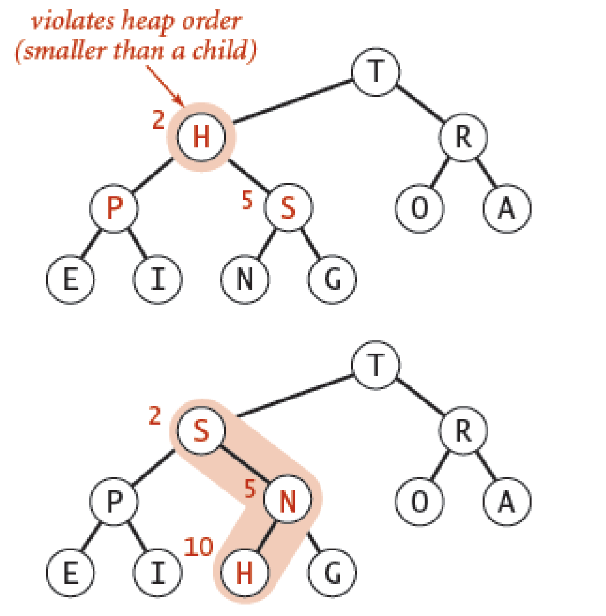
Top-down reheapify (sink)

Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation: Why not smaller child?
  - Swap key in parent with key in **larger** child (of two)
  - Repeat until heap order restored.

```
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)

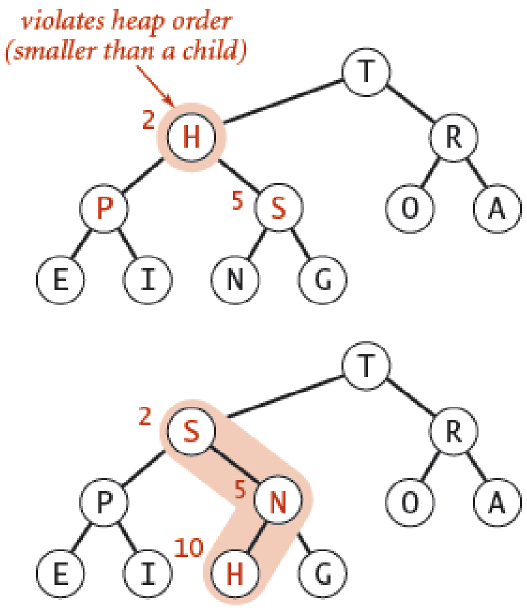
Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation: Why not smaller child?
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

```
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)

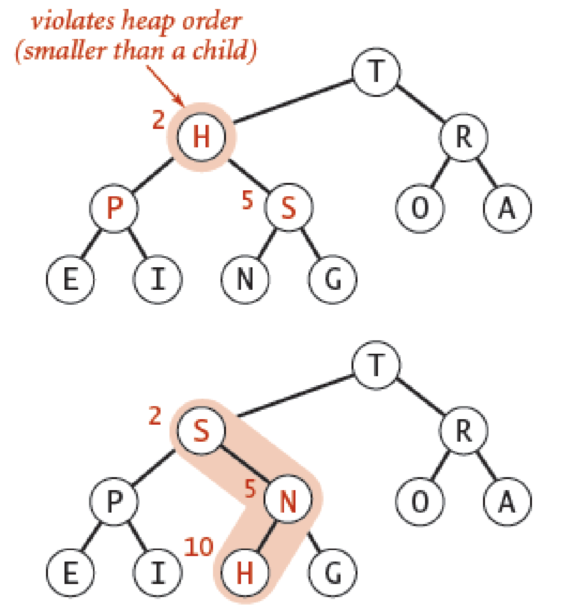
Demotion in a heap: sink

- 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

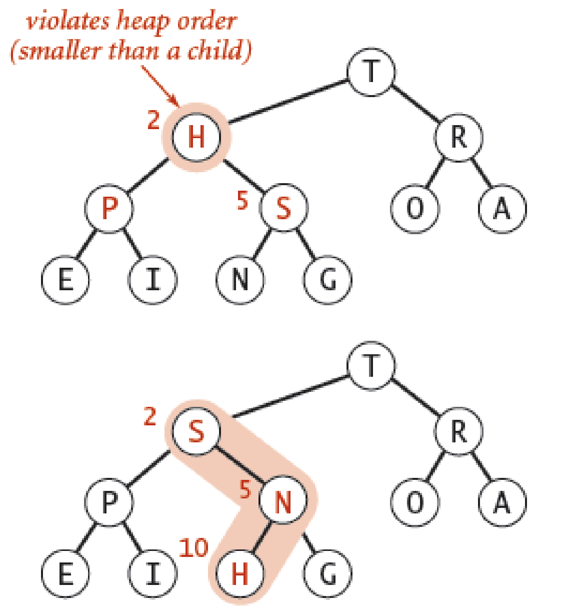
- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation:
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

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)

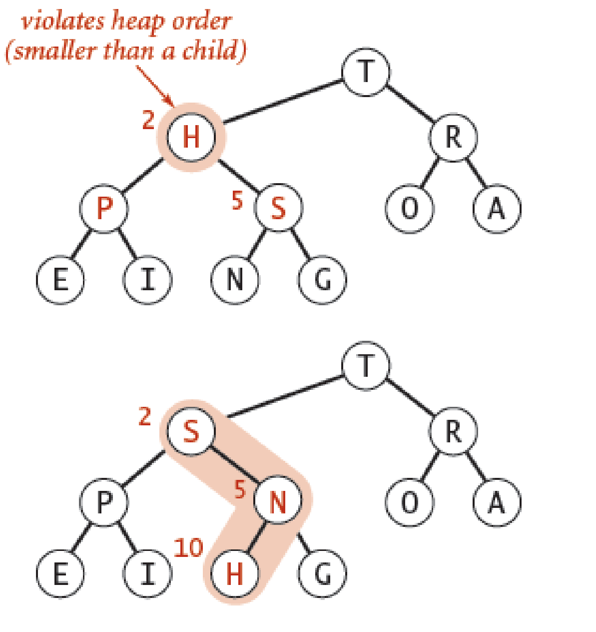
Demotion in a heap: sink

- 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.

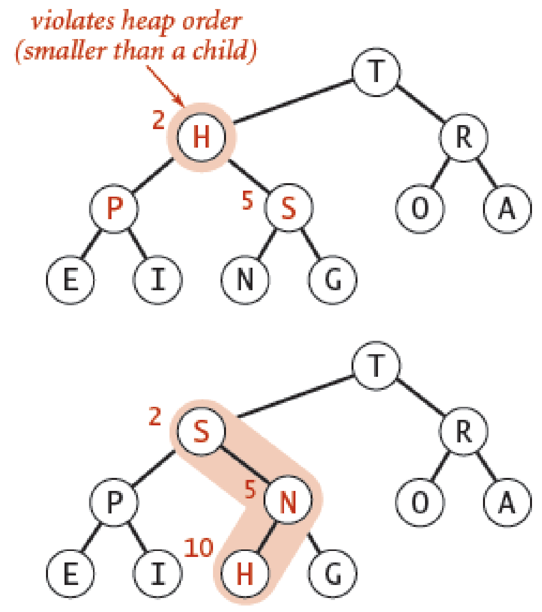
Why not smaller child?

```
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++;
```

```
}
```

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

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.)



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.

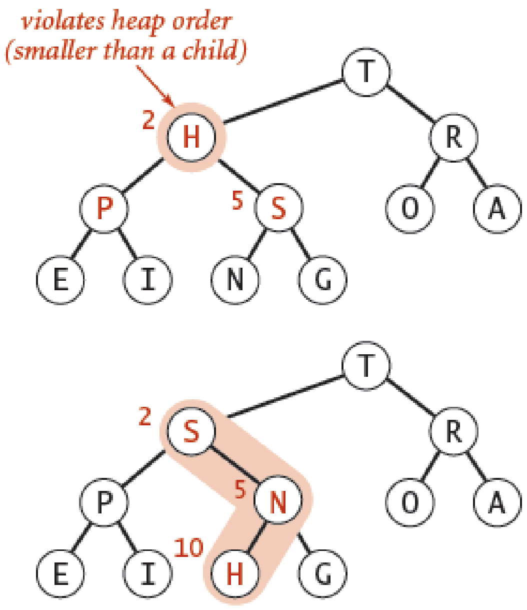
Why not smaller child?

```
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 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)

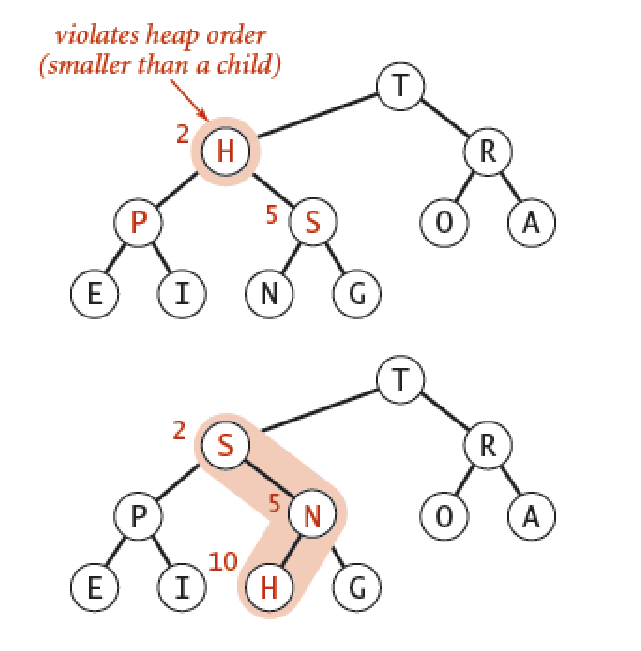
Demotion in a heap: sink

- 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;
    swap(h, k, j);
    set k to the next node (which is j.)
  }
}
```

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



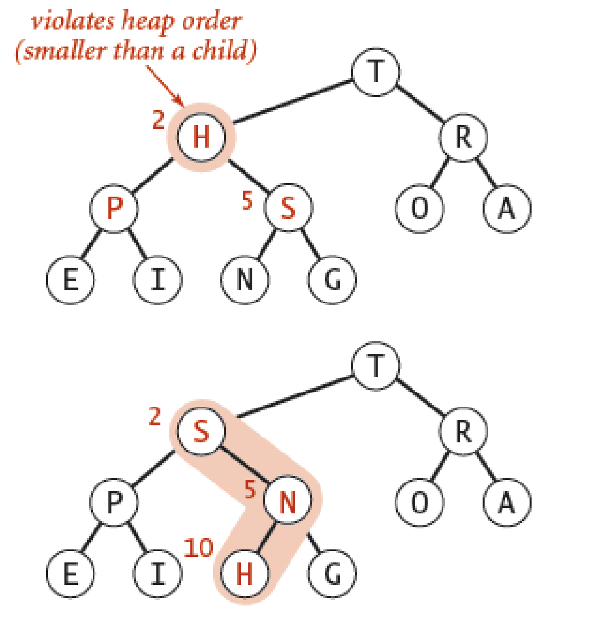
Top-down reheapify (sink)

Demotion in a heap: sink

- Parent's key becomes **smaller** than one (or both) of its children's.
- To eliminate the violation: Why not smaller child?
  - Swap key in parent with key in **larger** child
  - Repeat until heap order restored.

```
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

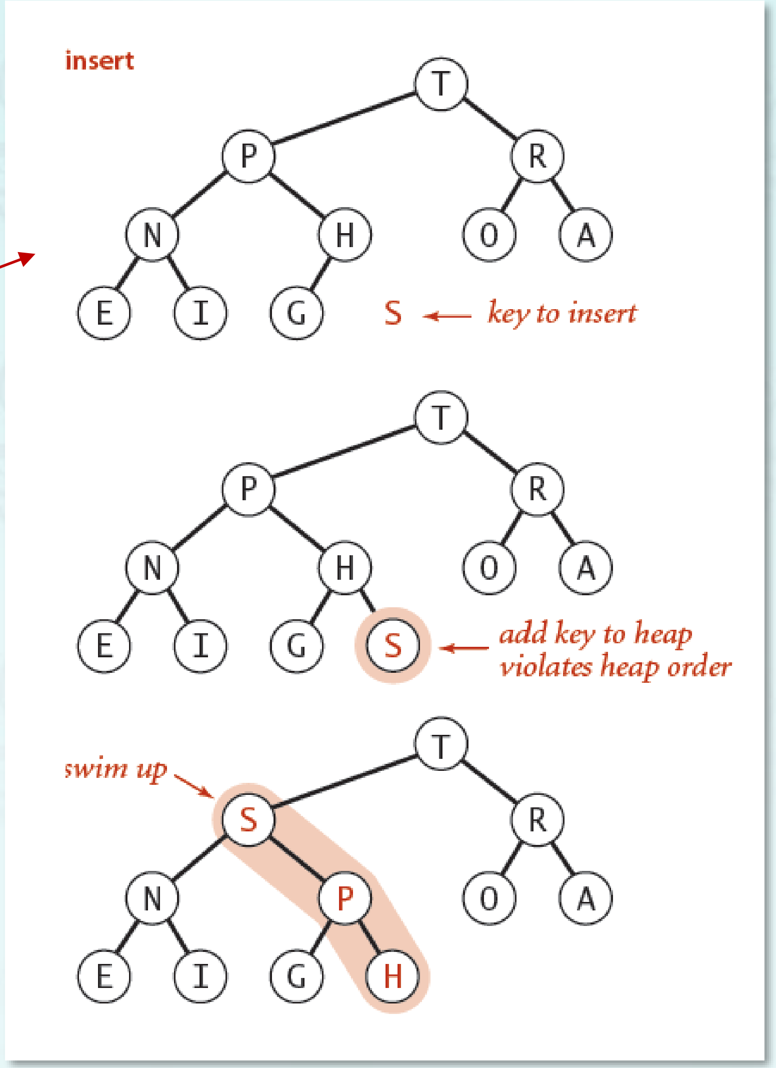
- Insert: Add node at end, then **swim** it up.
  - Cost: At most 1 + log N compares.

What is N now?

Insert

Step 1

Step 2



heap coding

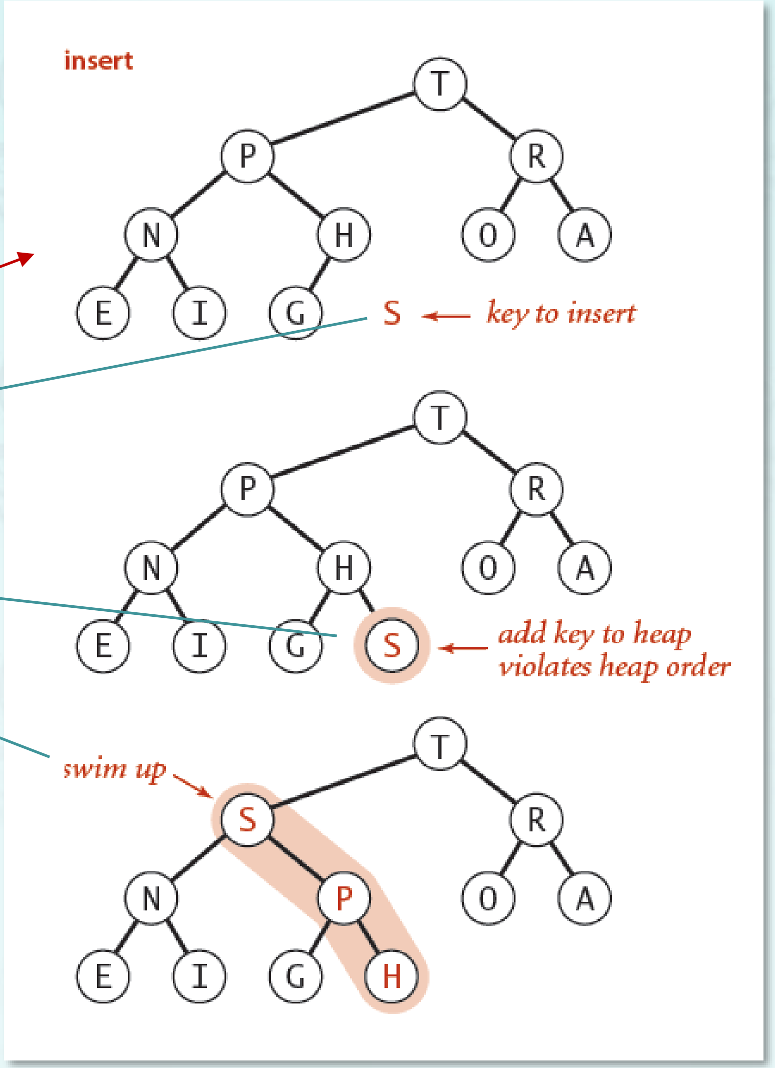
Insertion in a heap:

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

What is N now?

```
void insert(heap h, int key)
{

}
```

```
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
   //
};
using heap = *Heap;
```



heap coding

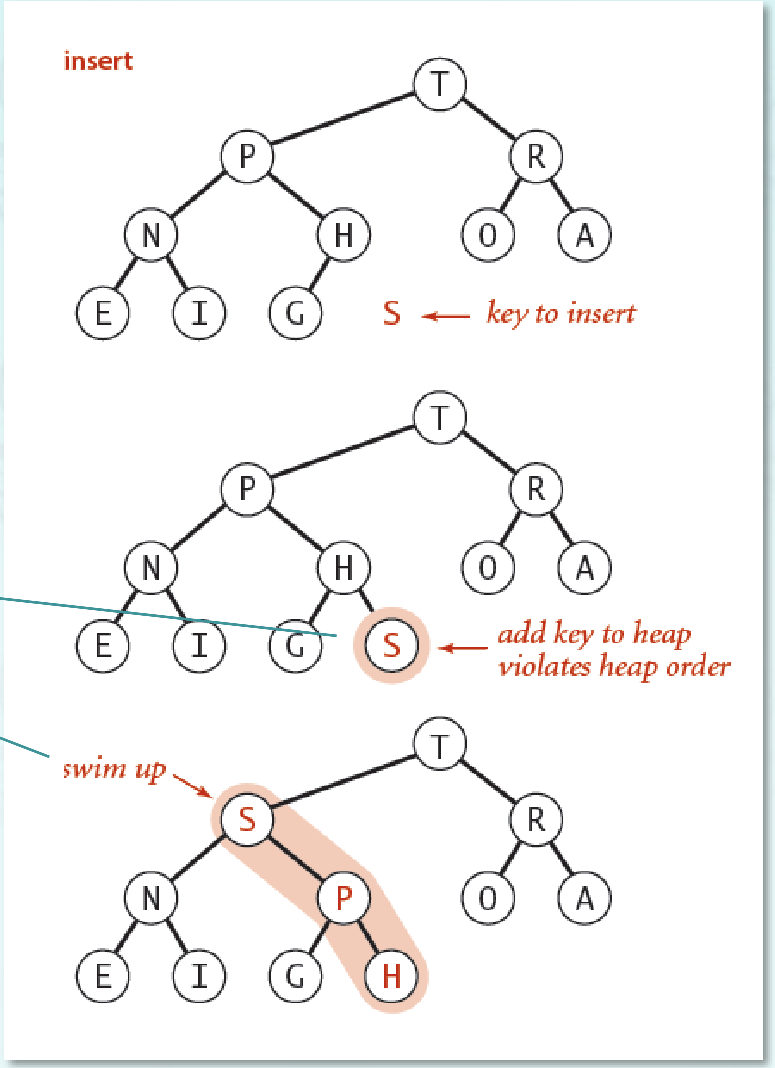
Insertion in a heap:

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

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

}
```

```
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
  //
};
using heap = *Heap;
```

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



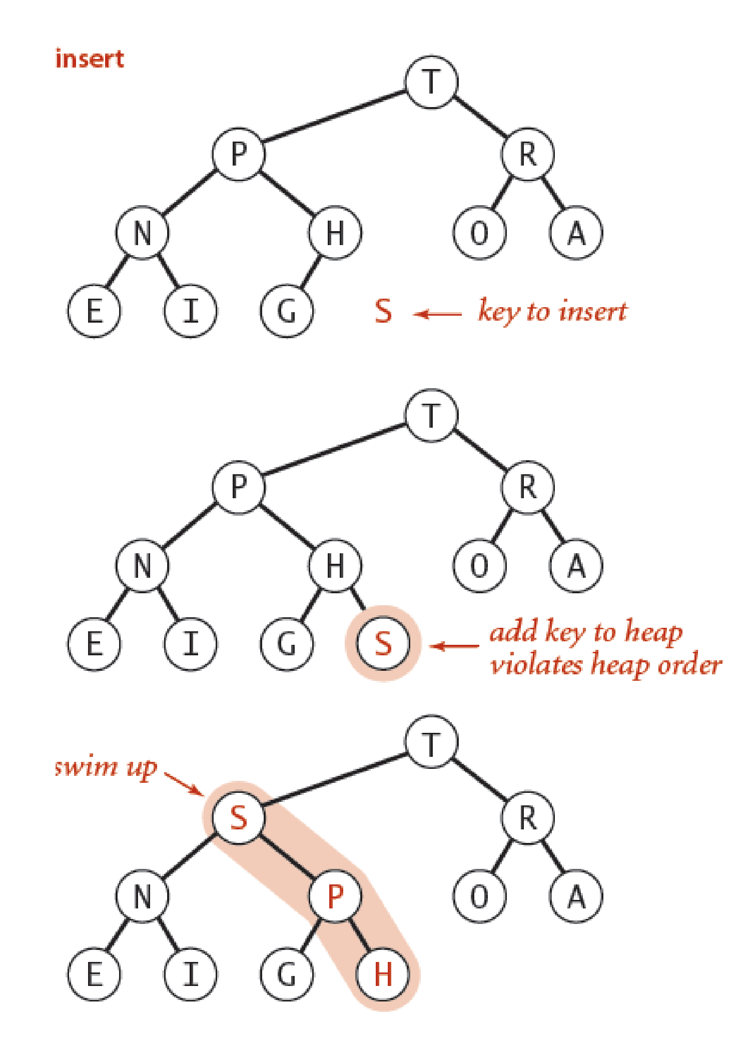
heap coding

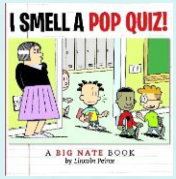
Insertion in a heap:

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

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

```
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
   //
};
using heap = *Heap;
```

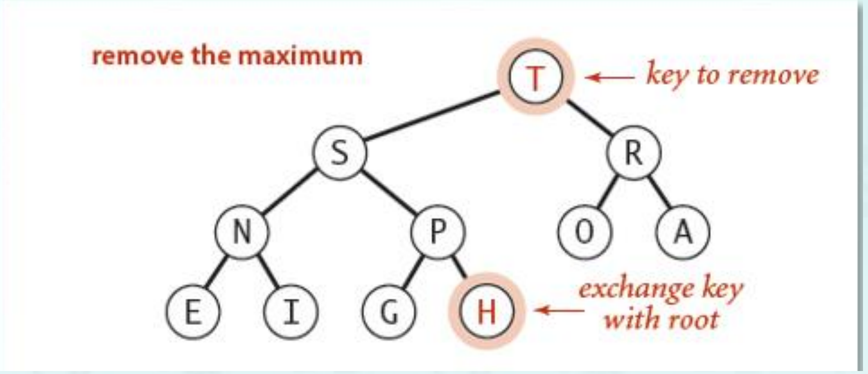




heap coding

(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)
```

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.

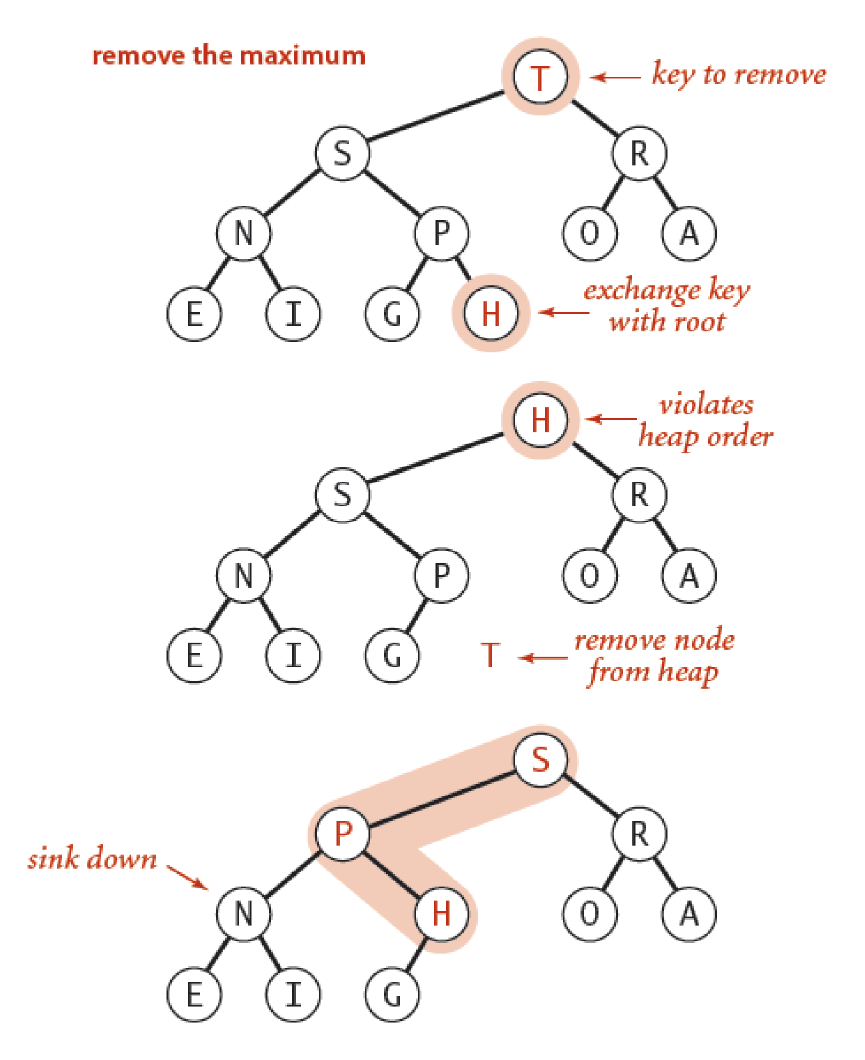
```
void delete(heap h) {

}
```

```
    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:

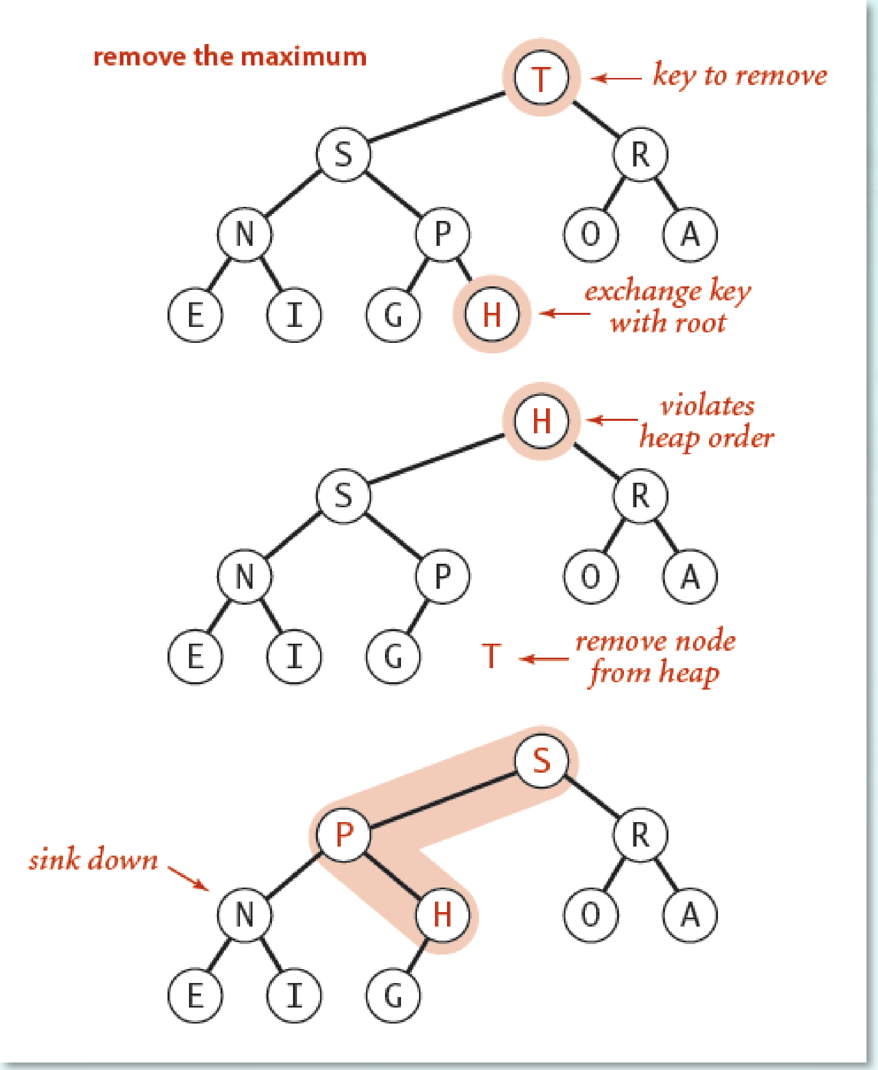
- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

```
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:

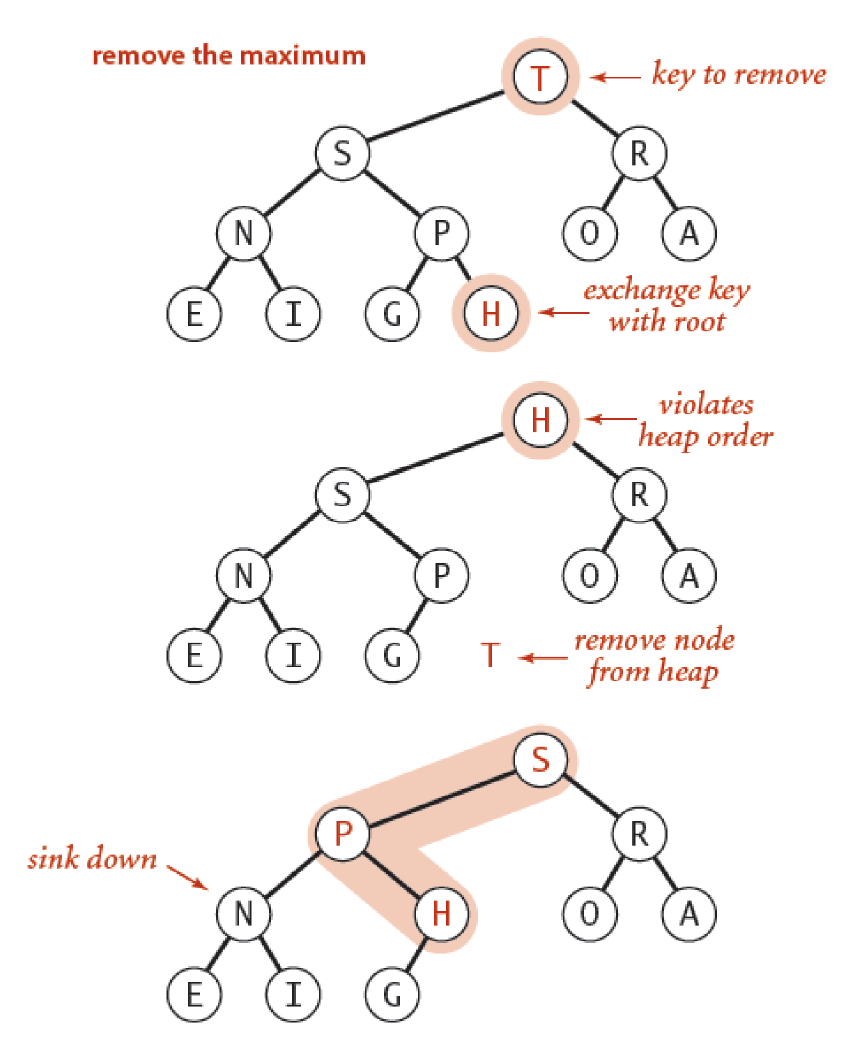
- **Delete root:** Swap root with node at end, then sink it down.
- **Cost:** At most 2 log N compares.

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

}
```

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

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

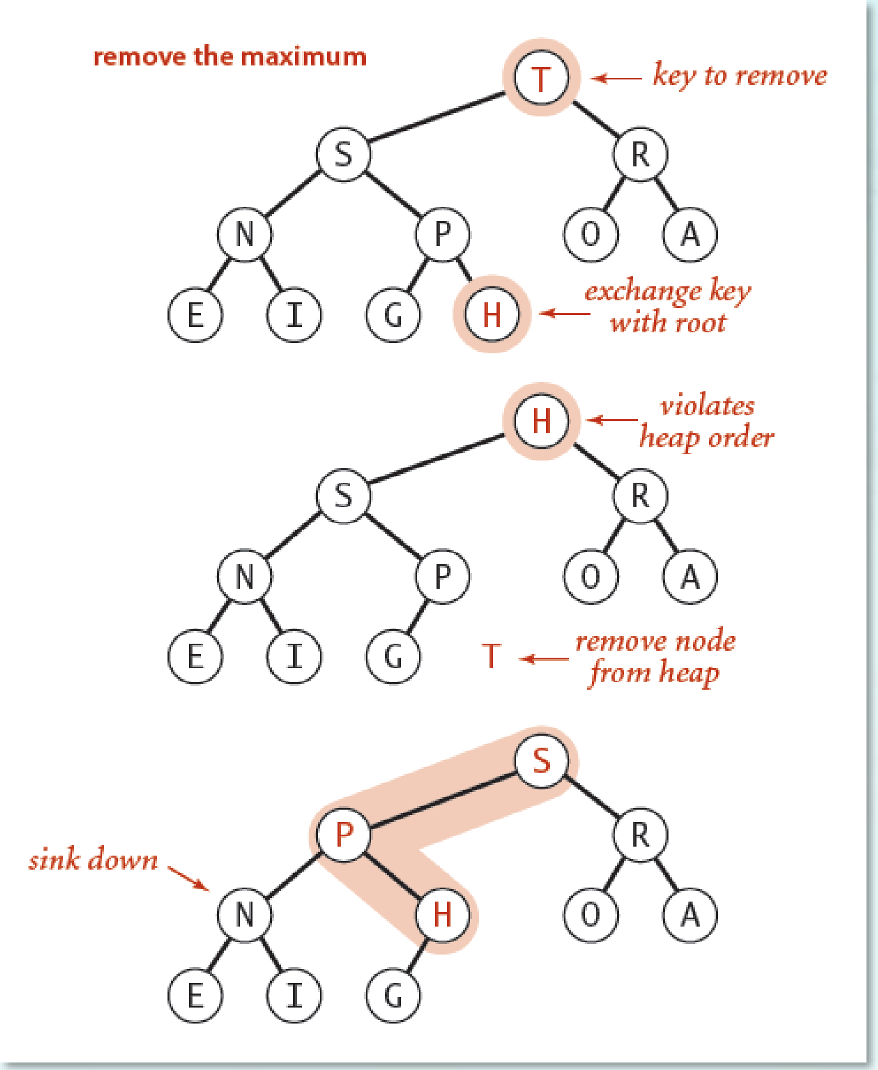
```
void swap(heap h, int p, int c)
```



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

```
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

```
// 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

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

```
void swap(heap hp, int i, int j) {
    int 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

```
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";

}
```

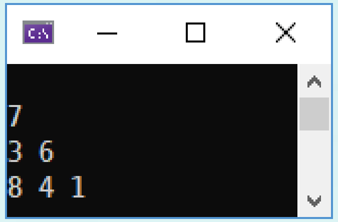
Heap Coding

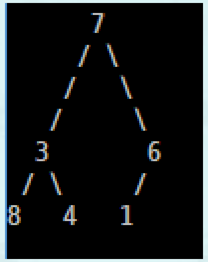
newCBT()	with a given array, instantiate a new complete binary tree its result is neither maxheap nor minheap.
heapify()	make a complete binary tree into a max/minheap
heapsort()	use max/min-heap to sort elements in heap
heapprint()	convert heap to tree for display purpose only

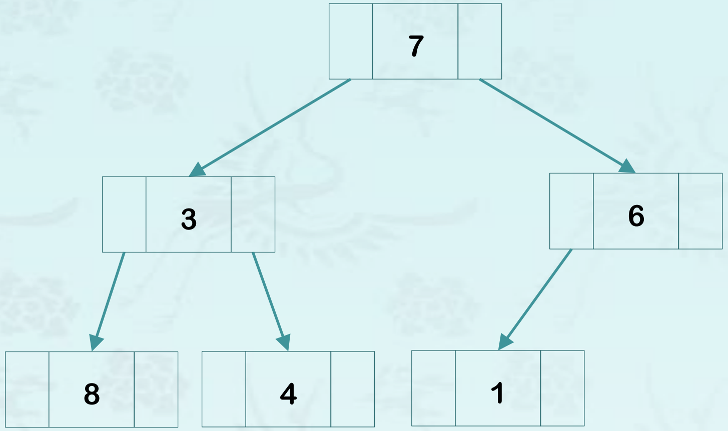
newCBT() – convert an int array to CBT

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

    heap p = new Heap{ capa };

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

heapprint() – convert heap to tree using queue



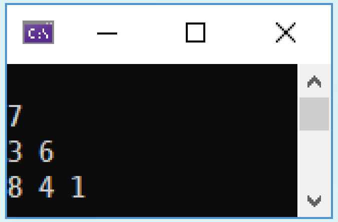





```
struct Heap {
  int *nodes;
  int capacity;
  int N;
  bool (*comp)(Heap*, int, int);
  Heap(int capa = 2) {
    capacity = capa;
    nodes = new int[capacity];
    N = 0;
    comp = nullptr;
  };
  ~Heap() {};
};
using heap = Heap*;
```

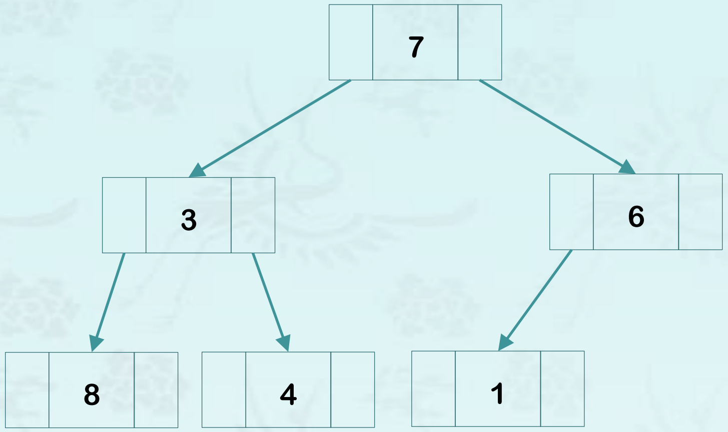
```
struct TreeNode {
  int key;
  TreeNode  *left;
  TreeNode  *right;
  TreeNode(const int k = 0,
    TreeNode* l = nullptr,
    TreeNode* r = nullptr) {
    key = k; left = l;right =r;
  }
  ~TreeNode() {}
};
using tree = TreeNode*;
```

heapprint() – convert heap to tree using queue



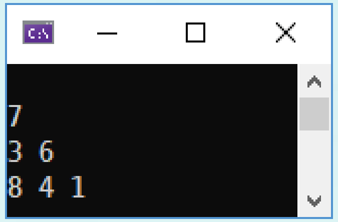
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

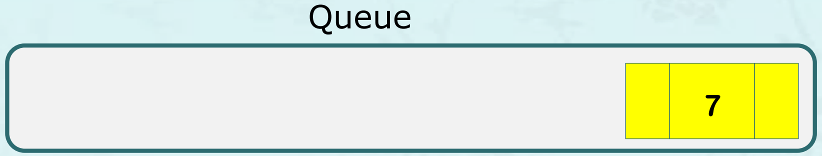
`hp->nodes[]`   `hp->N`



1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
   else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue



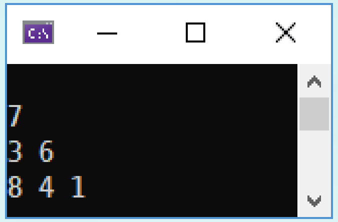


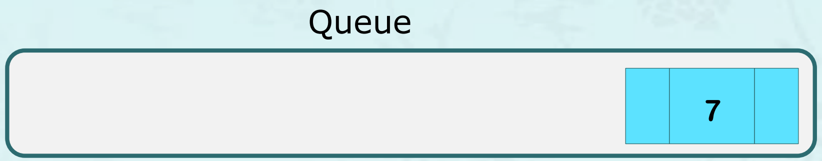
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

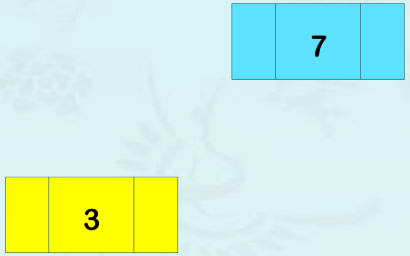
`hp->nodes[]` `hp->N`

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





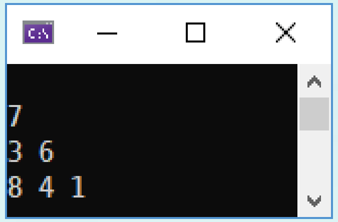


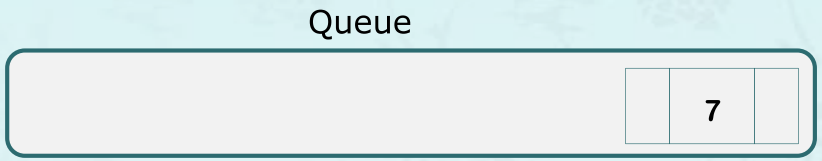
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

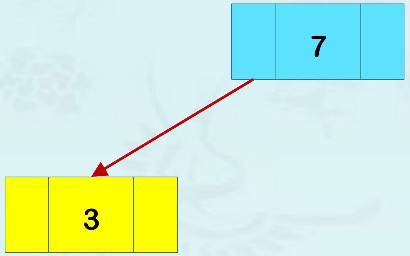
hp->nodes[] hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





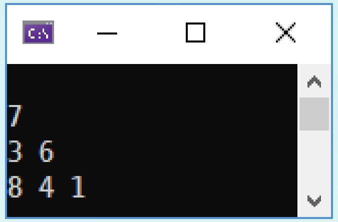


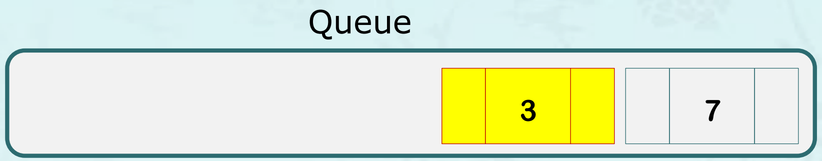
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

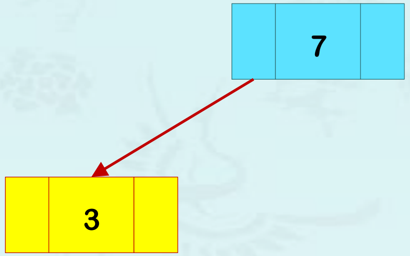
hp->nodes[]　　　　hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. **If the left of the tree node doesn't exist,**
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





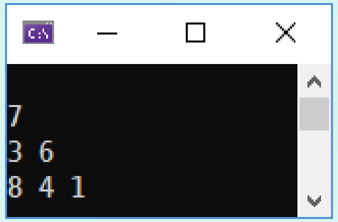


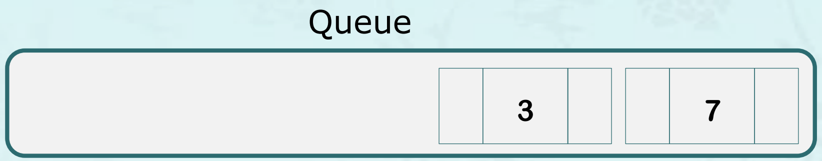
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

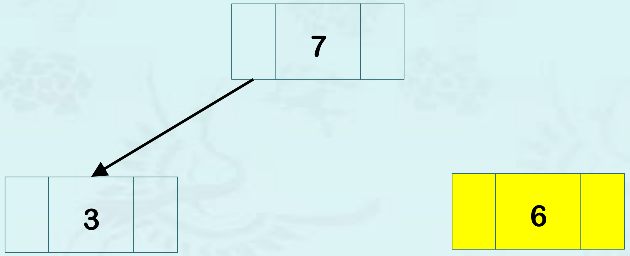
`hp->nodes[]` `hp->N`

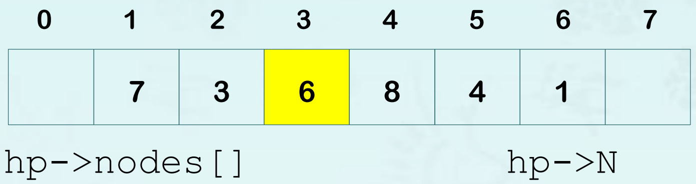
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
   else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue



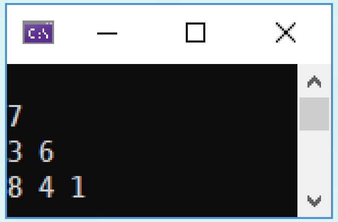


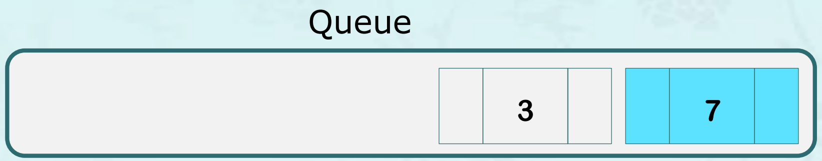


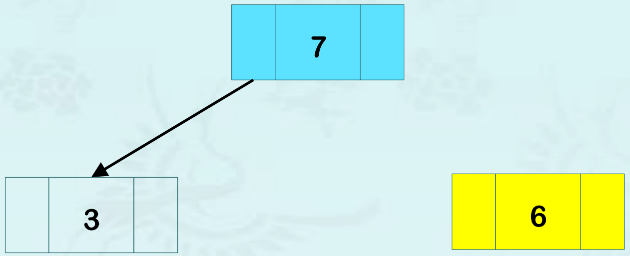


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
     set the new node to the left of the tree node.
     else if the right of this tree node doesn't exist,
     set the new node to the right of the tree node.
   - D. If this tree node is full, pop (or dequeu) it.
   - E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





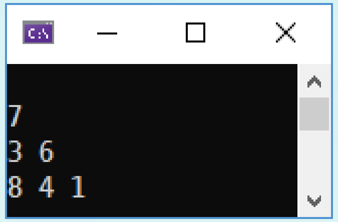


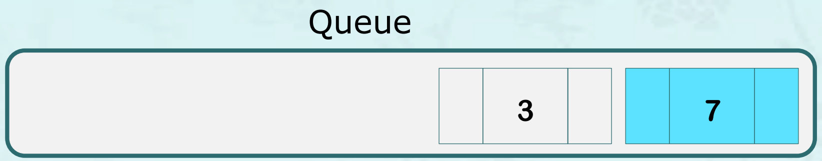
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

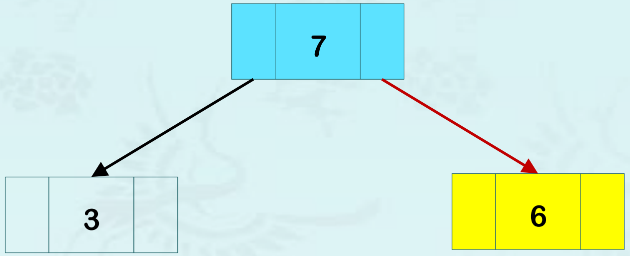
hp->nodes[]　　　　hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





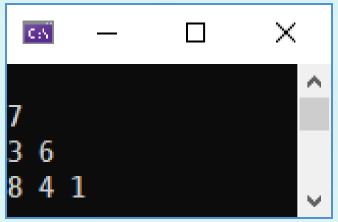


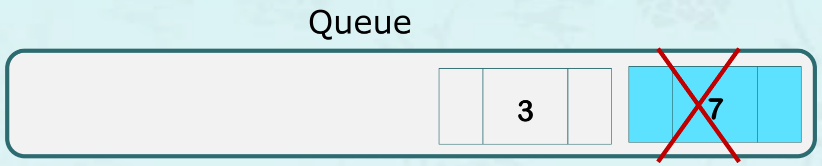
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

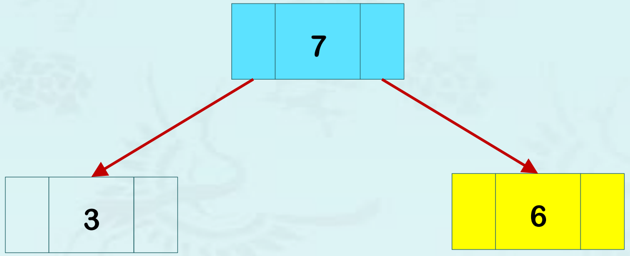
hp->nodes[]　　hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





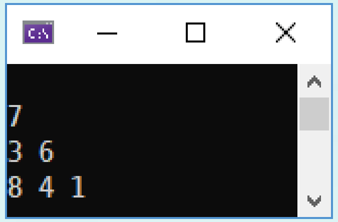


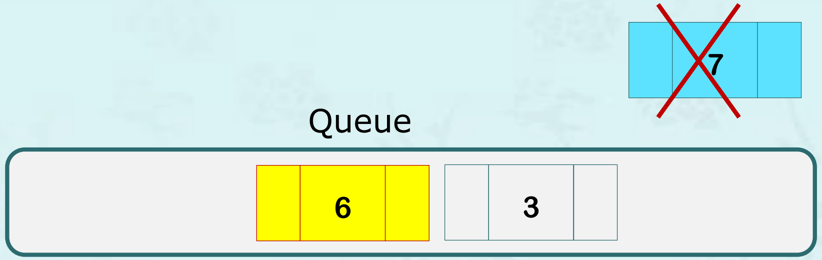
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

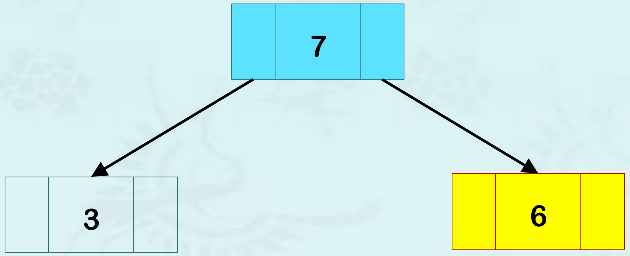
hp->nodes[] hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





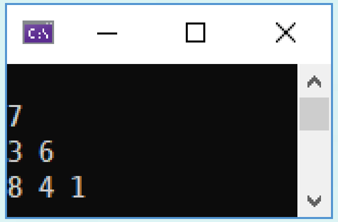


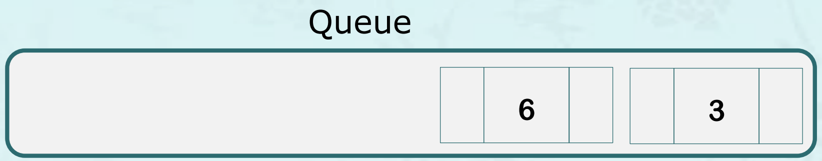
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

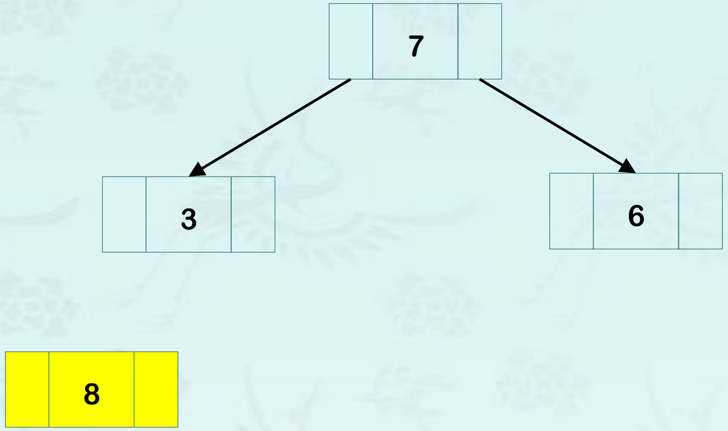
hp->nodes[] hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





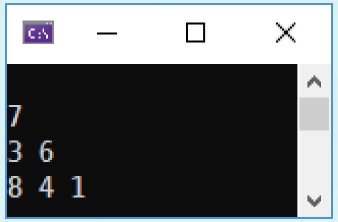


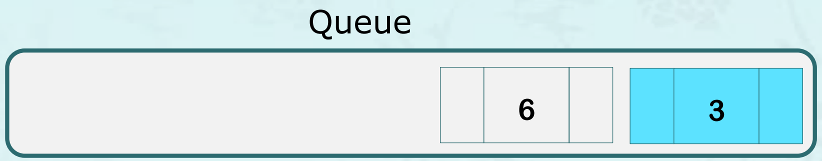
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

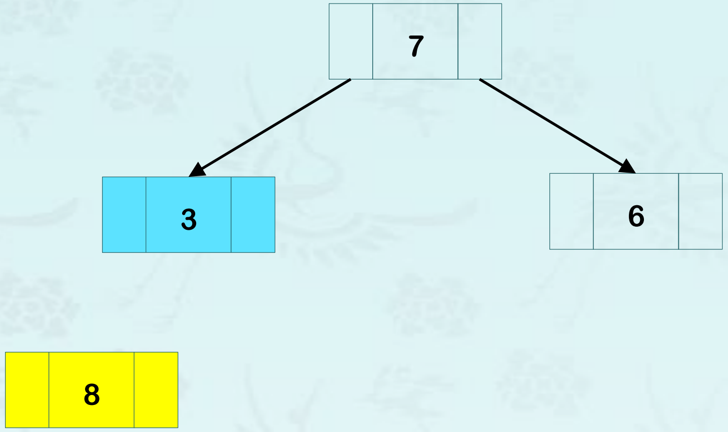
hp->nodes[]   hp->N

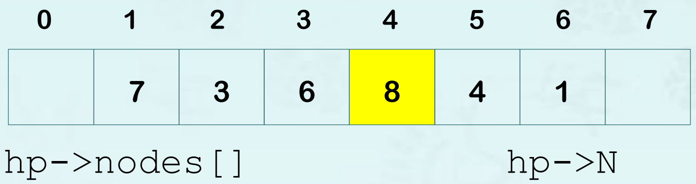
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue



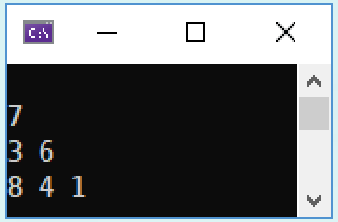


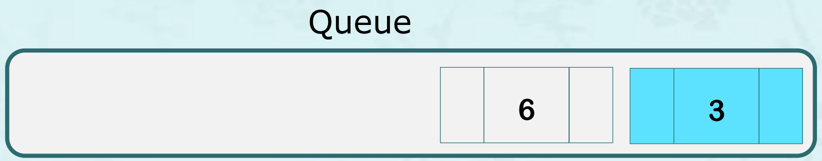


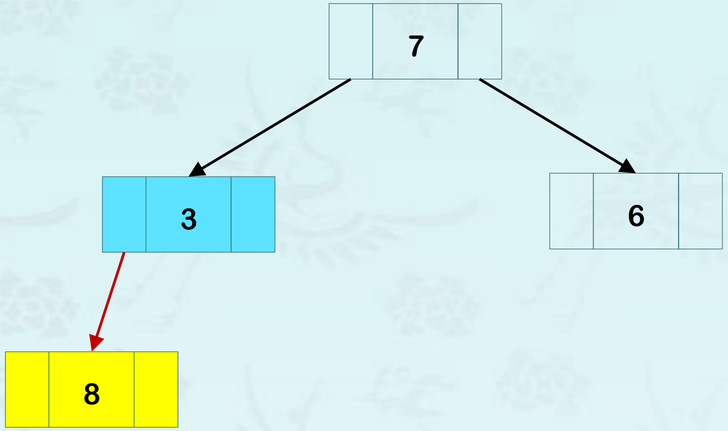


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





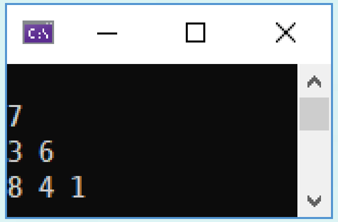


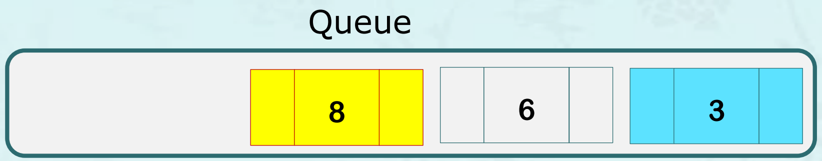
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

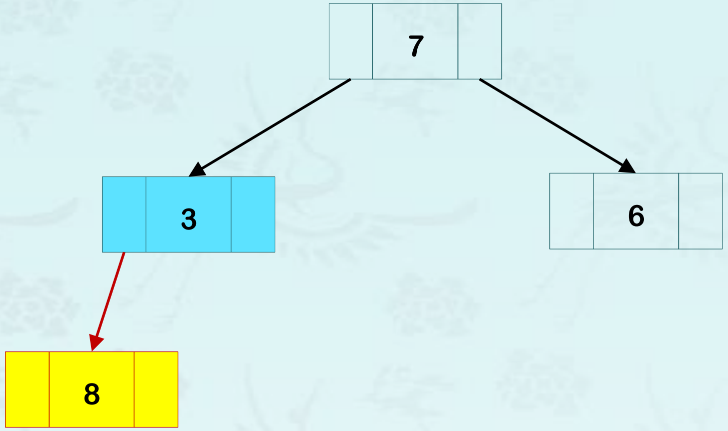
hp->nodes[]   hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. **If the left of the tree node doesn't exist,**
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue





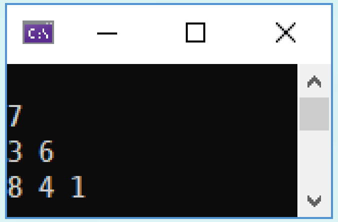


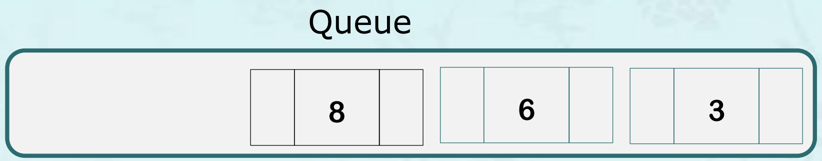
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

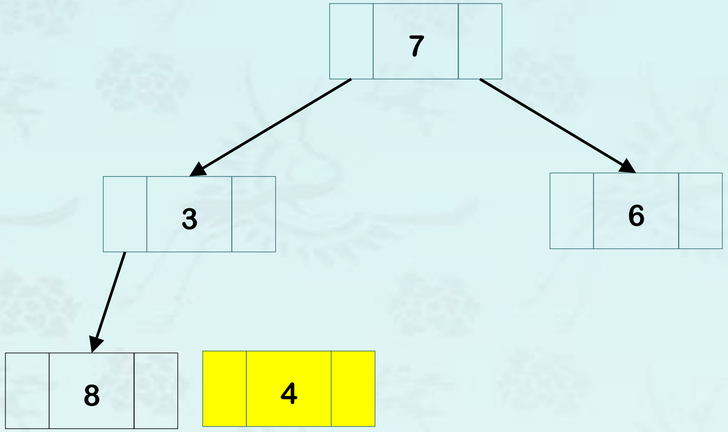
hp->nodes[]   hp->N

1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
     set the new node to the left of the tree node.
     else if the right of this tree node doesn't exist,
     set the new node to the right of the tree node.
   - D. If this tree node is full, pop (or dequeu) it.
   - E. **enqueue the new node (to add children later if any).**
4. treeprint(root)

heapprint() – convert heap to tree using queue

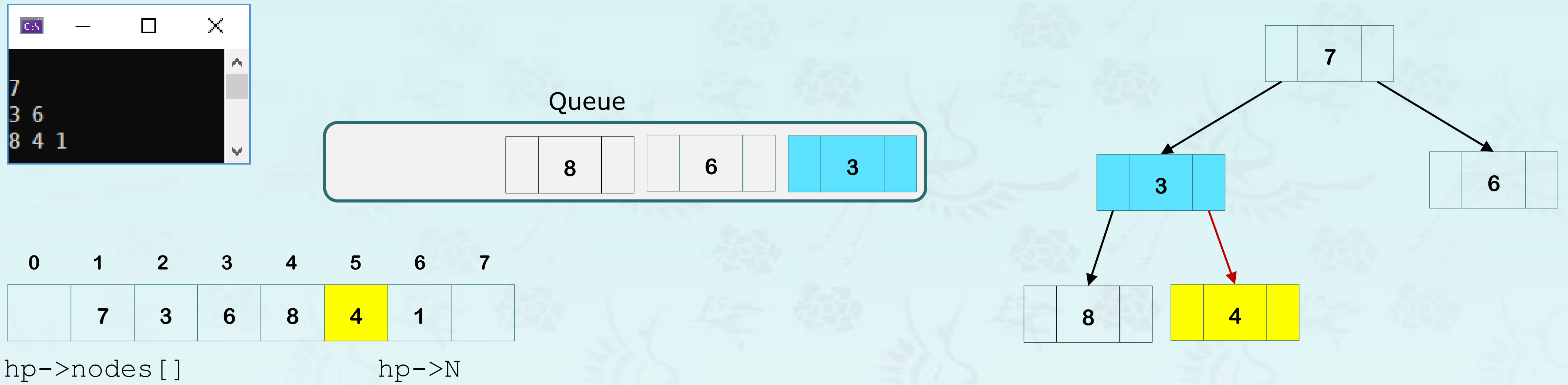






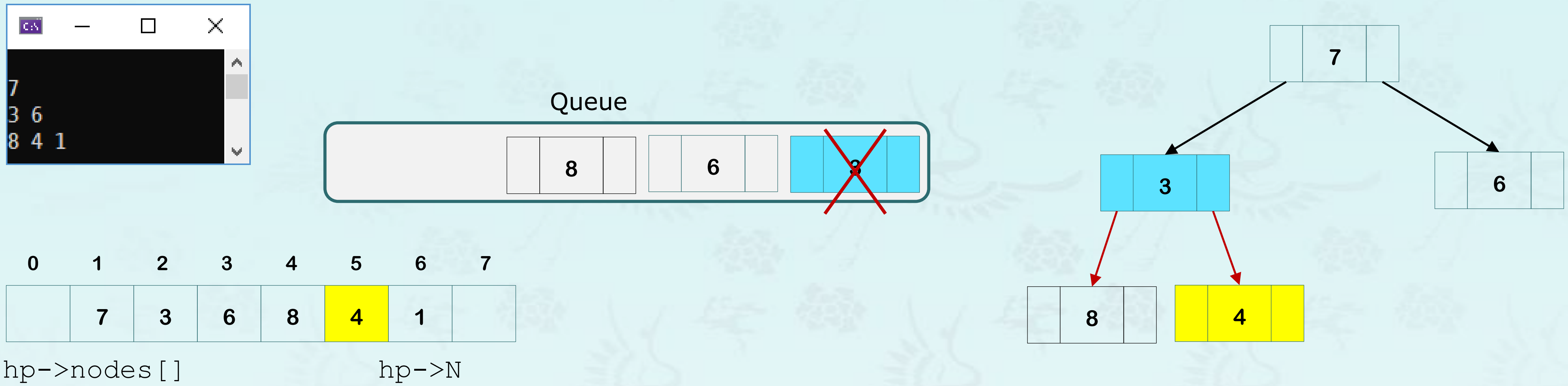
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue



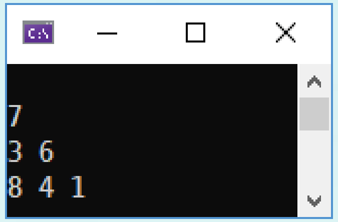
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

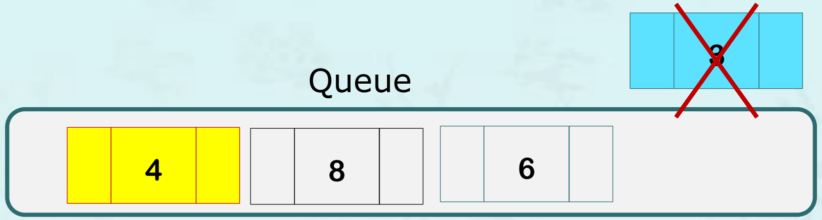
heapprint() – convert heap to tree using queue

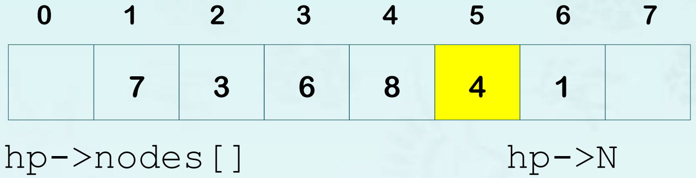


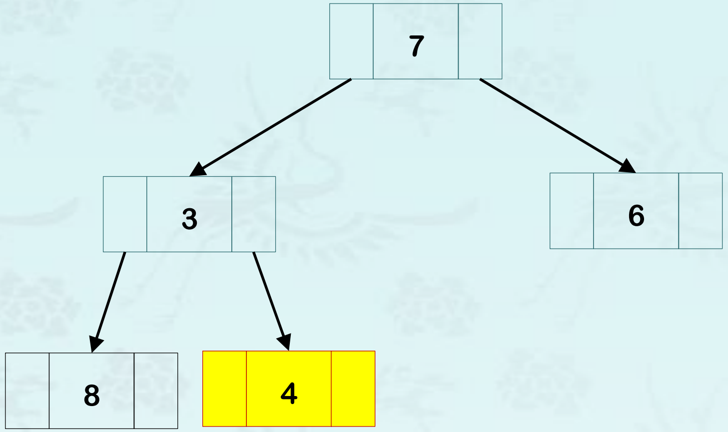
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. **If this tree node is full, pop (or dequeu) it.**
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue



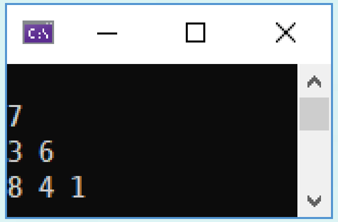


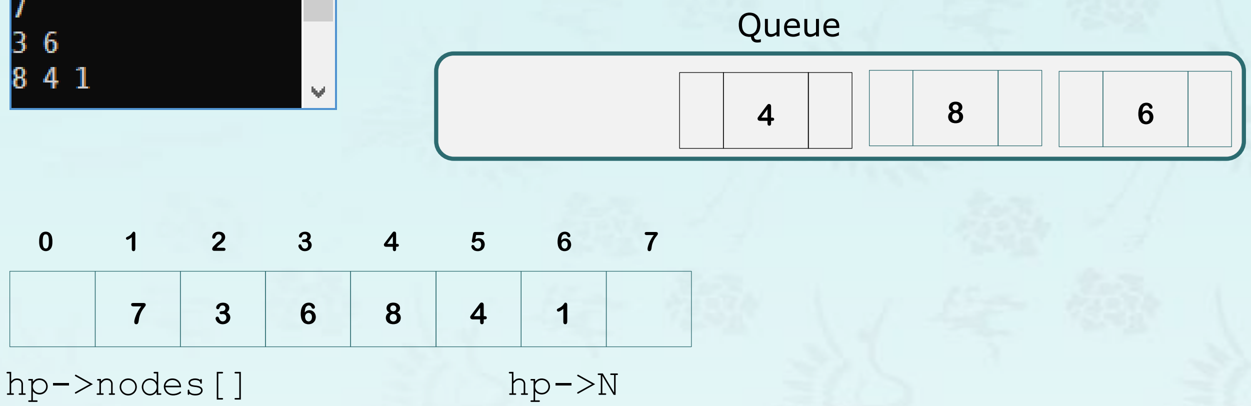


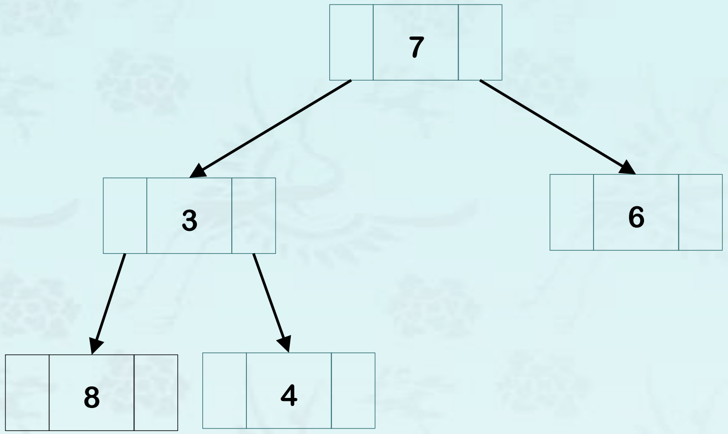


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
     set the new node to the left of the tree node.
     else if the right of this tree node doesn't exist,
     set the new node to the right of the tree node.
   - D. If this tree node is full, pop (or dequeu) it.
   - **E. enqueue the new node (to add children later if any).**
4. treeprint(root)

heapprint() – convert heap to tree using queue

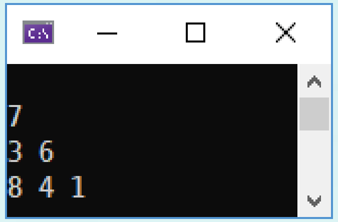


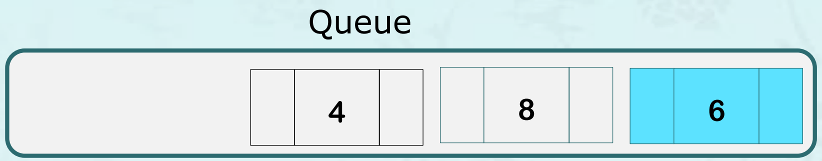


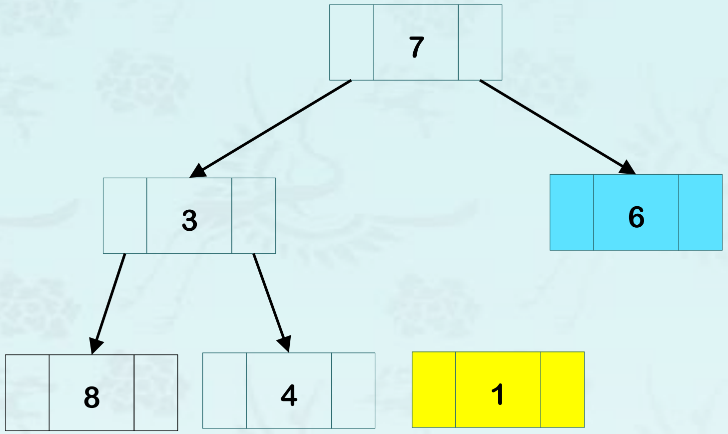


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
     set the new node to the left of the tree node.
     else if the right of this tree node doesn't exist,
     set the new node to the right of the tree node.
   - D. If this tree node is full, pop (or dequeu) it.
   - E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue

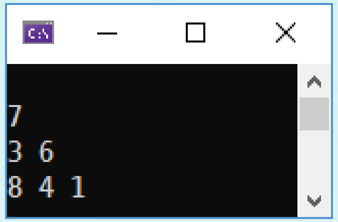


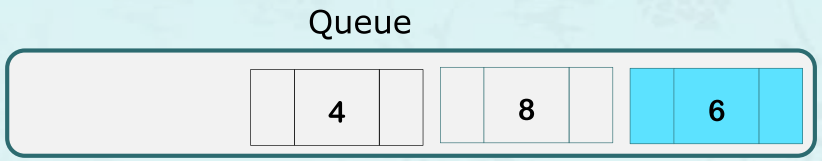




1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
     set the new node to the left of the tree node.
     else if the right of this tree node doesn't exist,
     set the new node to the right of the tree node.
   - D. If this tree node is full, pop (or dequeu) it.
   - E. enqueue the new node (to add children later if any).
4. treeprint(root)

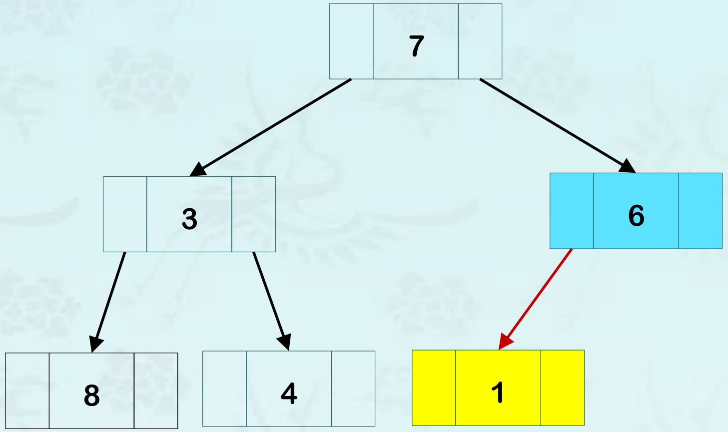
heapprint() – convert heap to tree using queue





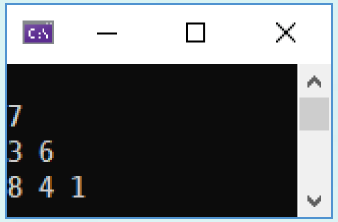
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

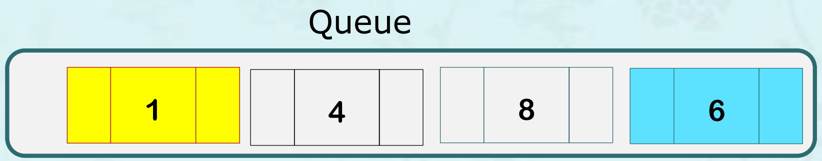
hp->nodes[] hp->N

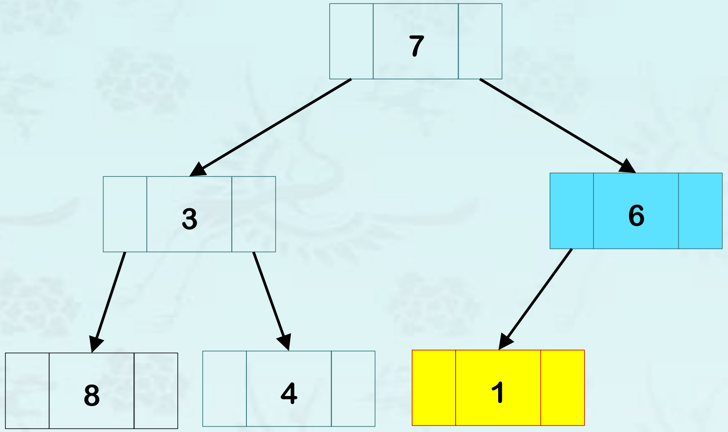


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. **If the left of the tree node doesn't exist,**
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using **queue**

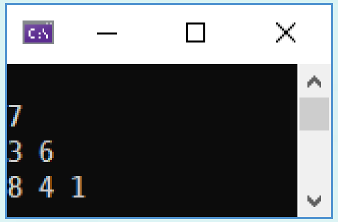


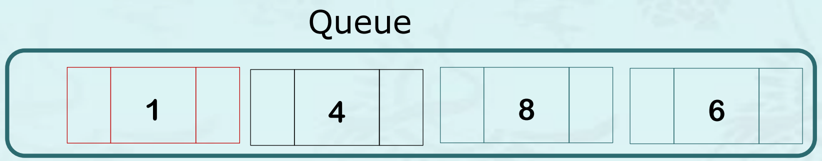


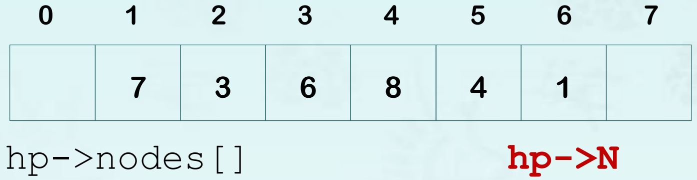


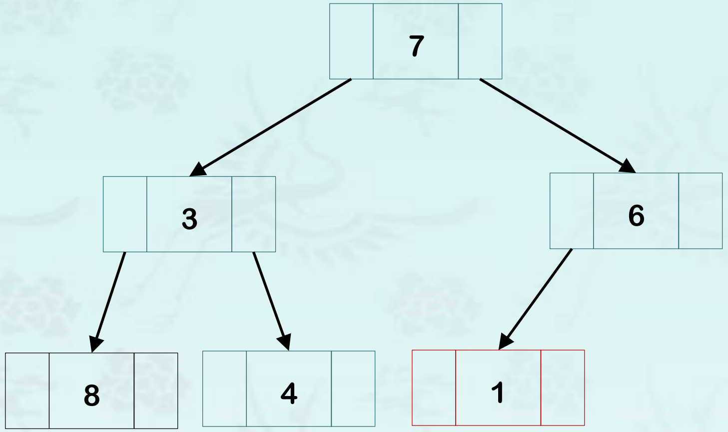
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using queue









1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. **Loop through from the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set the new node to the left of the tree node.
      else if the right of this tree node doesn't exist,
      set the new node to the right of the tree node.
   D. If this tree node is full, pop (or dequeu) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – convert heap to tree using recursion

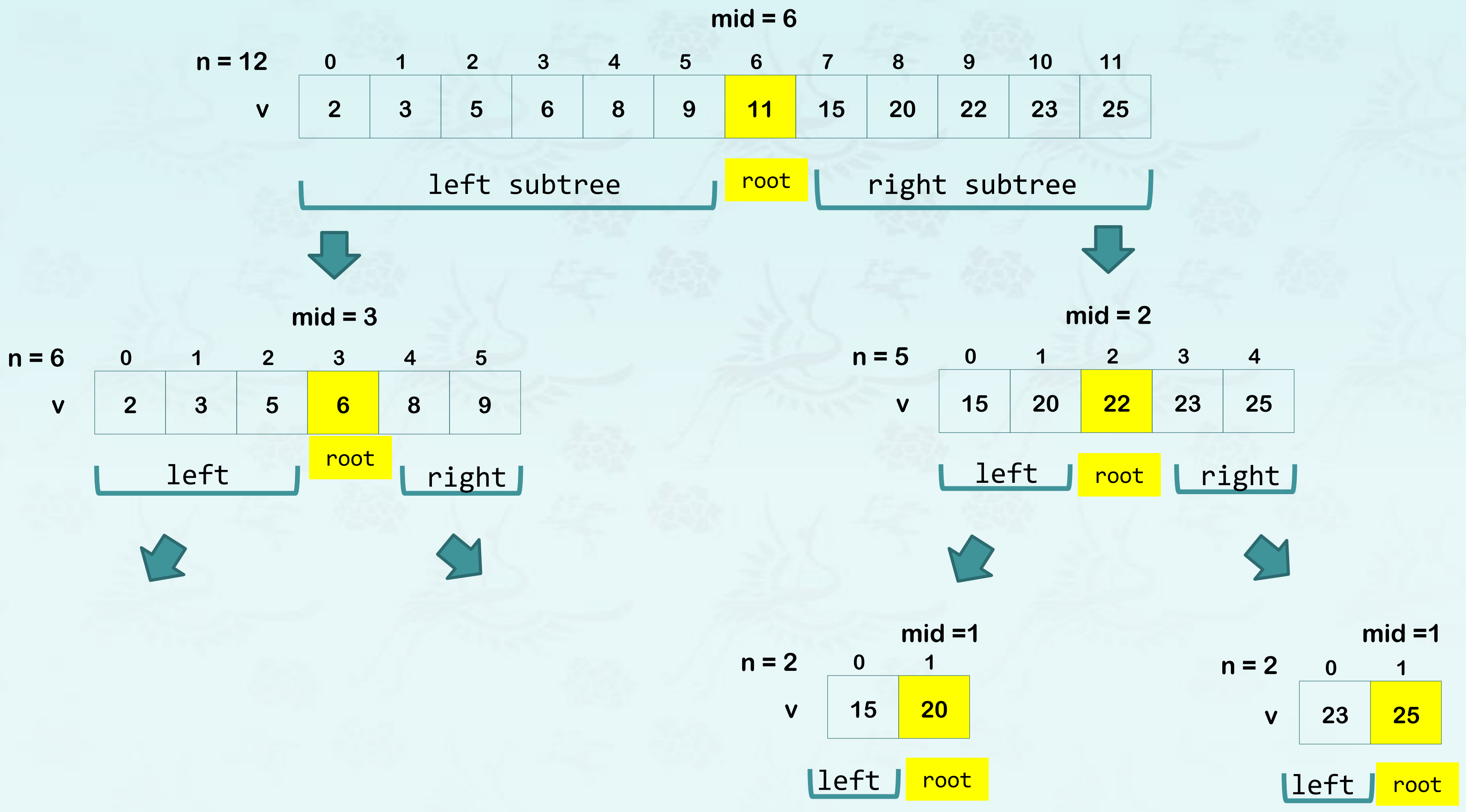
Building AVL tree from BST in O(n) – Review

mid = 6

n = 12	0	1	2	3	4	5	6	7	8	9	10	11
v	2	3	5	6	8	9	11	15	20	22	23	25

$\underbrace{\text{left subtree}}$ root $\underbrace{\text{right subtree}}$

Building AVL tree from BST in O(n) – Review



Building AVL tree from BST in O(n) – Review

```
// rebuilds an AVL tree with a list of keys sorted.
// v – an array of keys sorted, n – the array size

tree buildAVL(int* v, int n) {
  if (n <= 0) return nullptr;

  int mid = n / 2;



  return root;
}
```

```
tree rebalanceTree(tree root) {
  if (root == nullptr) return nullptr;
  vector<int> v;        // get keys sorted
  inorder(root, v);     // O(n)
  clear(root);

  tree new_root = buildAVL(v.data(), v.size()); // O(n)
  return new_root;
}
```

heapprint() – convert heap to tree using recursion

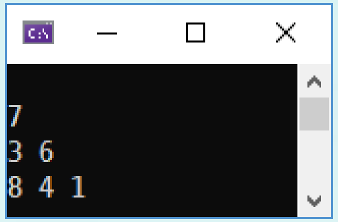
Create a recursive function that creates a binary tree from an int array. This function takes an int array, starting index, and size of the array and returns the root as shown below:

tree buildBT(int *nodes, int i, int n) {

1. If **i > n**, return nulltptr – terminate condition
2. Create the **tree (root) node** with **nodes[i]**).
   A. Invoke buildBT() for all its left children (or **i * 2**).
      Set its return to the left child of the root.
   B. Invoke buildBT() for all its right children (or **i * 2 + 1**).
      Set its return to the right child of the root.
3. return root

}

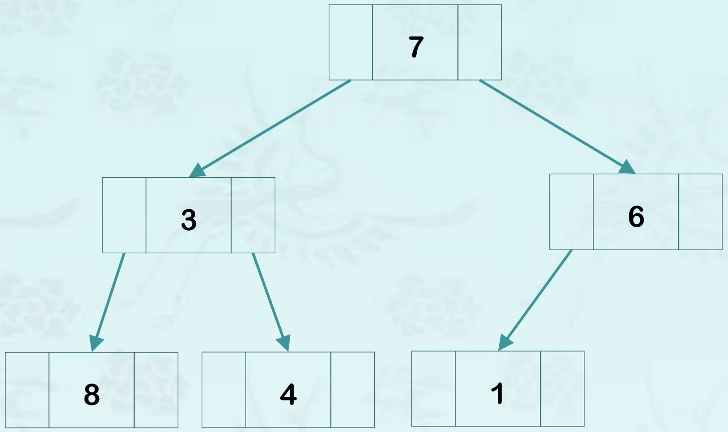
heapprint() – convert heap to tree using recursion

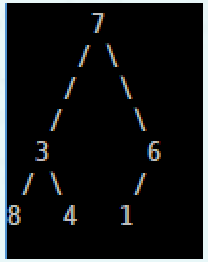


0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N

```
tree buildBT(int *nodes, int i, int n) {



    return  root;
}
```





heapprint() – convert heap to tree using recursion

```
7
3 6
8 4 1
```

0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

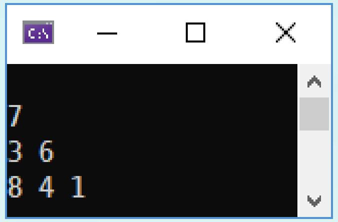
hp->nodes[]   hp->N

```
tree buildBT(int *nodes, int i, int n) {
  if (i > n) return nullptr;

  tree root =
  root->left  =
  root->right =
  return  root;
}
```

```
tree buildBT(*nodes, i=1, n=6) { }
```

```
void heapprint(heap p) {
  if (empty(p)) return;
  tree root = buildBT(p->nodes, 1, size(p));
  treeprint(root);
}
```

heapprint() – convert heap to tree using recursion



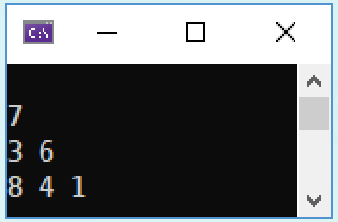
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N

```
tree buildBT(int *nodes, int i, int n) {
  if (i > n) return nullptr;

  tree root =
  root->left  =
  root->right =
  return  root;
}
```

```
tree buildBT(*nodes, i=1, n=6) { }
```

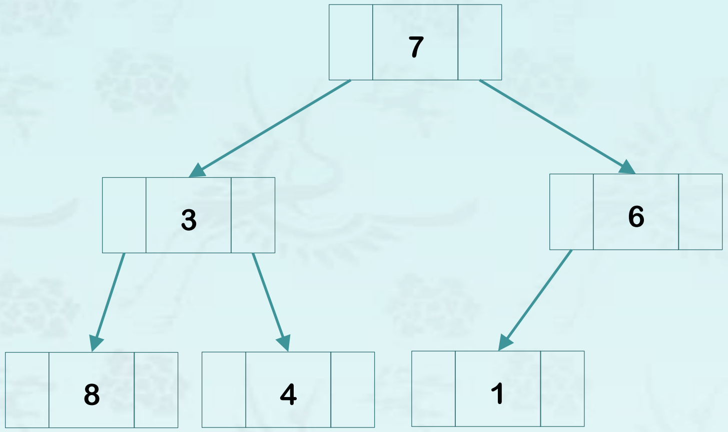
heapprint() – convert heap to tree using recursion

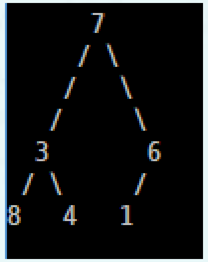


0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N

```
tree buildBT(int *nodes, int i, int n) {
  if (i > n) return nullptr;

  tree root =
  root->left  =
  root->right =
  return  root;
}
```





heap

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