Data Structures
Chapter 5: Heap and Priority Queue

1. Heap & Priority Queue
2. Heapsort
3. **Heap & PQ Coding**

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm202, Handong Global University

자기 아들을 아끼지 아니하시고 우리 모든 사람을 위하여 내주신 이가 어찌 그 아들과 함께 모든 것을 우리에게 주시지 아니하겠느냐 (로마서 8:32)

우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에게는 모든 것이 합력하여 선을 이루느니라 (로마서 8:28)

Heap ADT - heap.h

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

```
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*;
```

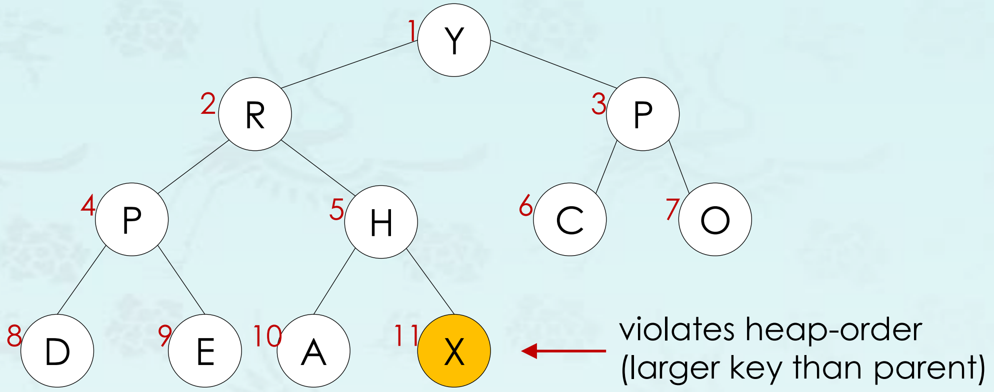
Heap ADT - heap.h

```
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.



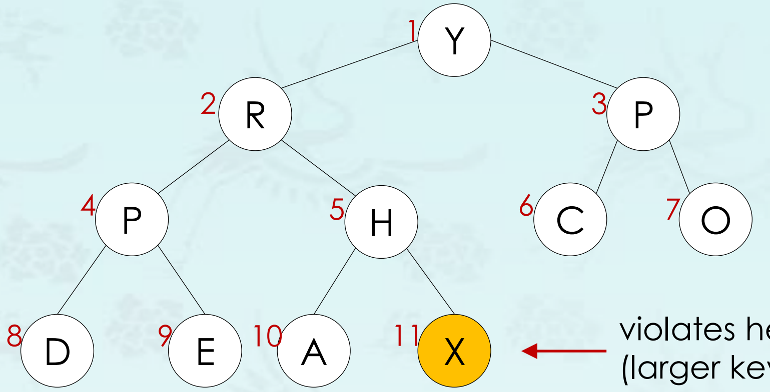
swim up or sink down ?

Promotion in a heap: swim

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

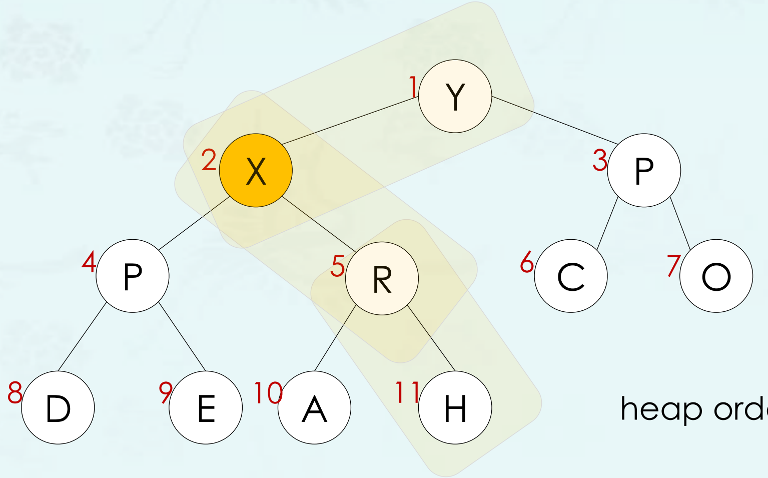
This is a maxheap example.

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



violates heap-order
(larger key than parent)

```
void swap (heap h, int p, int c) {
    int item = h->nodes[p];
    h->nodes[p] = h->nodes[c];
    h->nodes[c] = item;
}  // Inside this swap(), we may use swap() in c++
```

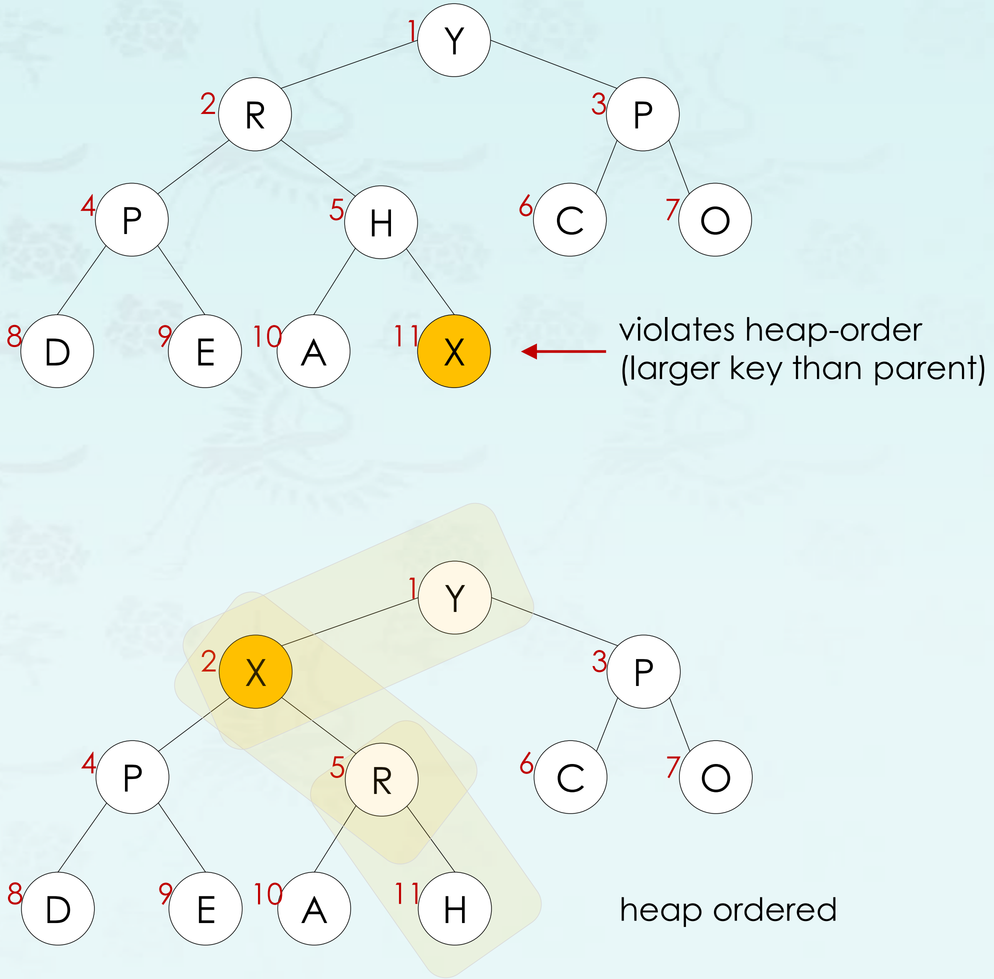


heap ordered

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.




```
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.

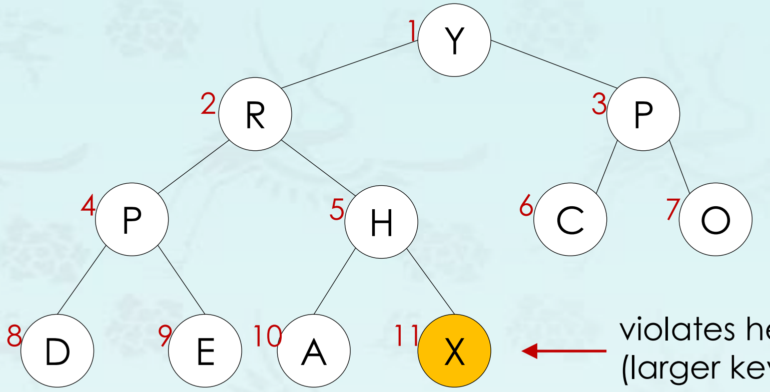
This is a maxheap example.

not reached at root , parent is less than its child(k)

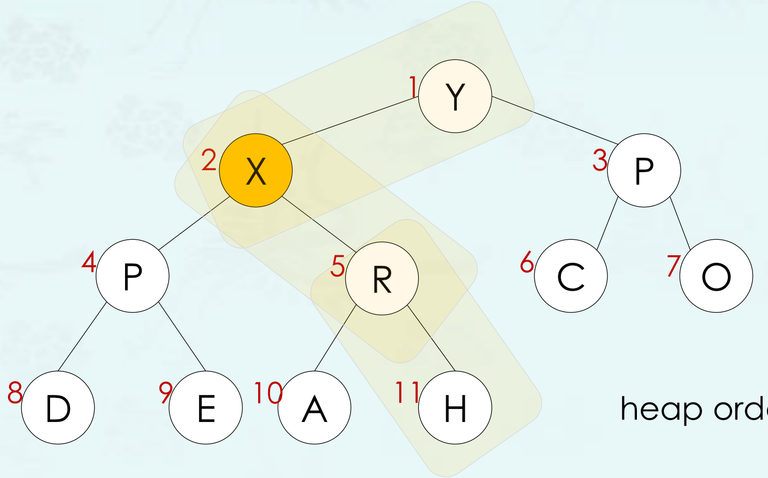
```
{
  while (k > 1 && less(h, k / 2, k))
  {
    swap(h, k / 2, k);
    k = k / 2;
  }
}
```

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

move up one level



violates heap-order (larger key than parent)

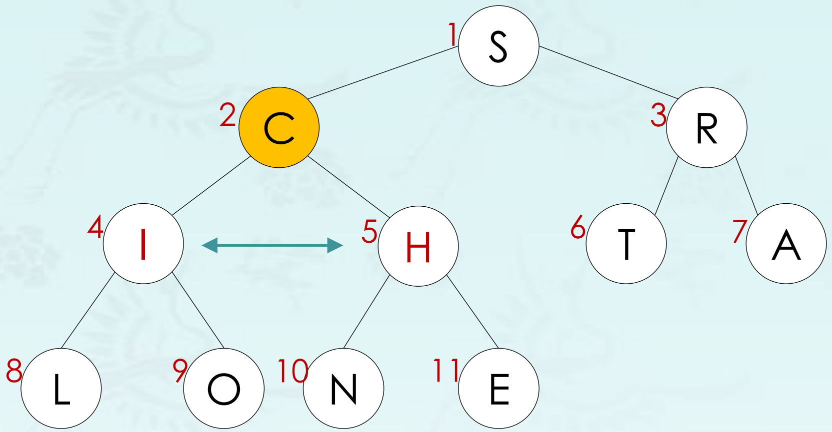


heap ordered

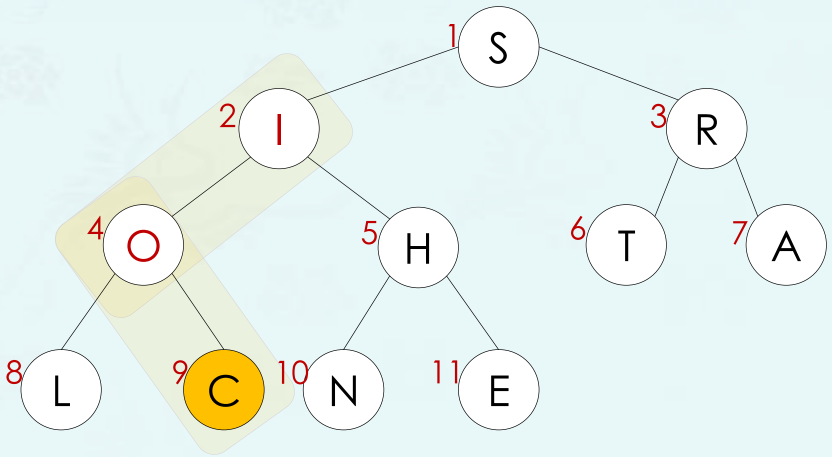
Demotion in a heap: sink

- 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.

Top-down reheapify (sink)



swim up or sink down ?



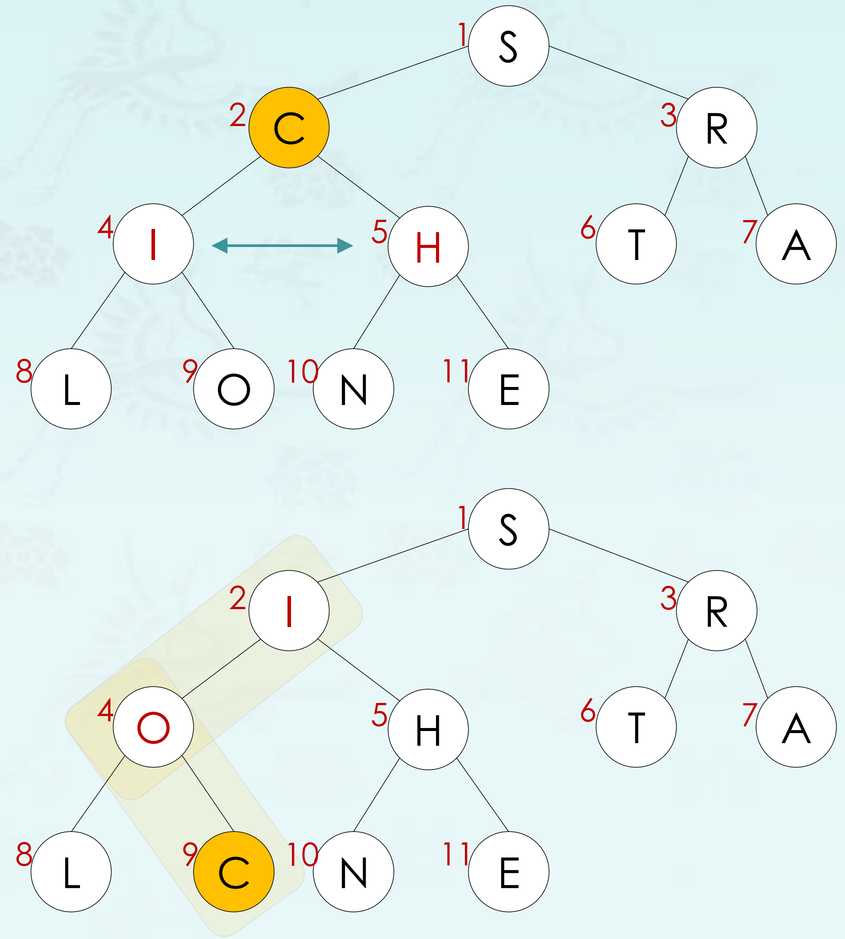
Demotion in a heap: sink

- 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)

```
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.
   }
}
```

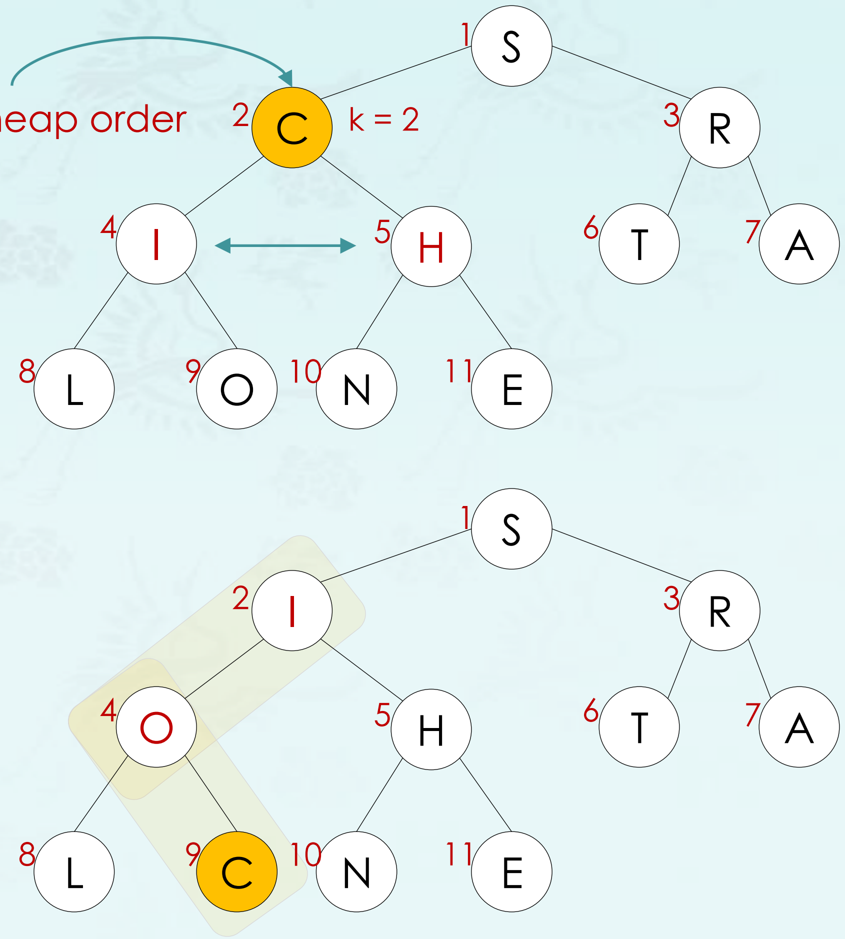


Demotion in a heap: sink

- 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)




```
void
{
    while (2 * k <= h->N)
    {
        int j = 2 * k;
                                  j++;
    }
}
```

while (k's child not reached the last)

find the larger child of k

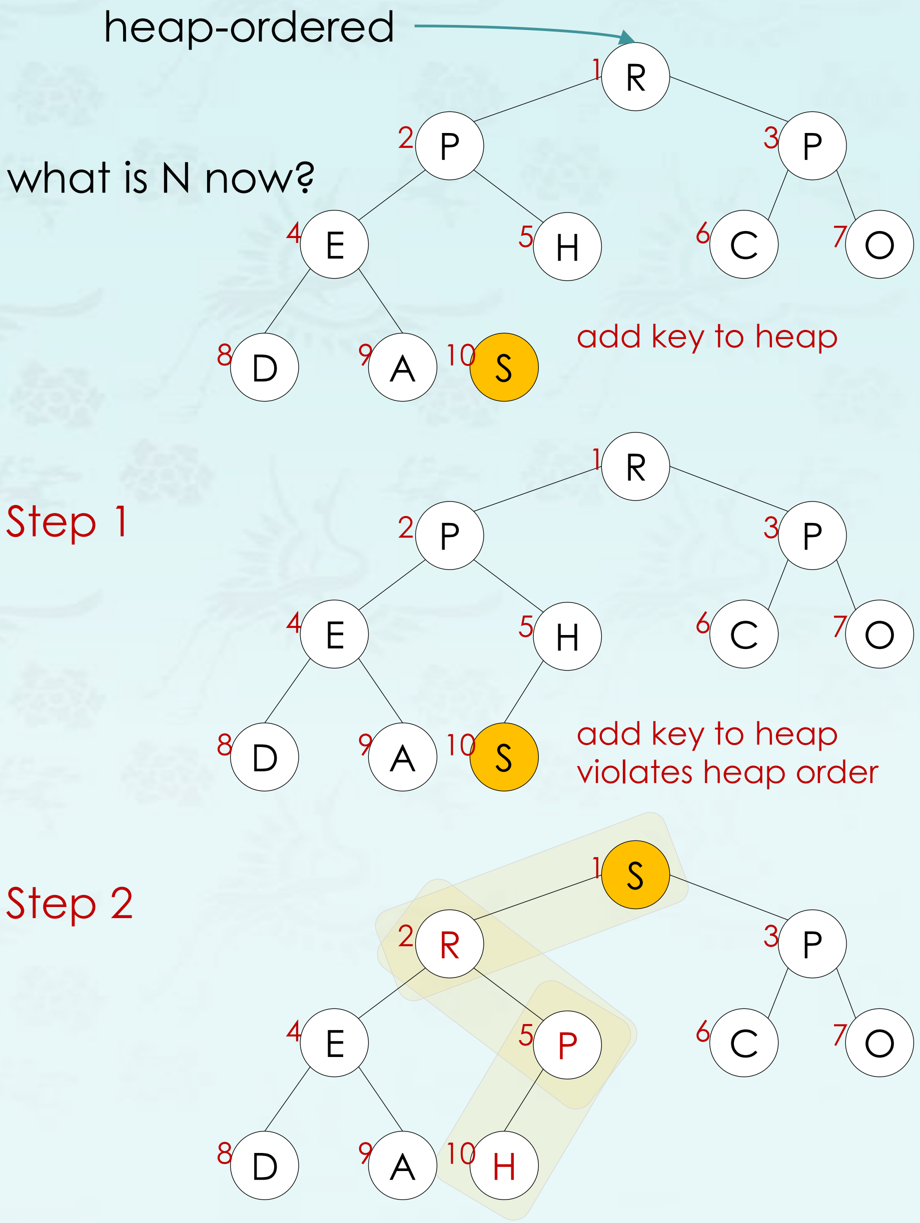
if k is not less than j, break;
swap k and j since k < j
set k to the next node (which is j)

Insert 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;
  int capacity;
  int N;
};
using heap = *Heap;
```



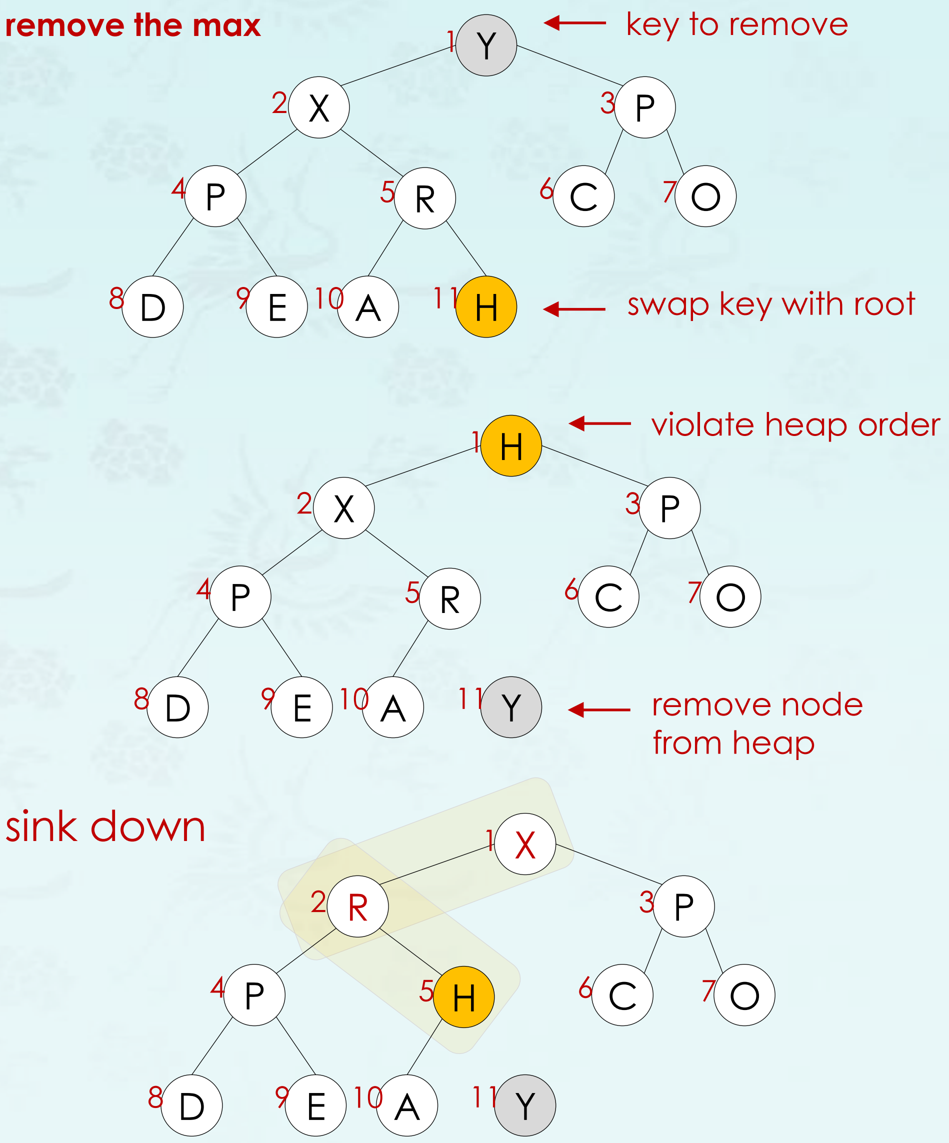
Delete in a heap

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 ADT

```
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 it heap-ordered?
```

Data Structures
Chapter 5: Heap and Priority Queue

1. Heap & Priority Queue
2. Heapsort
3. **Heap & PQ Coding**

Heap Coding

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

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

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

Heap Coding

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

```
void swap(heap hp, int i, int j) {
    int t = hp->nodes[i];
    hp->nodes[i] = hp->nodes[j];
    hp->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(hp)) 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()     build a binary tree from heap/CBT 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;
}
```

```
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*;
```

heapify – convert an int array to max/minheap

```
// start sink() at the last internal node(or parent of the last node)
// since leaf nodes already satisfy the max/min priority property
// This is O(n) algorithm.
void heapify(heap p) {
  for (int k = p->N; k >= 1; k--)
    sink(p, k);
} // this works but inefficiently. Fix it if you can.
```

Convert maxheap to minheap and vice versa

```
        ...
        case 'z':   // turn into max-heap or min-heap
          if (ordered)
            maxType = maxType ? false : true;
          else
            maxType = true;

          setType(hp, maxType);
          heapify(hp);
          ordered = true;
          break;
        ...

// driver.cpp
```

```
// sets the compare function less() for maxheap, more() for minheap.

void setType(heap p, bool maxType) {
  p->comp = maxType ? _________ : _____________;  // comparator fp
} // heap.cpp
```

Priority Queue

It is like a regular queue or stack data structure, but where additionally **each element has a "priority" associated with it.** In a priority queue, an element with high priority is served before an element with low priority.

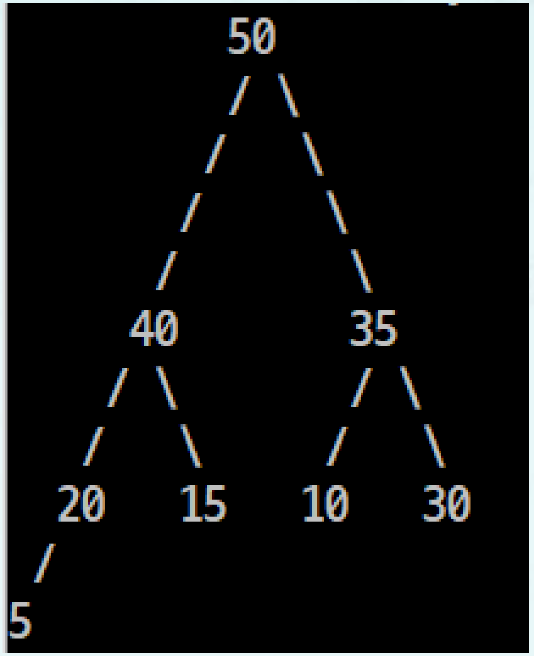
- **"trim"** removes the root which has the highest priority
- **"priority queue"** lets user modify the priority (or value) of an element) and move it to the position based on its new priority in the queue.

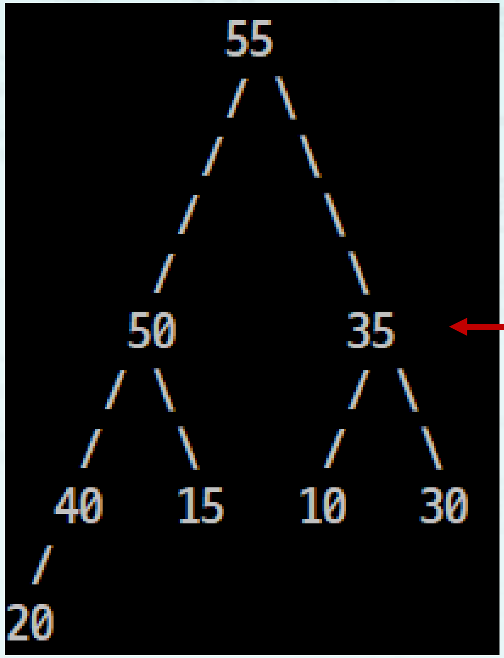
```
        [Tree built using iteration]
     7
    / \
   /   \
  /     \
 4       6
/ \     / \
0   1   5   3

        PQ(Maxheap): mn:0 mx:7 sz:7 ht:2 cp:8
        g - grow                h - heapify
        t - trim                s - heapsort
        p - priority queue      o - heap-ordered?

        x - grow N              w - switch to CBT
        y - trim N              z - switch to [PQ:max/minheap]
        c - clear               m - show mode:[tree]
        Command(q to quit):
```

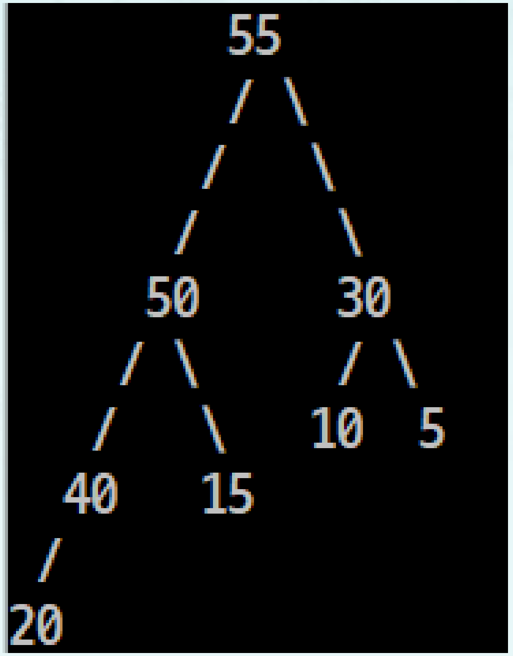
Priority Queue

For example:

- If you **change 5 to 55**, it will go up to the root and 20 is placed at the bottom.
- If you **change 35 to 5**, 30 will go up where 35 is, then 5 goes down to the right corner.







grow() - inserts a new key to the max-heap or min-heap.

grow(heap p, int key)

1. if full(p), invoke **reserve()** to double the size of nodes[]. Use p->capacity * 2.
2. add the key to nodes[]. The index of nodes[] must be ++p->N.
3. swim up to maintain heap invariant.

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

  if (full(p))
    ...
  p->nodes ...
  swim...

  return;
}
```

growN()

1. Find the max key(max) in heap or CBT.
2. Set a function pointer to the function to insert a node.
3. Allocate a Key type array such as keys to store random keys.
4. Invoke randomN() function to generate keys in the range [(max + 1)..(max + count)]
5. Invoke the function to insert keys in keys[], but one key at a time.
6. Print the heap if DEBUG is defined whenever a node is inserted.
7. Don't forget freeing the array of keys you allocated in Step 3.

```
void growN(heap p, int count, bool heapOrdered) {
  int max = empty(p) ? 1 : maximum(p) + 1;
  void(*insertFunc)(heap h, int key) = heapOrdered ? grow : growCBT;




                                                    // insertFunc(p, keys[i]);




}
```

growN()

1. Find the max key(max) in heap or CBT.
2. Set a function pointer to the function to insert a node.
3. Allocate a Key type array such as keys to store random keys.
4. Invoke randomN() function to generate keys in the range [(max + 1)..(max + count)]
5. Invoke the function to insert keys in keys[], but one key at a time.
6. Print the heap if DEBUG is defined whenever a node is inserted.
7. Don't forget freeing the array of keys you allocated in Step 3.

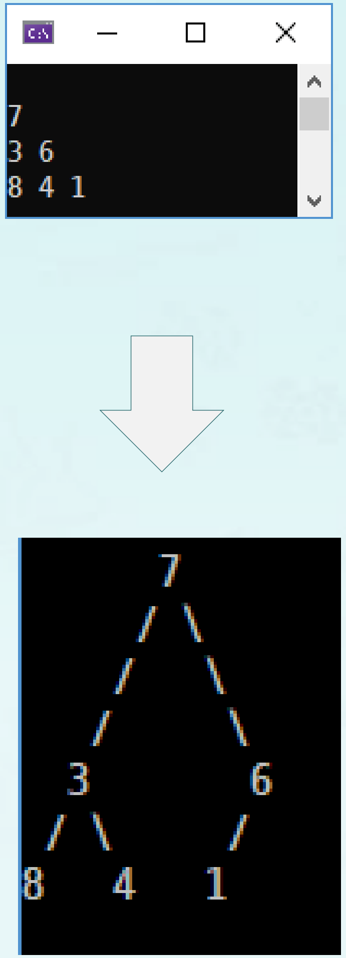
```
void growN(heap p, int count, bool heapOrdered) {
  int max = empty(p) ? 1 : maximum(p) + 1;
  void(*insertFunc)(heap h, int key) = heapOrdered ? grow : growCBT;
```

```
  7
 / \
5   6
```

```
       Enter a number of nodes to grow: 4
     11
     / \
    /   \
   /     \
  10      9
 / \     / \
5   7   6   8
```

```
}
```

Heapprint(): build a tree from CBT - heapprint.cpp




```
// print a heap using treeprint() –
// build a tree to call treeprint()

void heapprint(heap p, int mode) {
  if (empty(p)) return;
  if (size(p) % 2 == 0) {
    cout << "\t[Tree built using recursion]\n";
    root = buildBT(p->nodes, 1, size(p));  // using recursion
  }
  else {
    cout << "\t[Tree built using iteration]\n";
    root = buildBT(p);                      // using iteration
  }
  ...
  ... treeprint(root);
  ...
  clear(root);
}
```

heapprint() – build a binary tree from heap/CBT 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

left subtree | root | right subtree

mid = 3

n = 6

	0	1	2	3	4	5
v	2	3	5	6	8	9

left | root | right

mid = 2

n = 5

	0	1	2	3	4
v	15	20	22	23	25

left | root | right

mid =1

n = 2

	0	1
v	15	20

left | root

mid =1

n = 2

	0	1
v	23	25

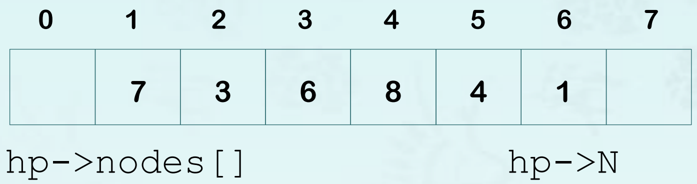
left | root

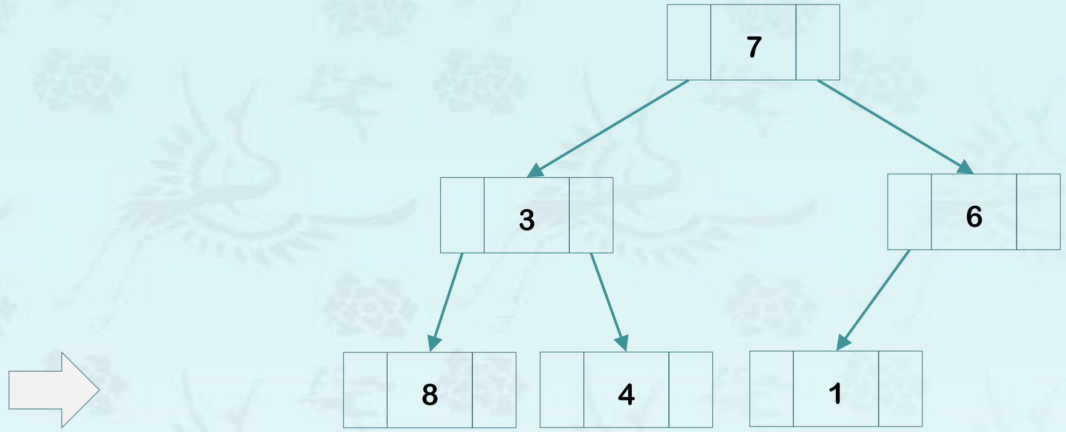
Building AVL tree from BST in O(n) – Review

```
// 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 reconstruct(tree root) {
  if (root == nullptr) return nullptr;
  if (size(root) > 10) { // recycling method
    vector<tree> v;      // get nodes sorted
    ...
    root = buildAVL(...);
  }
  else {                 // recreation method
    vector<int> v;       // get keys sorted
    ...
    ...
    root = buildAVL(...);
  }
  return root;
}
```

heapprint() – build a binary tree from heap/CBT 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) {`

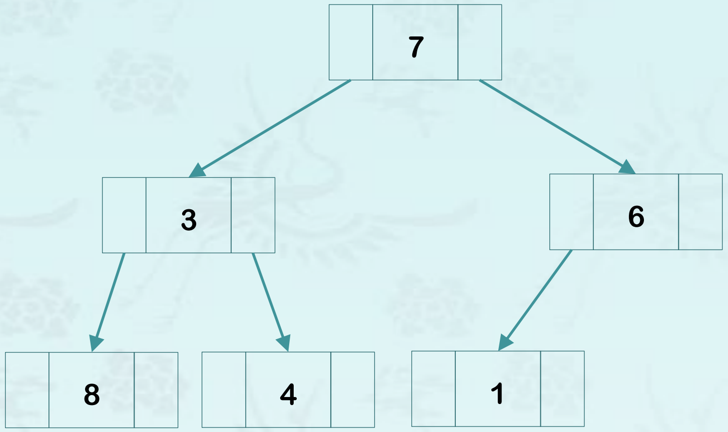
1. If **`i > n`**, return nulltptr – terminate condition
2. Create the **tree (root) node** with **nodes[i]**).
   A. Invoke buildBT() for all its left children (or **`i * 2`**). Set its return to the left child of the root.
   B. Invoke buildBT() for all its right children (or **`i * 2 + 1`**). Set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

}

heapprint() – build a binary tree from heap/CBT using recursion

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

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



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

}

```
tree buildBT(int *nodes, int i, int n) {
  if (i > n) return nullptr;
  tree root = new TreeNode{ nodes[i] };
  root->left  = ...
  root->right = ...
  return  root;
}
```

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

heapprint() – build a binary tree from heap/CBT 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

	7	

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

heapprint() – build a binary tree from heap/CBT 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

	7	

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

```
tree buildBT(*nodes, i=2, n=6) { }
```

System Stack

push stack

i=1, n=6
ret. addr.

	7	

heapprint() – build a binary tree from heap/CBT 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

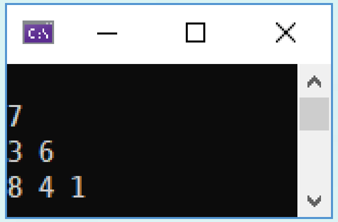
7

`tree buildBT(*nodes, i=2, n=6) { }`

System Stack

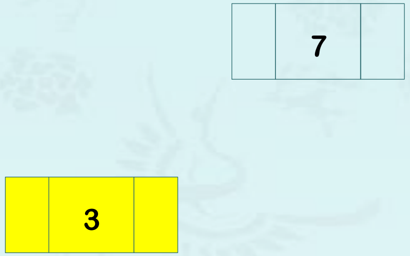
i=1, n=6
ret. addr.
7

heapprint() – build a binary tree from heap/CBT using recursion



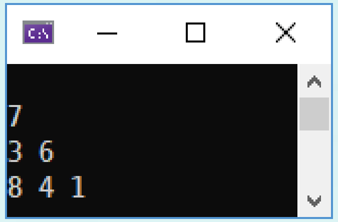
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N



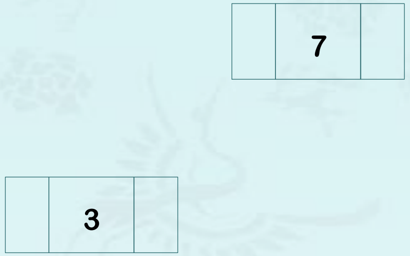
`tree buildBT(*nodes, i=2, n=6) { }`

heapprint() – build a binary tree from heap/CBT using recursion



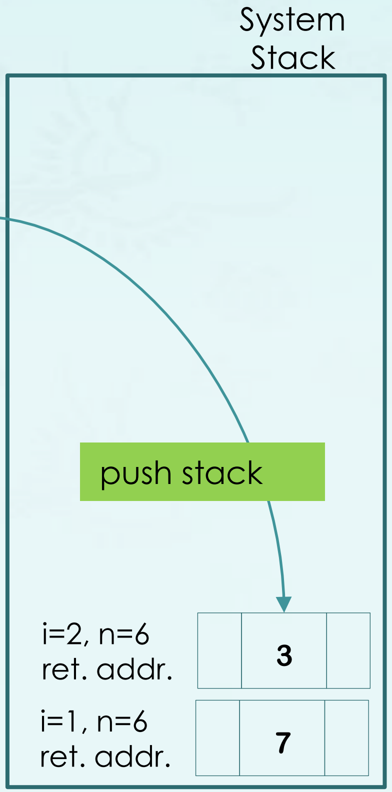
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

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

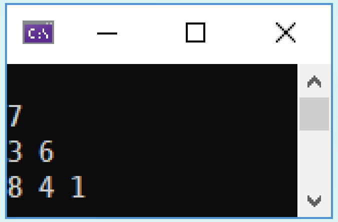



```
tree buildBT(*nodes, i=2, n=6) { }
```

```
tree buildBT(*nodes, i=4, n=6) { }
```

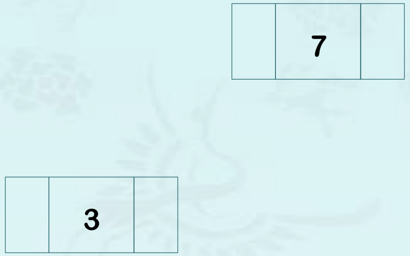


heapprint() – build a binary tree from heap/CBT using recursion



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

hp->nodes[] hp->N

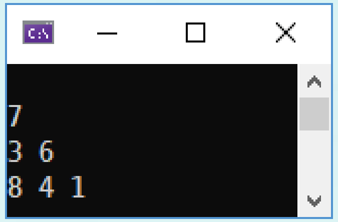



```
tree buildBT(*nodes, i=4, n=6) { }
```

System Stack

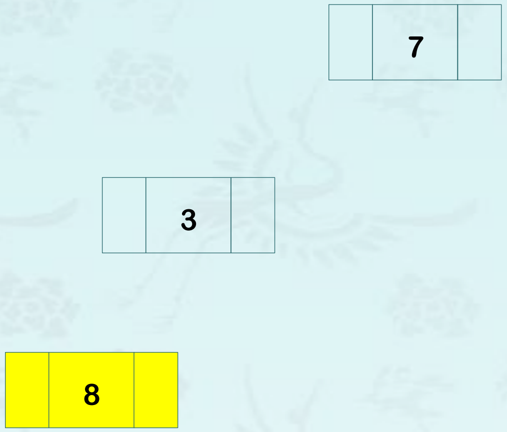
i=2, n=6 ret. addr.	3
i=1, n=6 ret. addr.	7

heapprint() – build a binary tree from heap/CBT using recursion



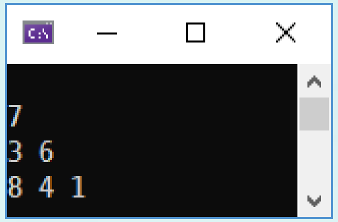
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N



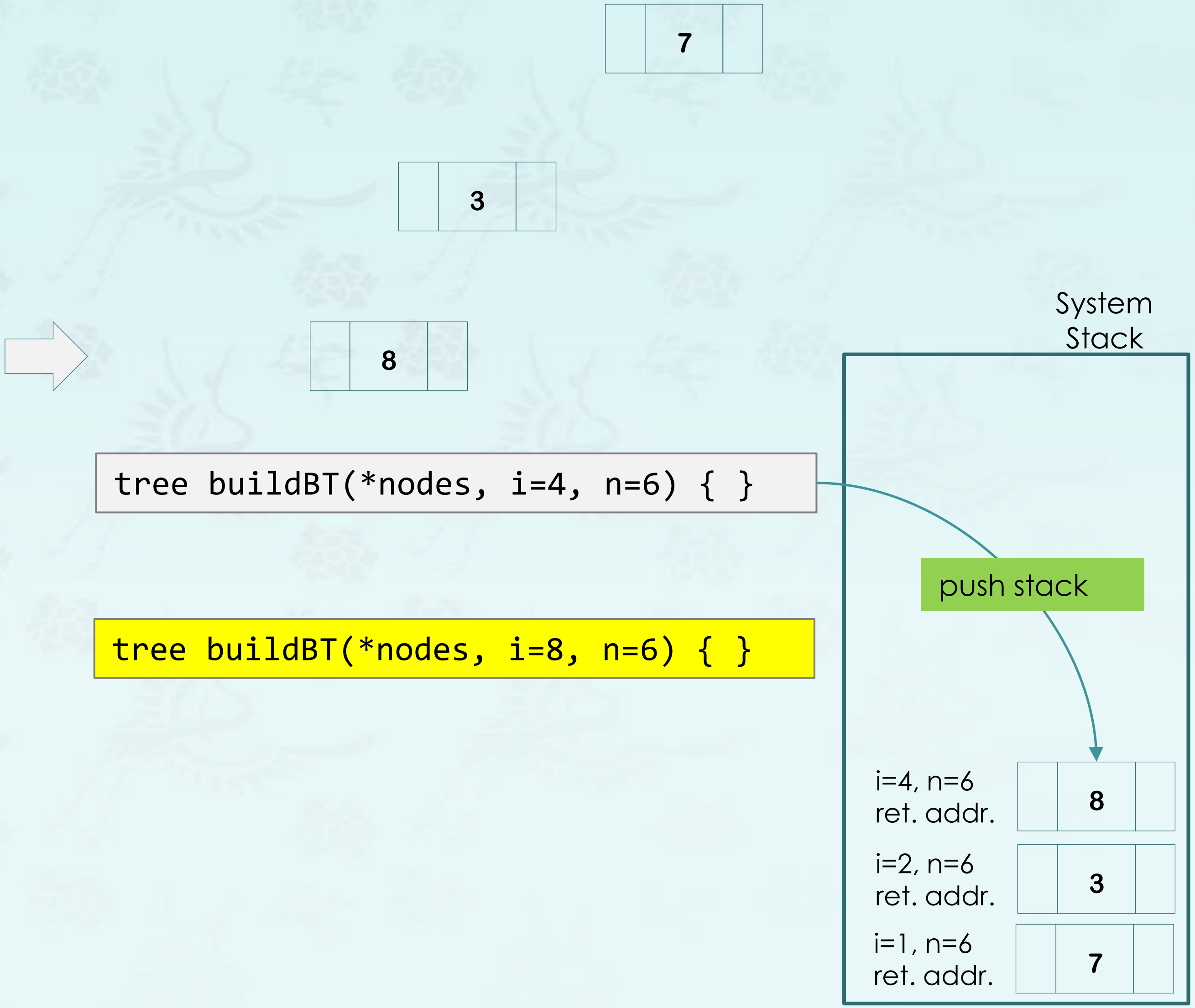

```
tree buildBT(*nodes, i=4, n=6) { }
```

heapprint() – build a binary tree from heap/CBT using recursion



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

hp->nodes[]   hp->N

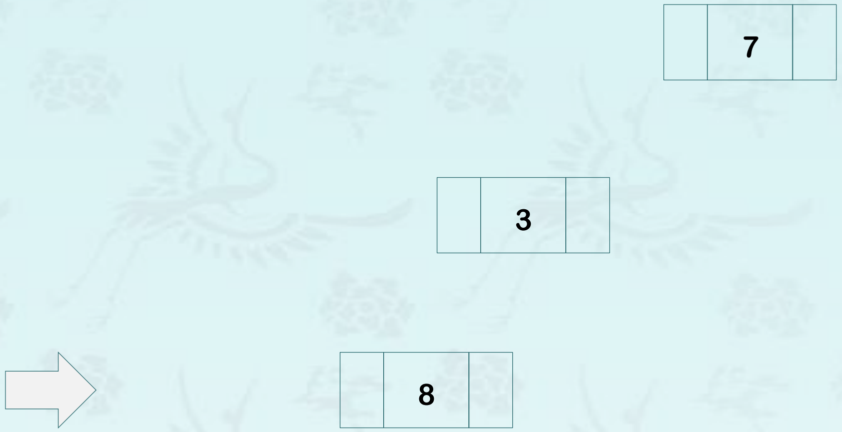


heapprint() – build a binary tree from heap/CBT using recursion

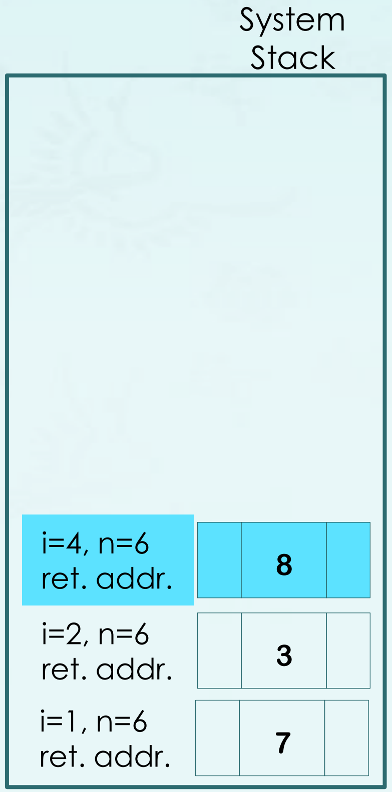
```
7
3 6
8 4 1
```

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

hp->nodes[] hp->N




```
tree buildBT(*nodes, i=8, n=6) { }
```

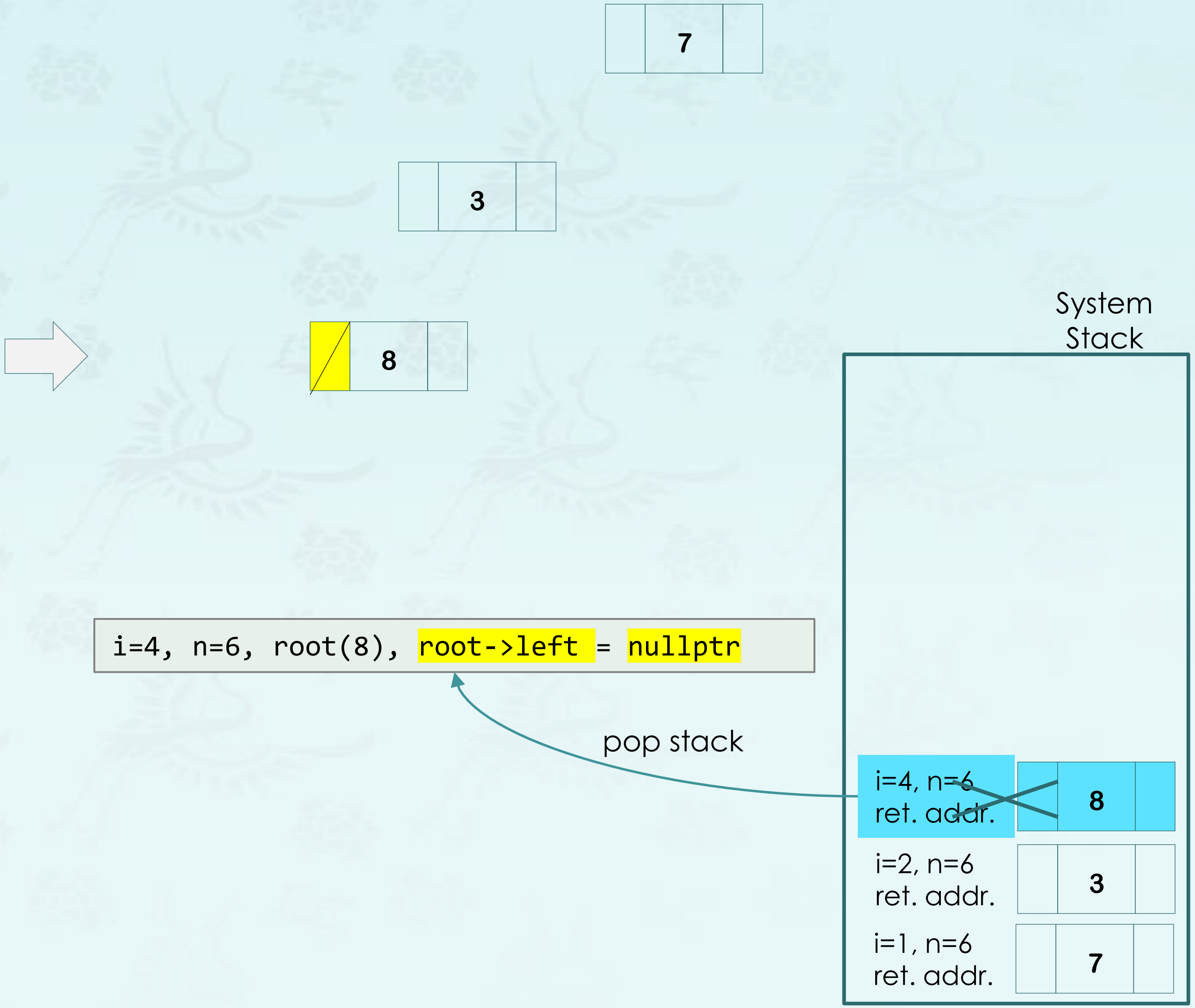


heapprint() – build a binary tree from heap/CBT using recursion

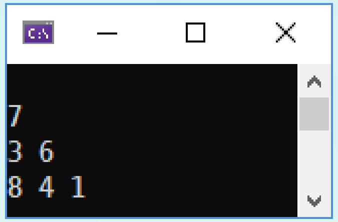
```
7
3 6
8 4 1
```

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

hp->nodes[]   hp->N

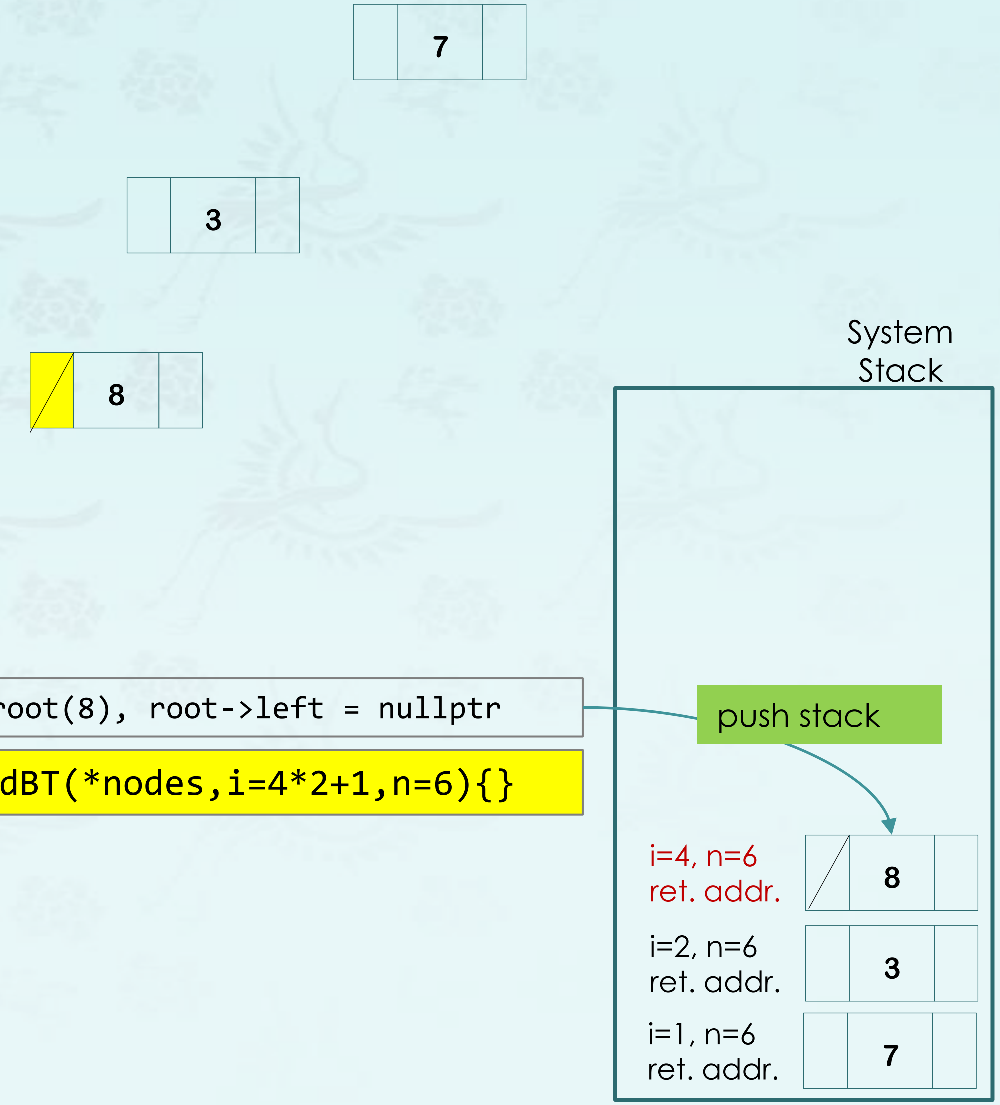


heapprint() – build a binary tree from heap/CBT using recursion



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

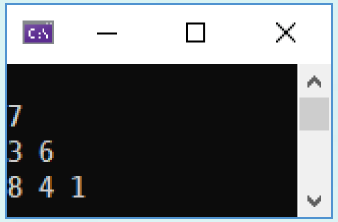
hp->nodes[] hp->N




```
i=4, n=6, root(8), root->left = nullptr
```

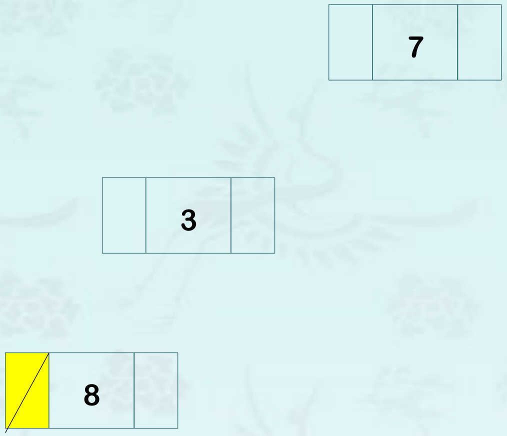
```
tree buildBT(*nodes,i=4*2+1,n=6){}
```

heapprint() – build a binary tree from heap/CBT using recursion

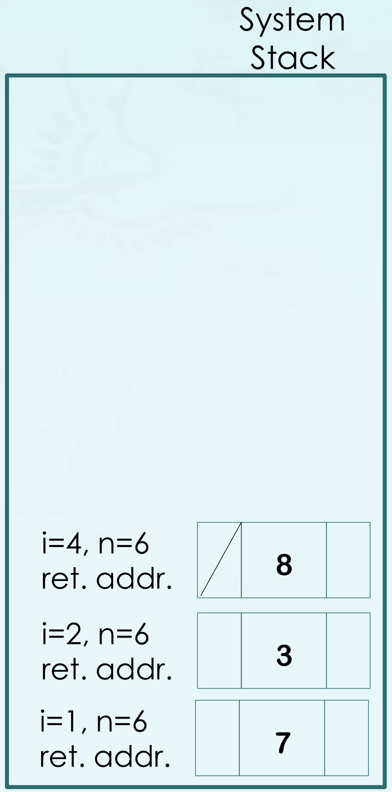


0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

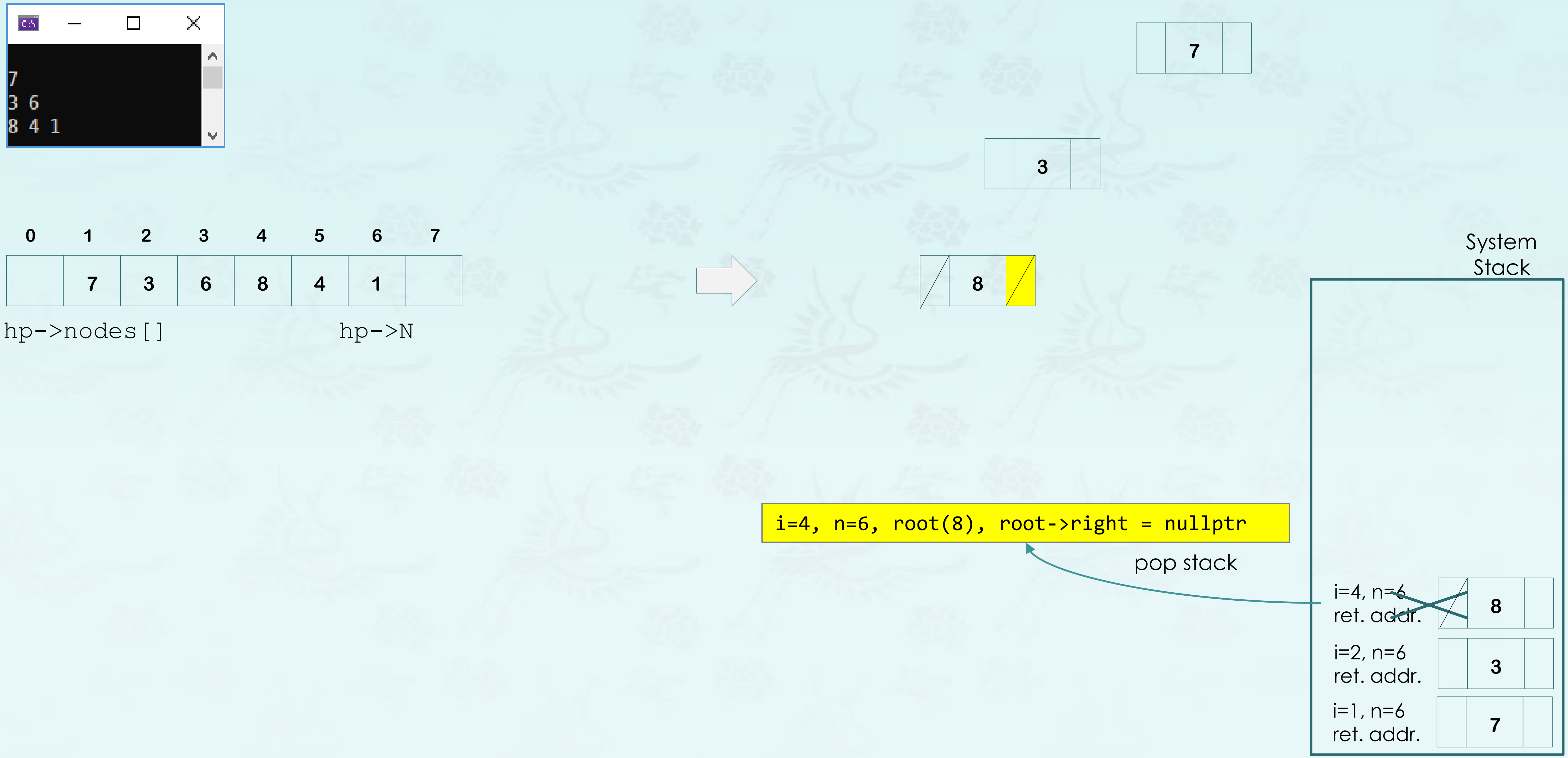
hp->nodes[] hp->N



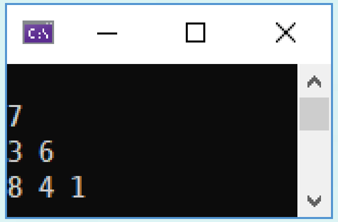

```
tree buildBT(*nodes, i=9, n=6) { }
```

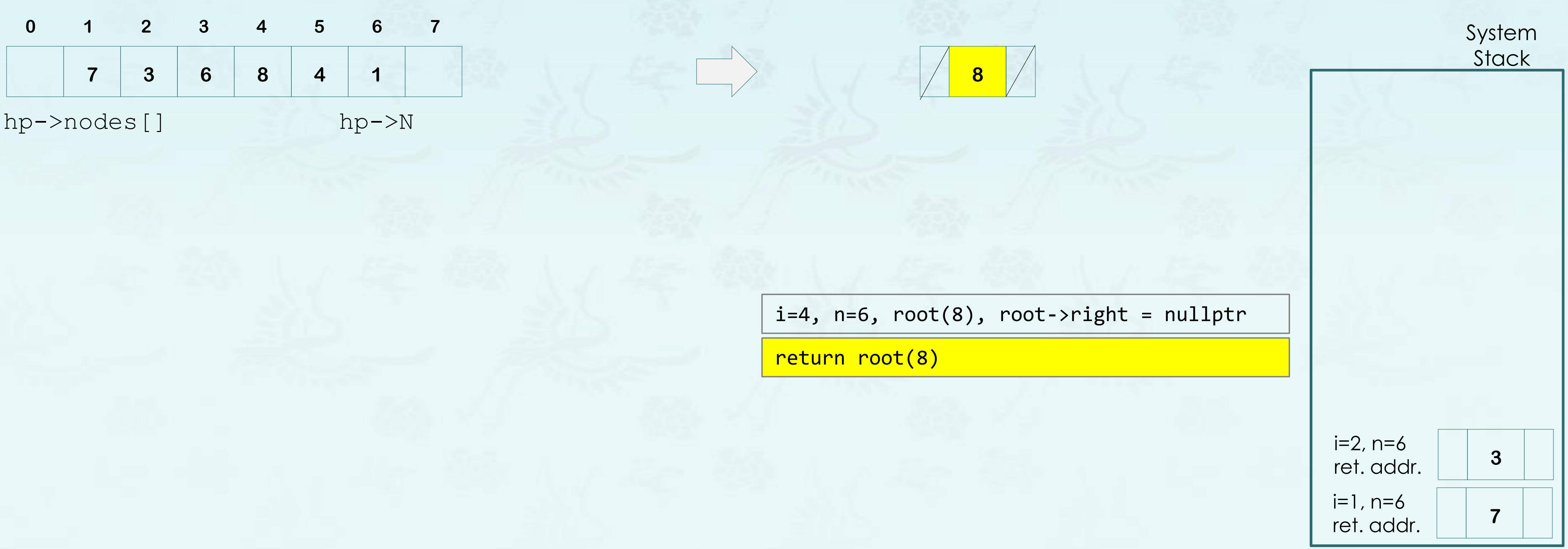


heapprint() – build a binary tree from heap/CBT using recursion

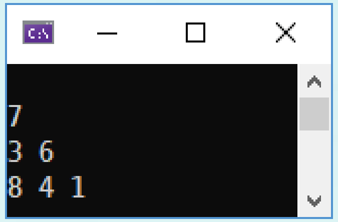


heapprint() – build a binary tree from heap/CBT using recursion



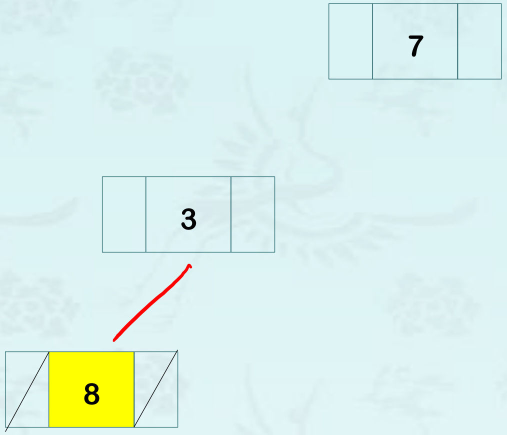


heapprint() – build a binary tree from heap/CBT using recursion



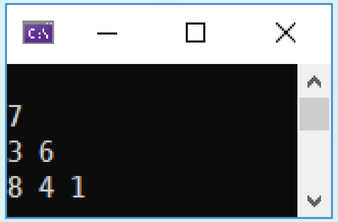
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[]   hp->N



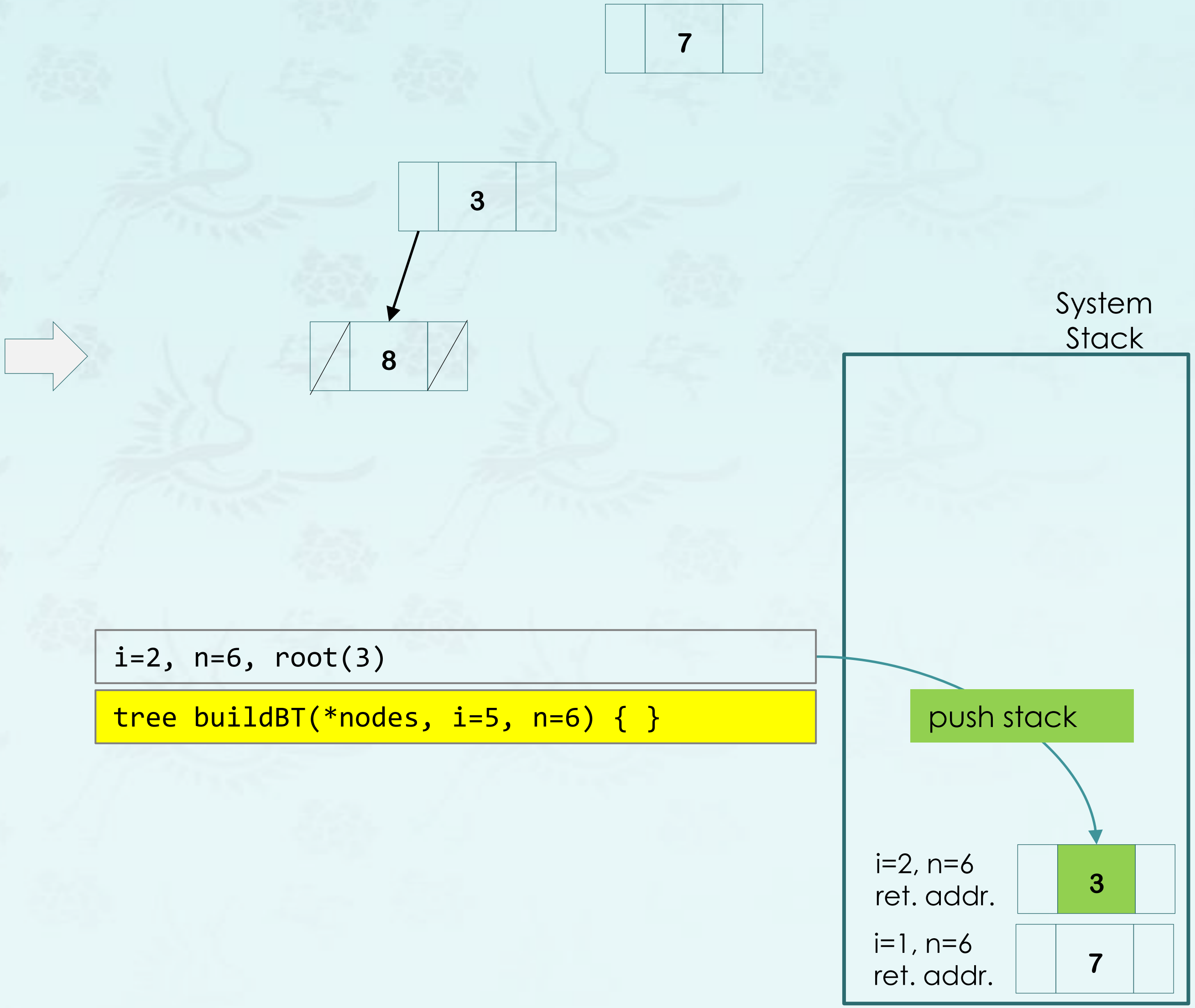

```
i=4, n=6, root(8), root->right = nullptr
return root(8)
```

heapprint() – build a binary tree from heap/CBT using recursion



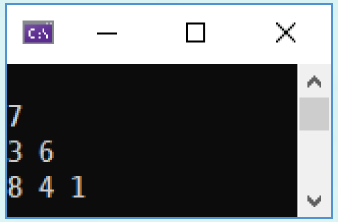
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N



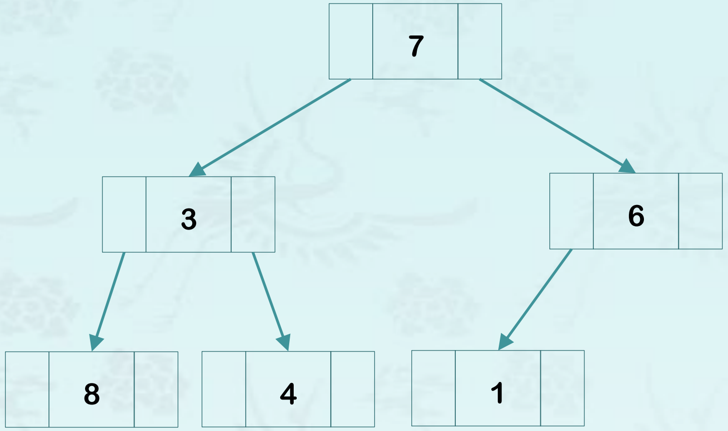
Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm202, Handong Global University

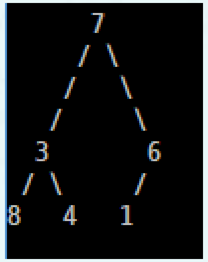
heapprint() – build a binary tree from heap/CBT using recursion



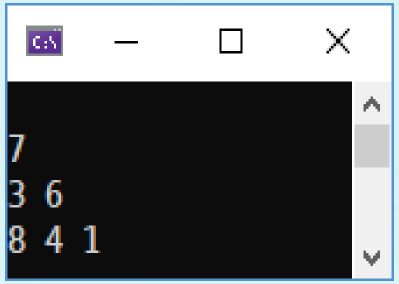
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

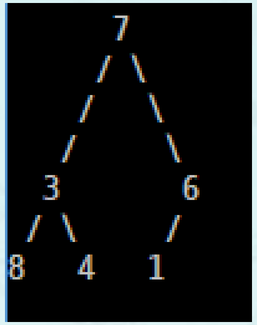
`hp->nodes[]` `hp->N`

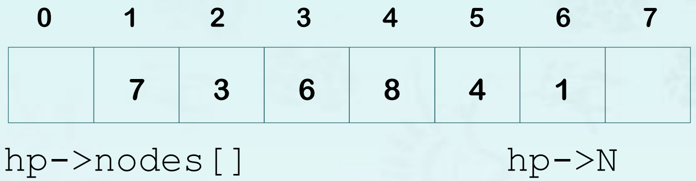


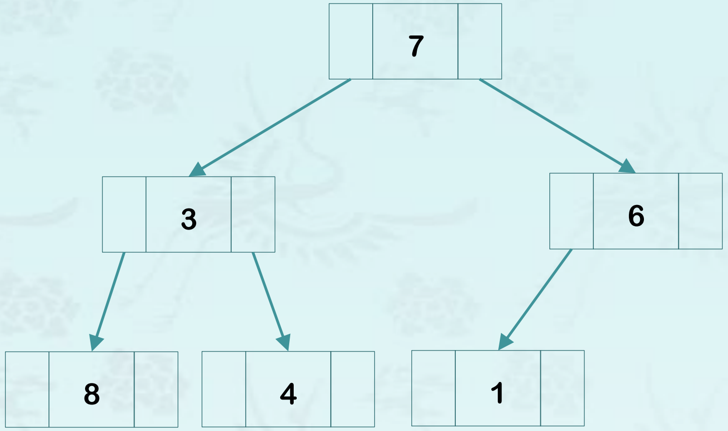


heapprint() – build a binary tree from heap/CBT using queue

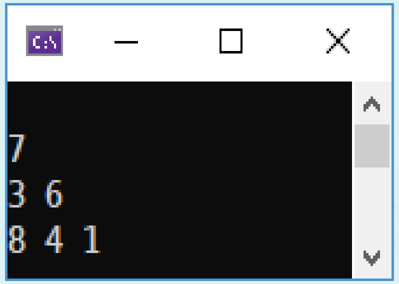


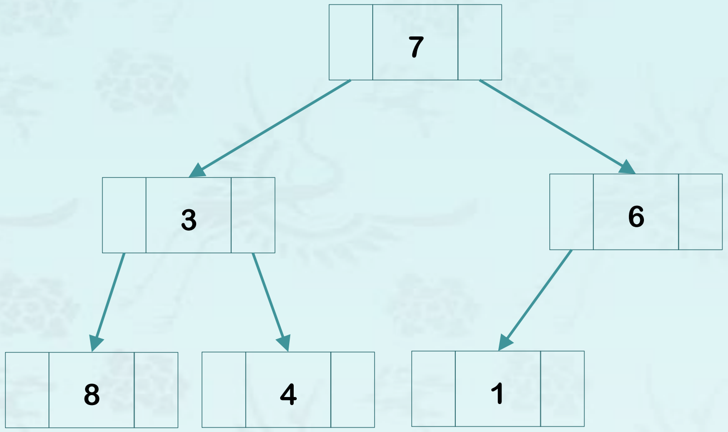






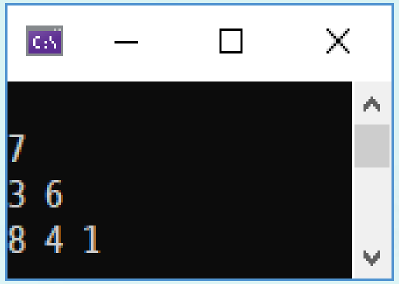
heapprint() – build a binary tree from heap/CBT using queue

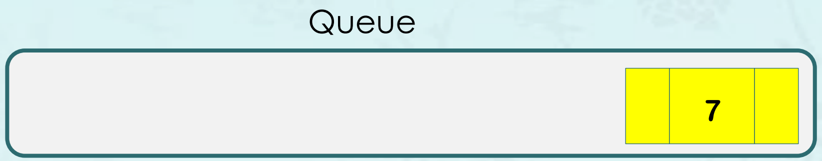




1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

heapprint() – build a binary tree from heap/CBT using queue



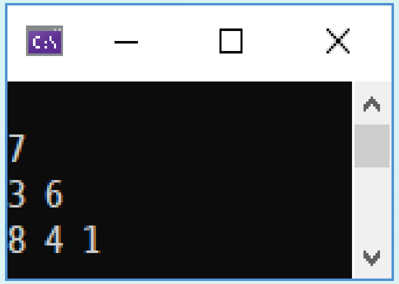


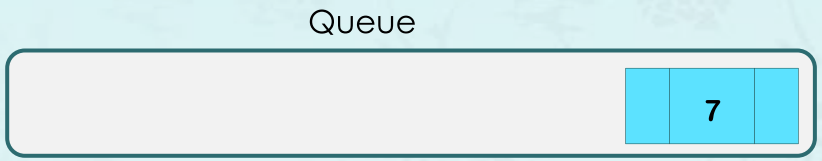
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

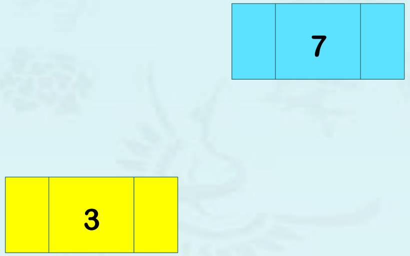
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





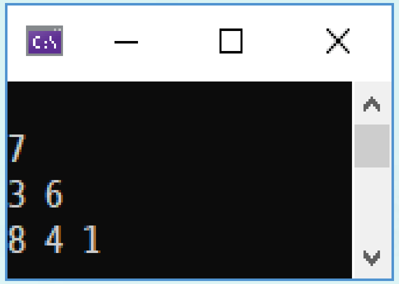


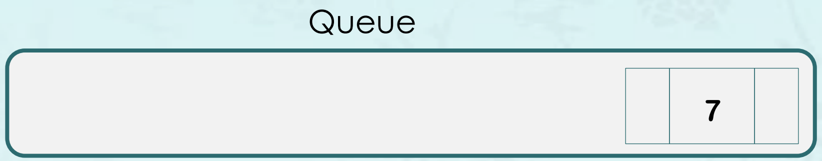
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

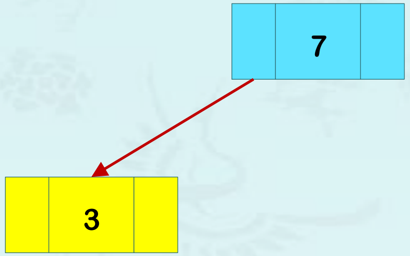
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





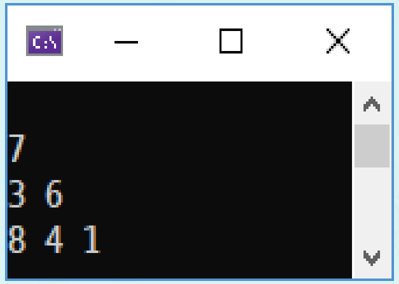


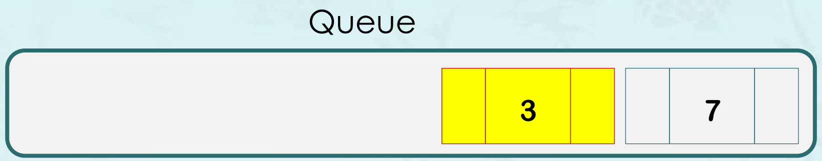
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

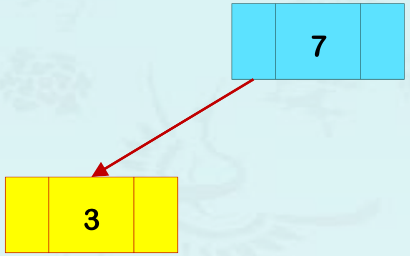
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





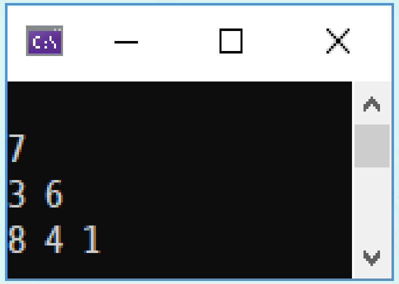


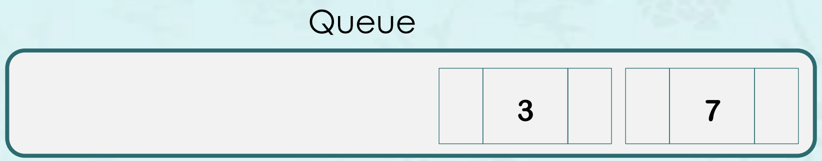
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

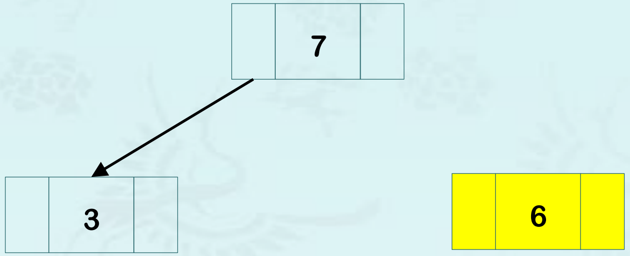
hp->nodes[] hp->N

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





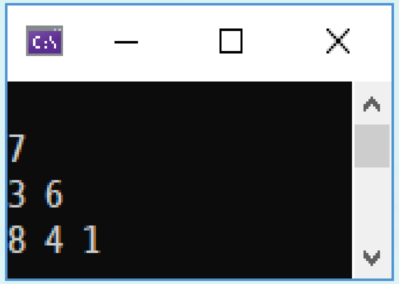


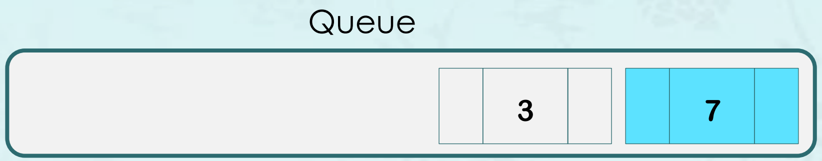
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

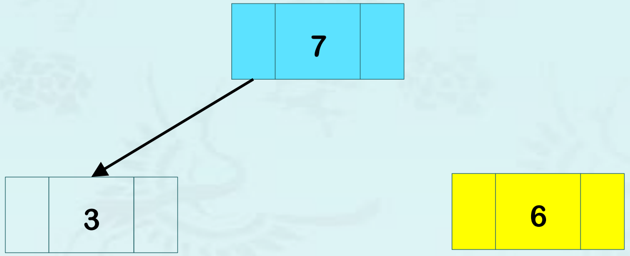
hp->nodes[] hp->N

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





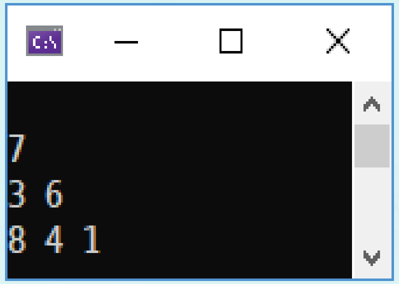


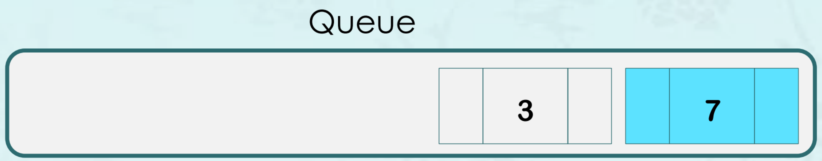
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

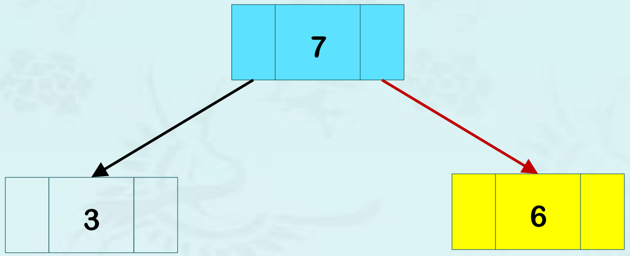
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





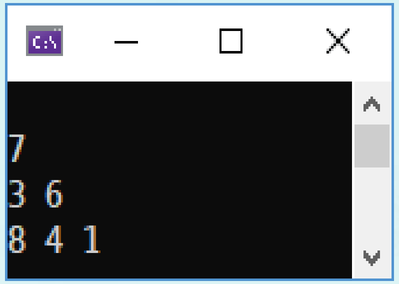


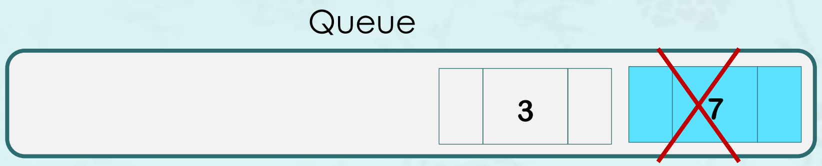
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

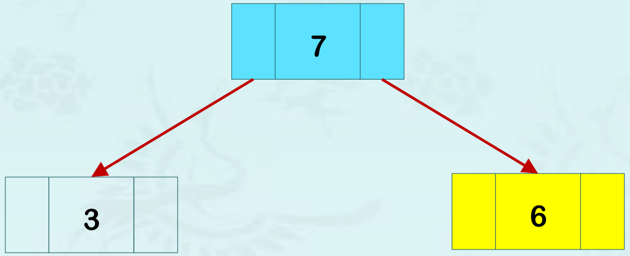
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





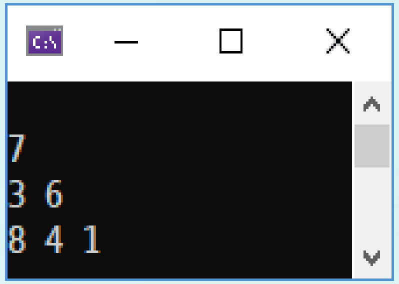


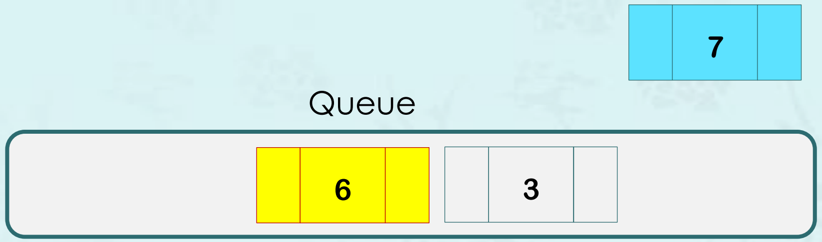
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

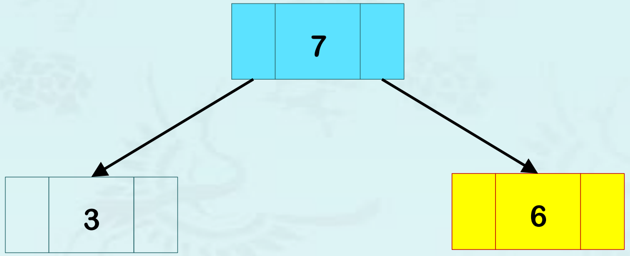
hp->nodes[] hp->N

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





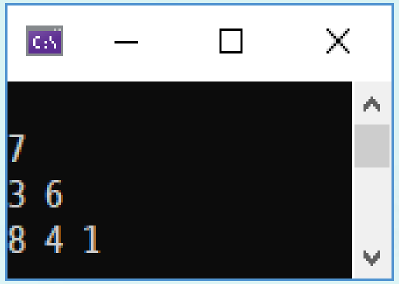


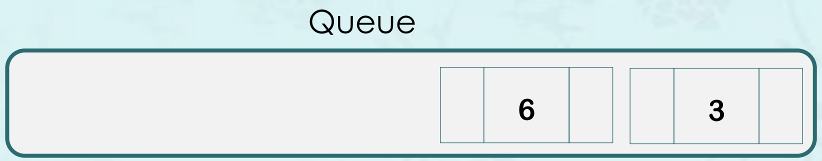
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

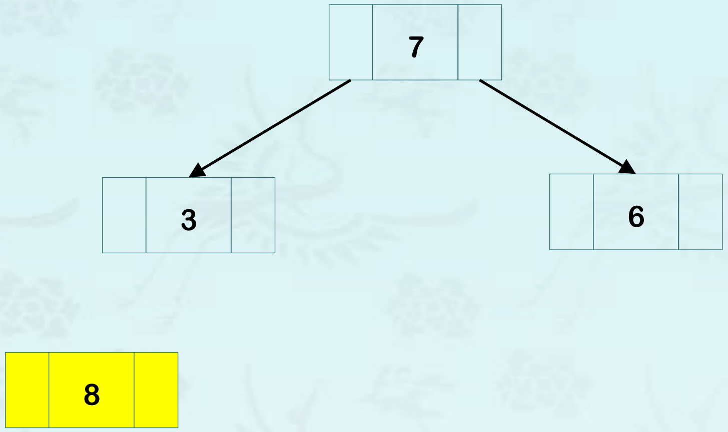
hp->nodes[] hp->N

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue





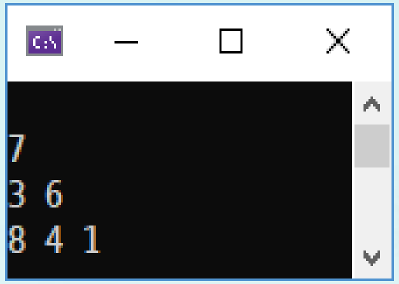


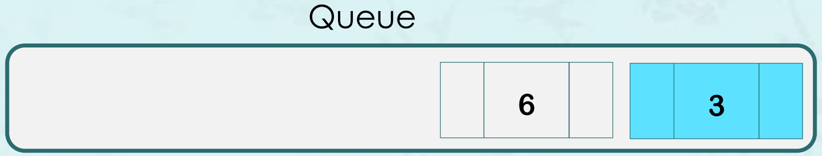
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

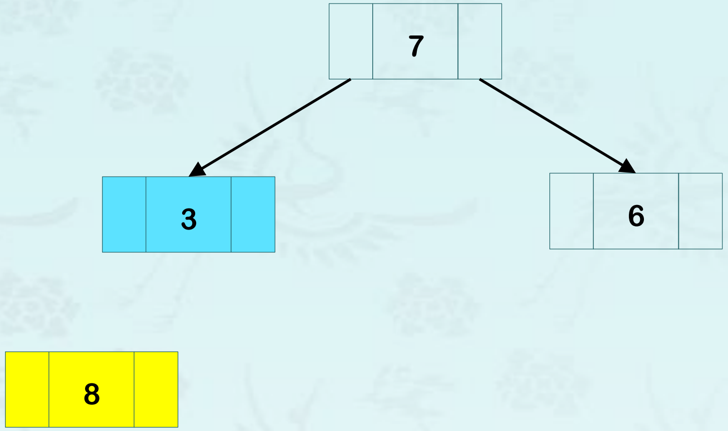
`hp->nodes[]` `hp->N`

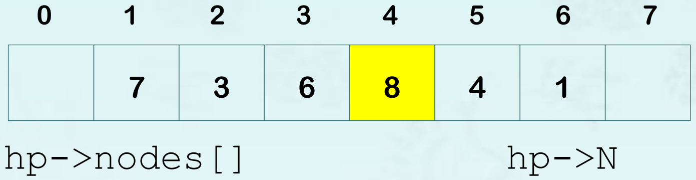
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[]**.
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

heapprint() – build a binary tree from heap/CBT using queue



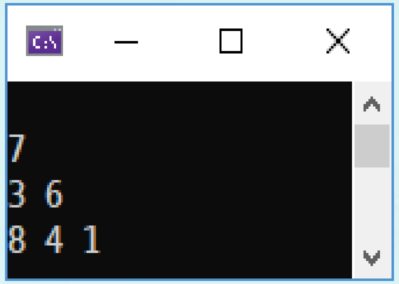


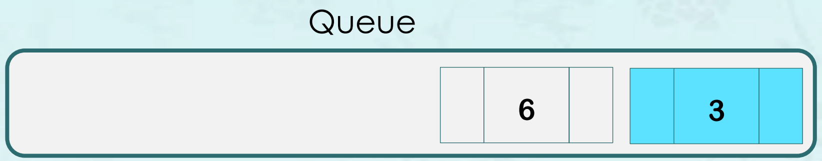




1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[].**
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
  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

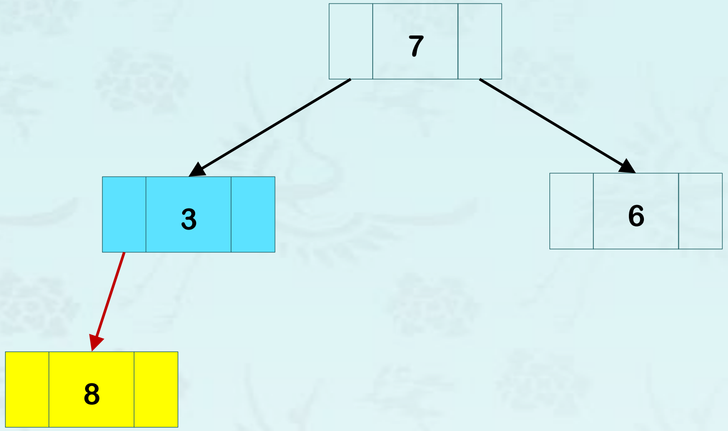
heapprint() – build a binary tree from heap/CBT using queue





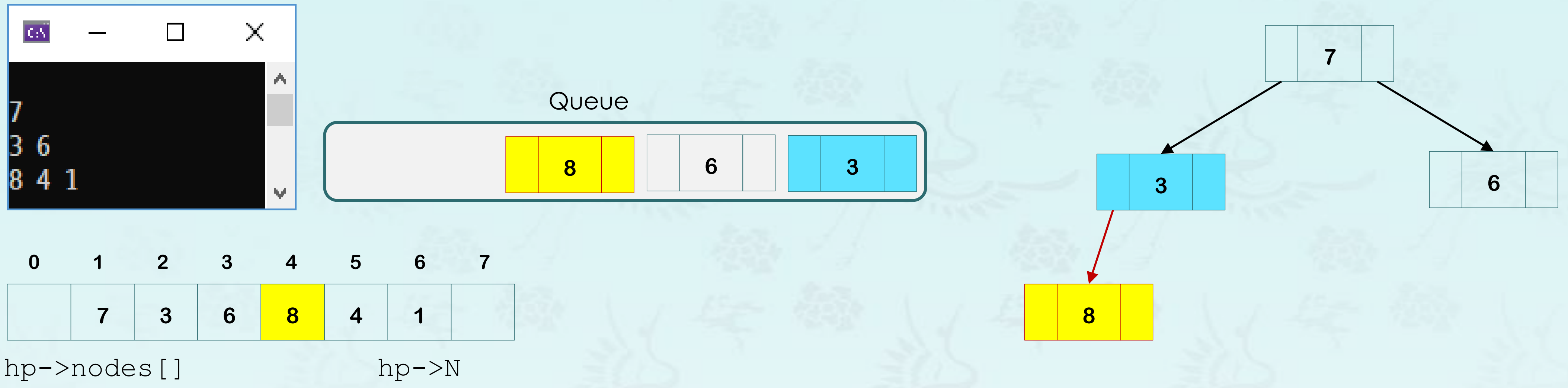
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[]   hp->N



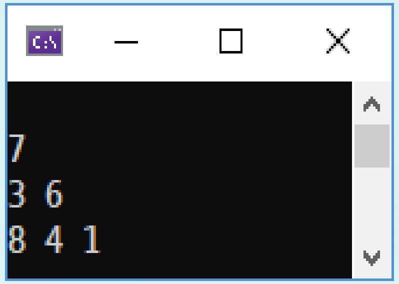
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[]**.
   - B. Get a **tree node** in the queue.
   - **C. If the left of the tree node doesn't exist,**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

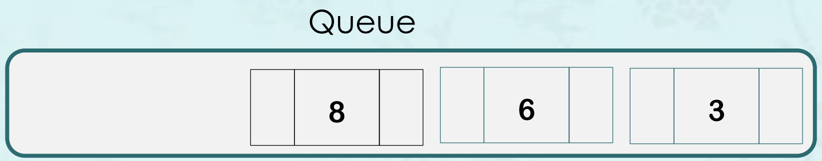
heapprint() – build a binary tree from heap/CBT using queue

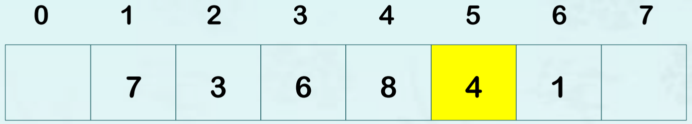


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[].**
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

heapprint() – build a binary tree from heap/CBT using queue

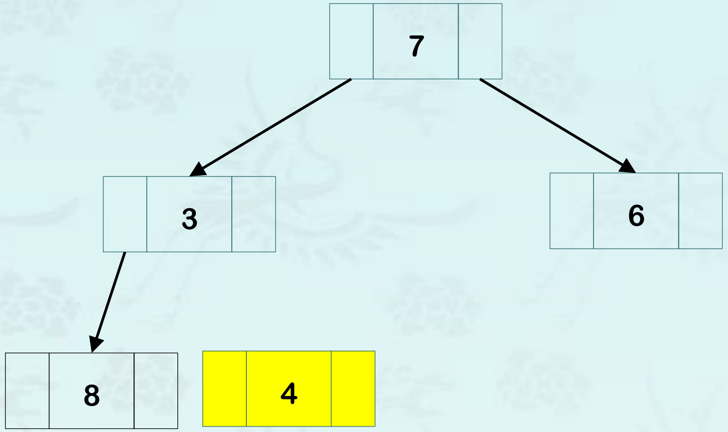






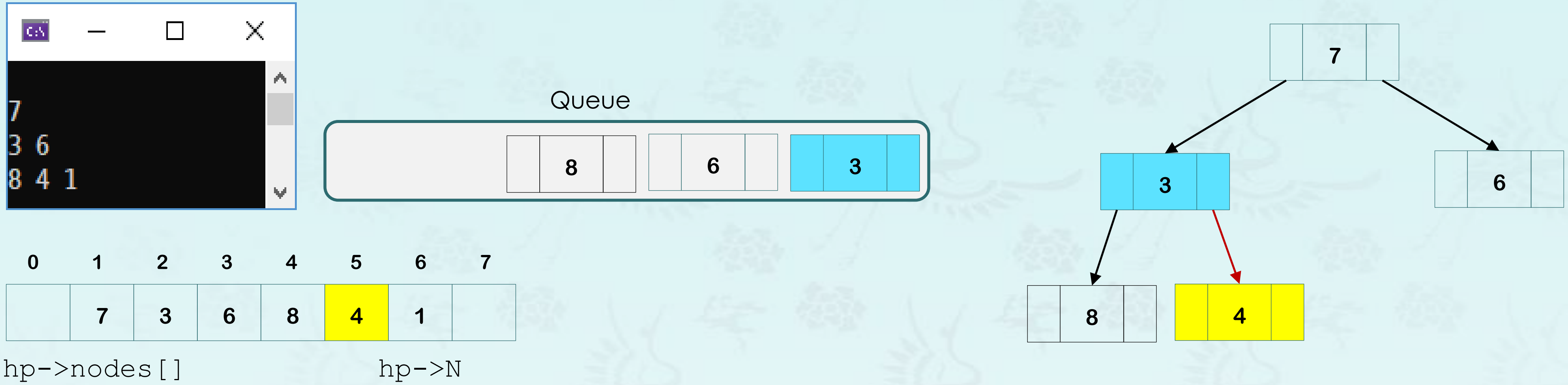
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

hp->nodes[] hp->N



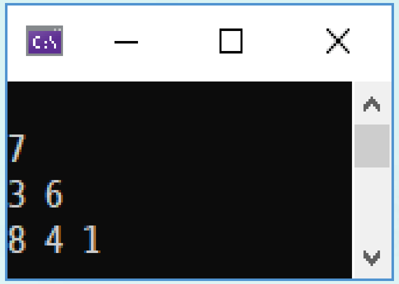
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[]**.
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
  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

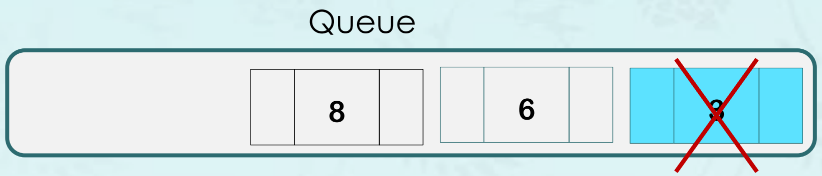
heapprint() – build a binary tree from heap/CBT using queue

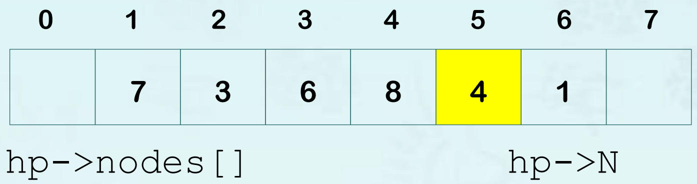


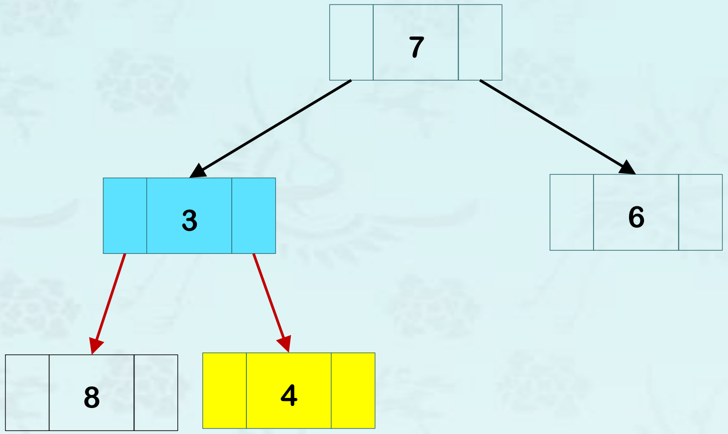
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   - A. Make a **new node from nodes[]**.
   - B. Get a **tree node** in the queue.
   - C. If the left of the tree node doesn't exist,
  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

heapprint() – build a binary tree from heap/CBT using queue



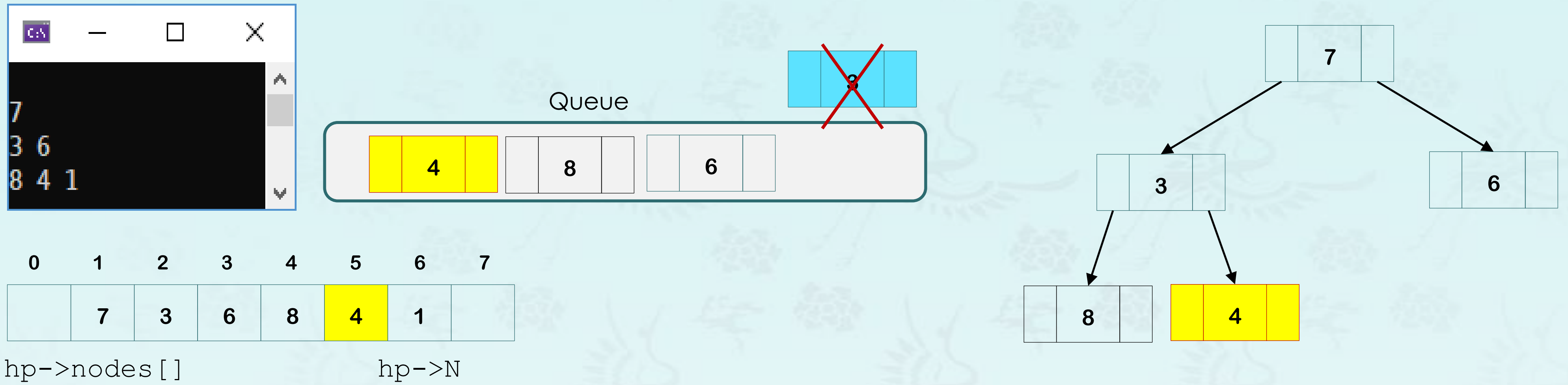






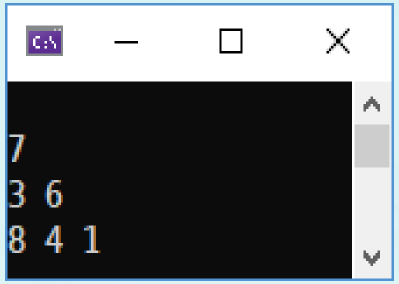
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[]**.
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

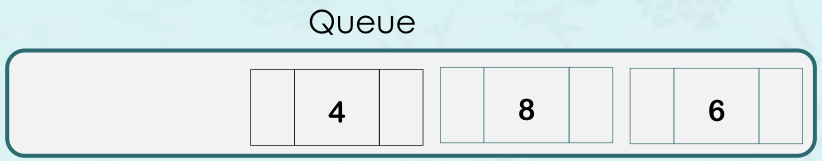
heapprint() – build a binary tree from heap/CBT using queue

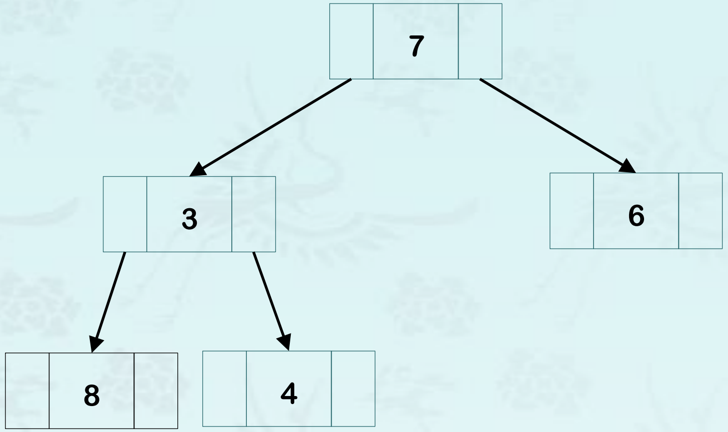


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[]**.
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

heapprint() – build a binary tree from heap/CBT using queue





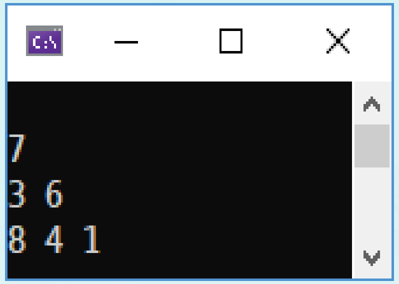


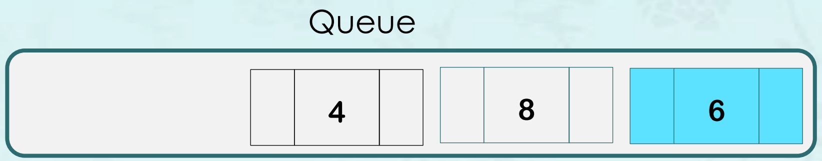
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

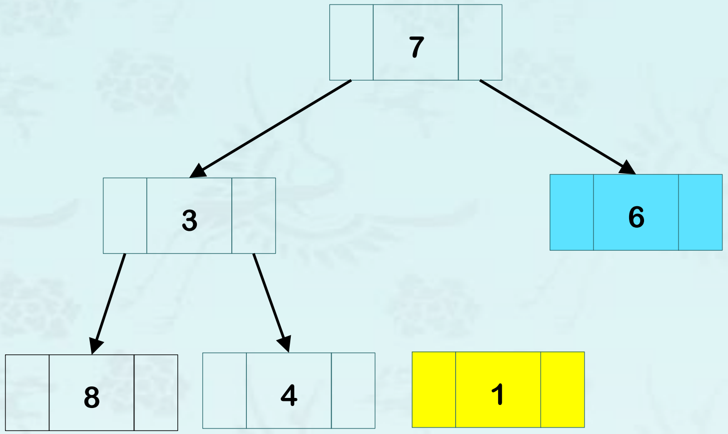
`hp->nodes[]` `hp->N`

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT using queue

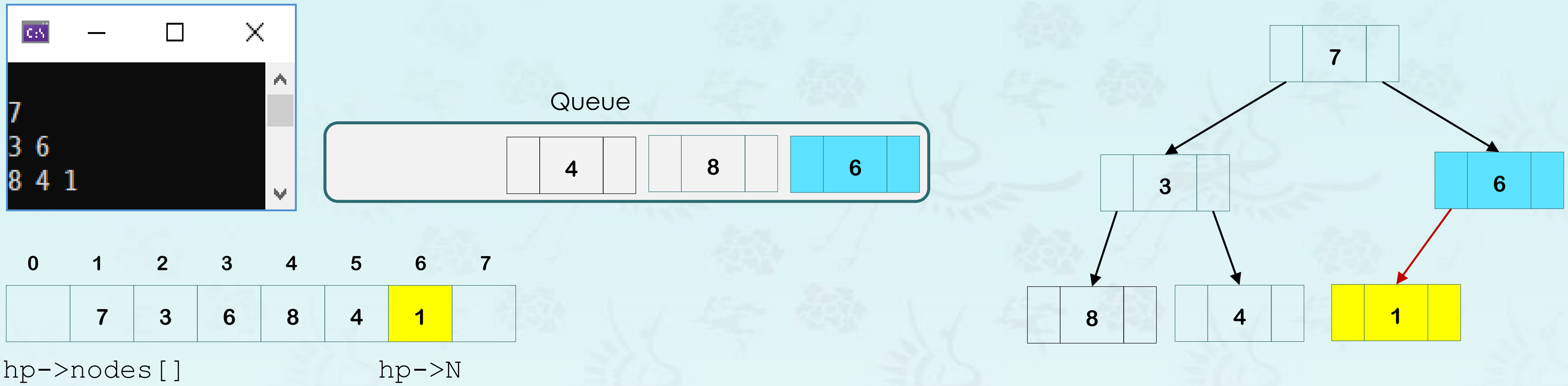






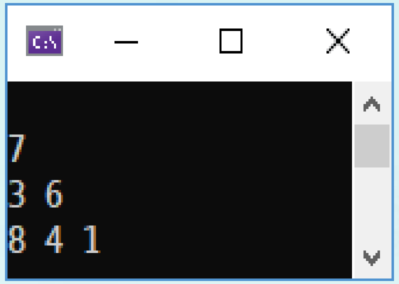
1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[]**.
   B. Get a **tree node** in the queue.
   C. If the left of the tree node doesn't exist,
      set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

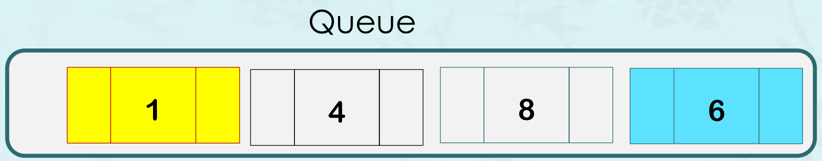
heapprint() – build a binary tree from heap/CBT using queue

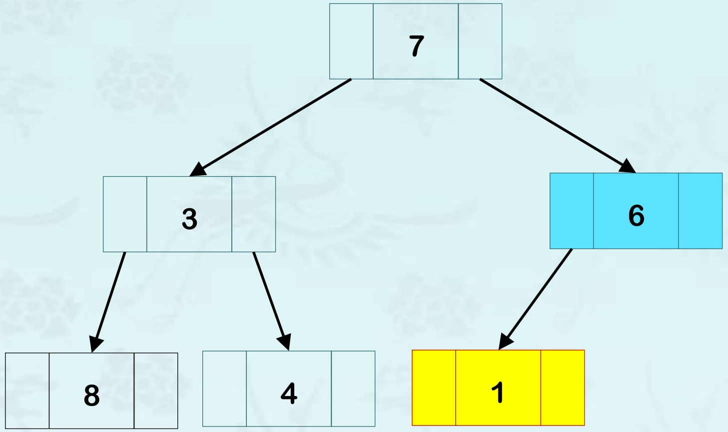


1. Create the **tree (root) node** with the first key from CBT (or **nodes[1]**).
2. Enqueue the root node.
3. Loop through from **the CBT nodes[2] to nodes[N]**
   A. Make a **new node from nodes[]**.
   B. Get a **tree node** in the queue.
   C. **If the left of the tree node doesn't exist,**
    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

heapprint() – build a binary tree from heap/CBT using queue





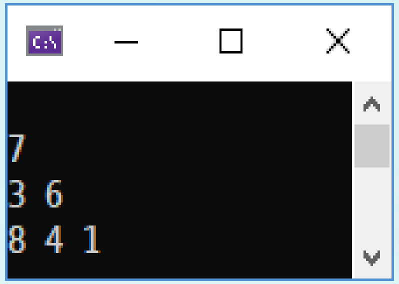


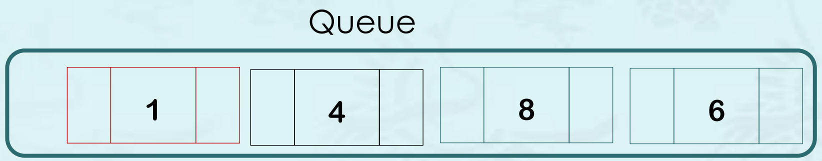
0	1	2	3	4	5	6	7
	7	3	6	8	4	1	

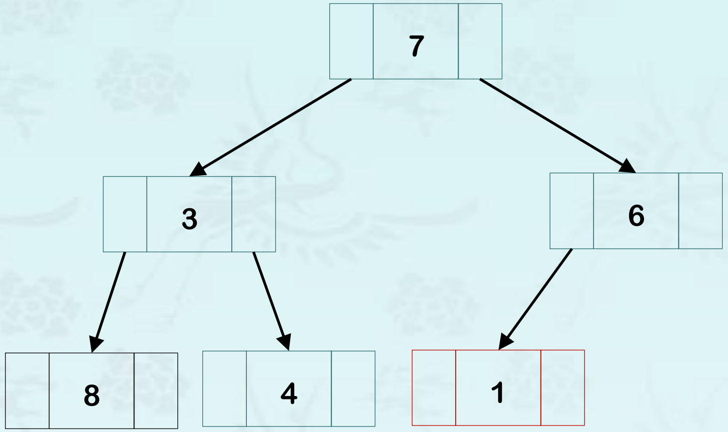
hp->nodes[] hp->N

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 dequeue) it.
   E. enqueue the new node (to add children later if any).
4. treeprint(root)

heapprint() – build a binary tree from heap/CBT 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 dequeue) it.
   - E. enqueue the new node (to add children later if any).
4. treeprint(root)

Data Structures
Chapter 5: Heap and Priority Queue

1. Heap & Priority Queue
2. Heapsort
3. **Heap & PQ Coding**