Heaps:

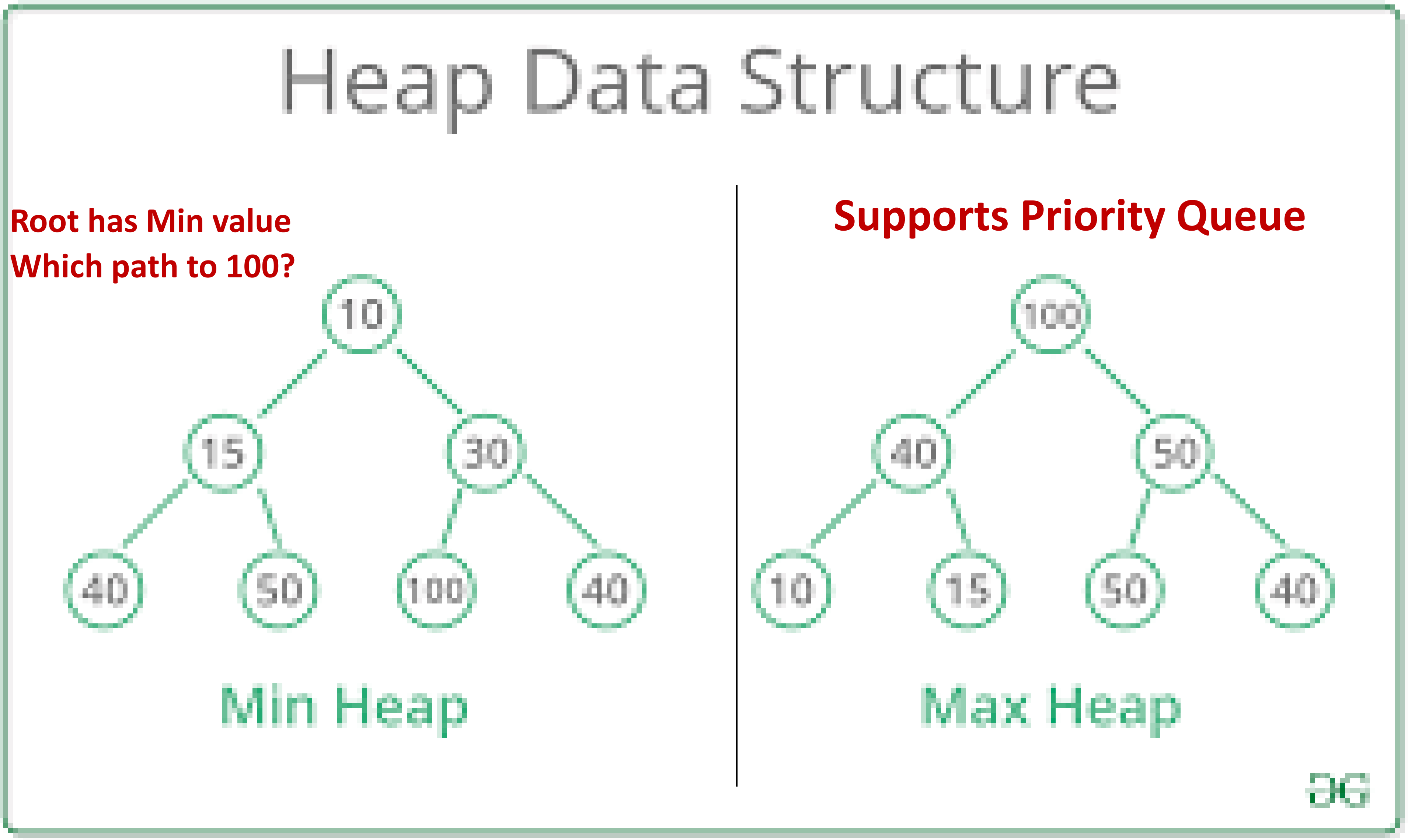
Heap Data Abstraction

- **Based upon Binary Tree Data Structure (like BST)**
- **Entire subtree under each Root node has either:**
  - All Values >= Root (**min** heap) → <u>Both</u> children are >=
  - All Values <= Root (**max** heap) → <u>Both</u> children are <=
- **Is a "Complete" Binary Tree**
  - Always balanced
  - Each level but last has max # of nodes.
  - Last layer filled left → right.
- **Differences with balanced AVL**
  - Unordered (no binary searches for matching key) **Why?**
  - Duplicate Key values allowed

Max & Min Heaps

Root has Min value
Which path to 100?

Supports Priority Queue



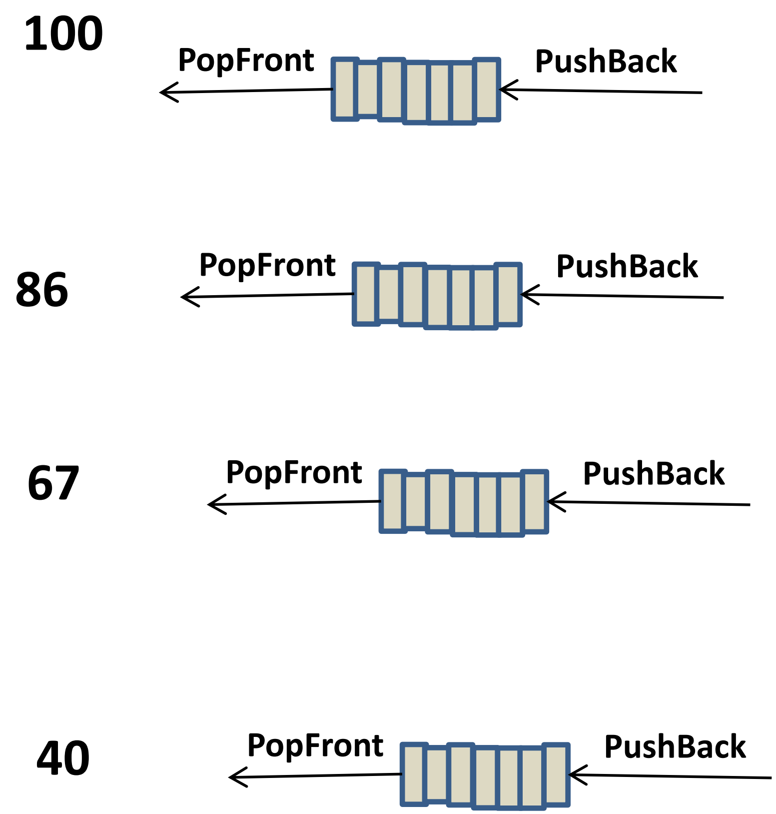
Priority Queue Data Abstraction

The next element selected is always at the front of the highest priority queue.

Ex: No element with priority 86 selected until <u>all</u> elements with priority 100 have been "serviced" (Priority 100 Queue is empty).

All elements in a queue have the same priority ➔ Heaps are "Bags".

Priority



Example of use: Student registration. Each student assigned a priority (Seniors, Continuing, Entering, HS) and assigned to that registration queue.

De Anza Registration

Priority Registration Groups

Student will be assigned registration dates in the following order.

GROUP 1

Veterans, Foster Youth, DSPS, EOPS and CalWorks students
who have completed orientation, assessment and an educational plan

GROUP 2

Student athletes who have
Selected an educational goal of transfer, degree or certificate
Declared a major and have not been on probation for 2 consecutive terms

GROUP 3

Continuing students who have
Selected an educational goal of transfer, degree or certificate
Declared a major and have not been on probation for 2 consecutive terms
Completed orientation, assessment and an educational plan

GROUP 4

New college students who have
Completed assessment, orientation and an educational plan
Selected an educational goal of transfer, degree or certificate
Declared a major

GROUP 5

New college students who have
Selected an educational goal of transfer, degree or certificate
Declared a major and have not been on probation for two consecutive terms
But have **NOT** completed assessment, orientation or an educational plan

GROUP 6

Returning students and new transfer students who have
Selected an educational goal of transfer, degree or certificate
Declared a major

GROUP 7

All other college students who have **NOT** declared a major
or selected an educational goal of transfer, degree or certificate

GROUP 8

Dual or concurrently enrolled high school students

Heap Element Addition / Removal

Insert node into max-heap inserts the node in the tree's bottom level, and then swaps the node with its parent, and continues until no max-heap violation occurs.

Inserts fill a level (left-to-right) before adding another level, so the Heap Tree's height is always the minimum possible (balanced). The upward movement of a node in a max-heap is called ***percolating***.

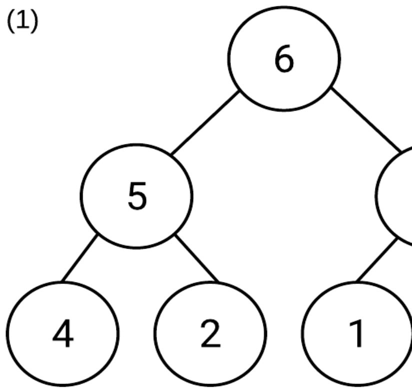
A ***Remove*** from a max-heap is always a removal of the root.

1. Make bottom level's rightmost node ➔ Root
2. Recursively swap node with its greatest child until no max-heap property violation occur.

Max Heap Insertion

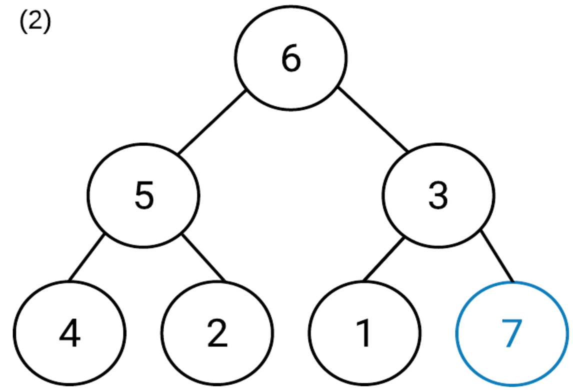
Inserting 7 into this heap

(1)



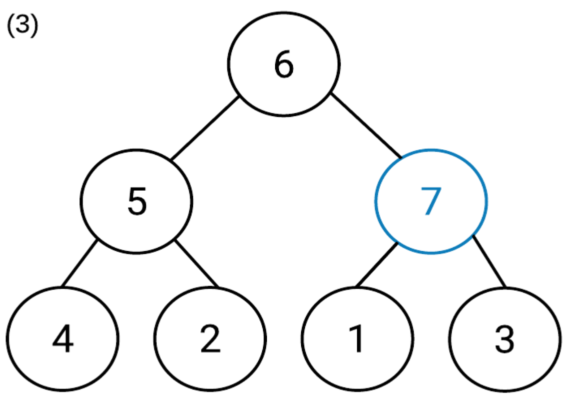
Starting with this max heap

(2)



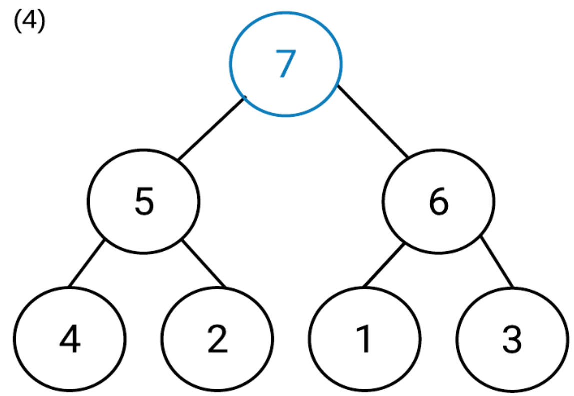
Step 1: 7 is inserted at the bottom most, right most position

(3)



Step 2: Because 7 is bigger than its parent, the 3 node, it gets swapped

(4)



Step 3: Once again, 7 is bigger than its parent, the 6 node, so it gets swapped

Max Heap Removal
(Root removed / last element “percolated”)

10
5 8
2 4 3 1

→ X
5 8
2 4 3 1

→ 1
5 8
2 4 3

Always swap with larger child

8
→ 5 1
2 4 3

→ 8
5 3
2 4 1

Heap Node: Underlying BST Structure

```
struct hNode {
  hNode* parent;   // Parent
  hNode* lhc;   // Left Hand Child
  hNode*rhc;   // Right Hand Child

  int priority;    // Priority of Node (~ the BST “key”)
  Thing* thing;  // Ptr to Thing collected in Heap
};
```

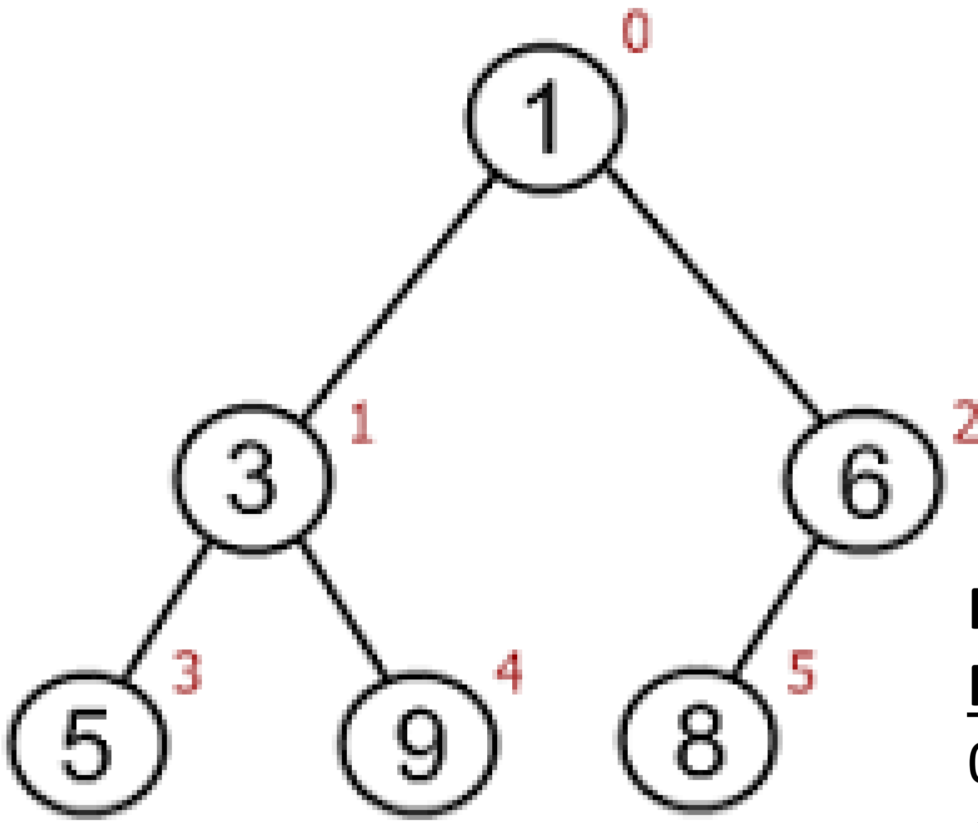
➔ **Looks a LOT like BST node.**

Max / Min Heap API

```
class maxHeap {  // Collection is max Heap.  Element 0 is Root
  private:
     hNode** ha; // Ptr to contiguous array of hNode ptrs
                 // 0th element is Heap root (maximum)
     int hsz;  // Current size of the array (and 1 past last node)
     int hax;  // Current index in ha Array
  public:
     void push (hNode* hn); // Put at bottom, percolates
     hNode*  pop(); // Pops root & repositions lower nodes
                    // Size adjusted (so last node known)
     hNode*  look();  // Peeks at Root Node
};
```

➔ **Looks like a Fifo Queue … except when inserted, an element moves to correct priority level within Heap**

Min Heap to Array Mapping!!



Position Mapping:

P	**Cs**
0 →	1, 2
1 →	3,4
2 →	5

Value	1	3	6	5	9	8
Index	0	1	2	3	4	5

N → 2N+1, 2N+2

Computing Heap Array Indices

Node index	Parent index	Child indices
0	N/A	1, 2
1	0	3, 4
2	0	5, 6
3	1	7, 8
4	1	9, 10
5	2	11, 12
…	…	…
i	$\lfloor (i-1)/2 \rfloor$	2 * i + 1, 2 * i + 2

- ➔ Node Index 57: Parent = 56/2 = 28. Children 115, 116
- ➔ Node index 58; Parent = 57/2 = 28, Children 117, 118

Tree Heap Node → Array Heap Item

```
struct hNode {     // Used in Tree
  hNode* parent;   // Parent
  hNode* lhc;   // Left Hand Child
  hNode*rhc;   // Right Hand Child
  int priority;    // Priority of Node (~ the Tree Node “key”)
  Thing* thing;  // Ptr to Thing represented in Heap
};
```

becomes:

```
struct hItem {    // Used in Array
  int priority;    // Priority of Node (~ the Array Item “key”)
  Thing* thing;  // Ptr to Thing represented in Heap
};
```

→ How is this possible?

Orderable hItem* Array

```
class EHI {  // Enhanced Array of Heap Item Pointers
  public:
     EHI (int initSz); // Create internal hItem* Array of 2X that Size
     EHI (ESI&) (); ~EHI();  // Copy constructor & Destructor
     unsigned int getNum ();  // Size of Array
     hItem* get (int index); // Get elem at index
                // Return -1 if specified index illegal
    int set (int index, hItem*ep); // Overwrite existing elem at index
    int append (hItem* ep); // Append to back, resize if needed
    Int prepend (hItem* ep); // Prepend to front, resize if needed
    int remove (int index);  // Remove hItem* elem, move others down
    int insert (int index, hItem *ep); // Insert hItem* elem
    };                         // Push others back.  Auto Resize if needed
```

Max Heap "Array" API (Template unchanged!!)

```
class maxHeap {  // Collection is max Heap.  Element 0 is Root
  private:
      EHI* ehi; // Ptr to Orderable array (of hNode Item ptrs)
                        // 0th element is Heap root (maximum)
      int hmax;  // Current size of the EHI (from EHI::getNum())
      int hidx;  // Current index to EHI elements
 public:
      int push (hItem* hi); // Push to bottom, percolate up
      hItem*  pop(); // Pops root & repositions lower nodes
                        // Size adjusted (so last node known)
      hItem*  look();  // Peeks at Root Node
};
```

➔ **Underlying Data Structure is an implementation detail**

➔ **Where are the inefficiencies?**

Max Heap Array: Append & Percolate Up

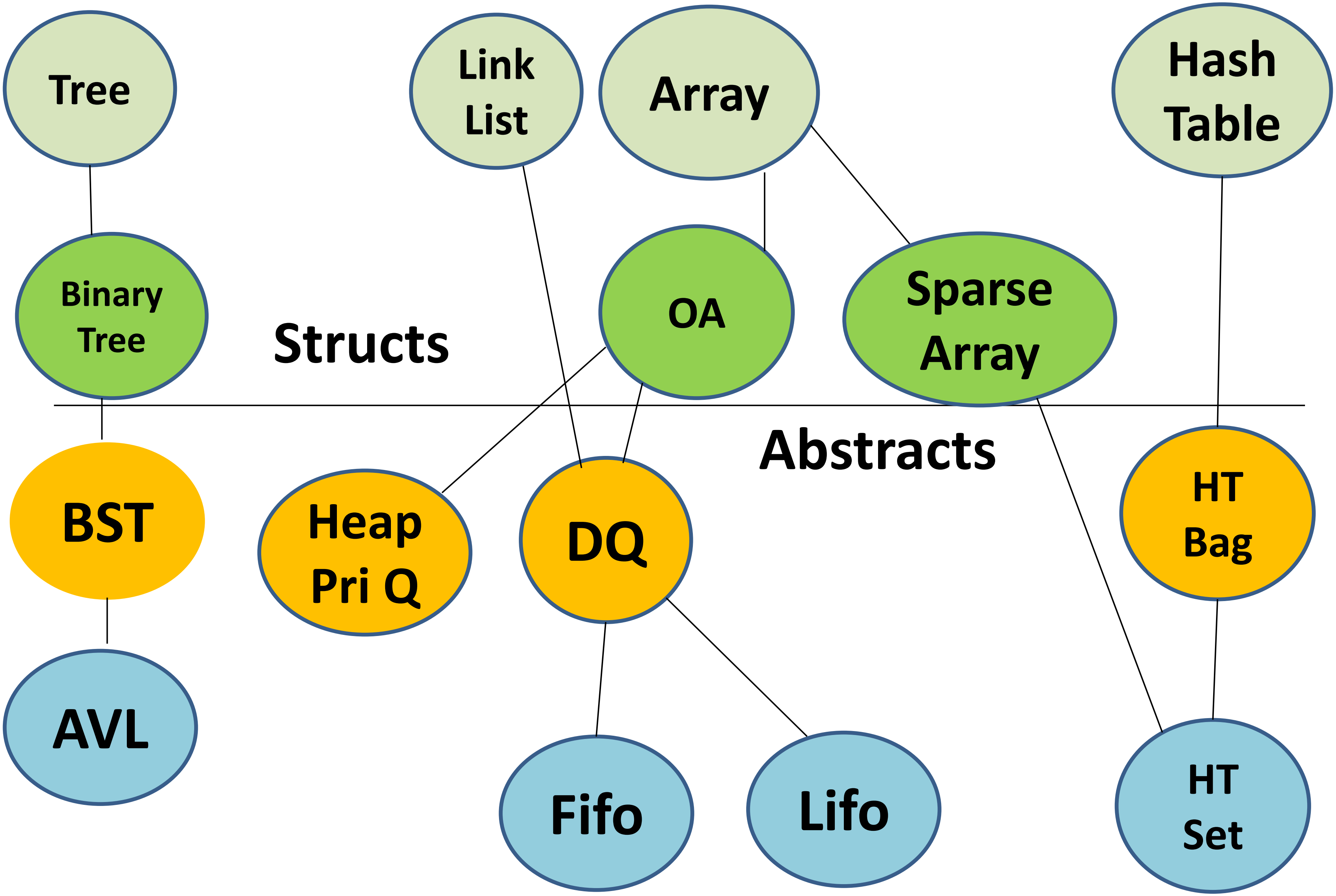
```
// Add new item to heap array
// Rebalance and return position of where new item is
      Append current item to back  //Perculate
      While (entry !=root)    // Check if priority > parent
            Get parent index
            Get ptr to parent item from EA
            If (new priority > parent)  // Swap
                 Store appended item in parent index
                 Store parent item in current index of appended item
                 Set current item index to parent index
            else
                 break   // Node in correct position
        Return current item final index // Node percolated
```

Max Heap Array: Append & Percolate Up

```
int maxHeap:: push (hItem* hp) {  // Add new item to heap array
        // Rebalance and return position of where new item is
   int pidx = 0; // Parent item index
   hItem*pp = null; // Holding item ptr of parent
   hidx = ehi->append(hp);//Append current element to back: perculate
    while  hidx > 0   { // While entry !=root, check if priority > parent
      pidx = (hidx -1) /2;  // Get parent index
      pp=ehi->get(pidx); // Get ptr to parent item
      if ((hp->priority) > (pp->priority)) { // New priority > parent. Swap
          ehi->set (pidx, hp);
          ehi->set (hidx, pp);
          hidx = pidx;  // Set current index to parent index, repeat
       }
       else  break; // It isn't greater.  Node in correct position
   }  return (hidx);  // Node percolated up to top (new max)
};
```

➔ **Extra credit. Do maxheap "pop"**

Relationship of Collections



Questions?

Heapify & Heap Sort

- **Heapify unsorted array ➔ max Heap Array**

  The heapify operation is used to turn an array into a heap. Since leaf nodes already satisfy the max heap property (higher than stubs), building a max-heap is achieved by **percolating down on every non-leaf node in reverse order**.

- **Heapsort max Heap Array ➔ Sorted "min" Array**

  Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order.

➔ **Left as an exercise for the reader ☺**

Treaps: Left for the Reader

A BST built from inserts of N nodes having random-ordered keys stays well-balanced and thus has near-minimum height, meaning searches, inserts, and deletes are O(logN). Because insertion order may not be controllable, a data structure that somehow randomizes BST insertions is desirable.

A ***treap*** uses a main key that maintains a binary search tree ordering property, and a secondary key generated randomly (often called "priority") during insertions that maintains a heap property. The combination usually keeps the tree balanced. The word "treap" is a mix of tree and heap. This section assumes the heap is a max-heap. Algorithms for basic treap operations include:

A treap ***search*** is the same as a BST search using the main key, since the treap is a BST.

A treap ***insert*** initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated. In a heap, a node is moved up via a swap with the node's parent. In a treap, a node is moved up via a *rotation at the parent*. Unlike a swap, a rotation maintains the BST property.

A treap ***delete*** can be done by setting the node's priority such that the node should be a leaf (-∞ for a max-heap), percolating the node down using rotations until the node is a leaf, and then removing the node.