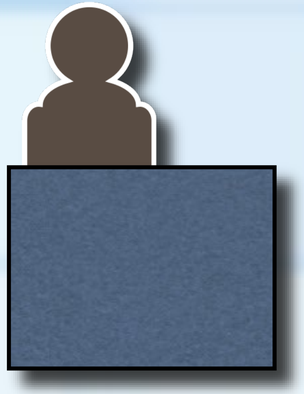
The ADT Priority Queue

THE ADT PRIORITYQUEUE

- **Removing an item always removes the highest priority item**
  - How priority is determined is implementation dependent
- **Highest priority item is at front of queue**
  - Order of other items is not important and depends on how the priority queue is implemented



dequeue()

front

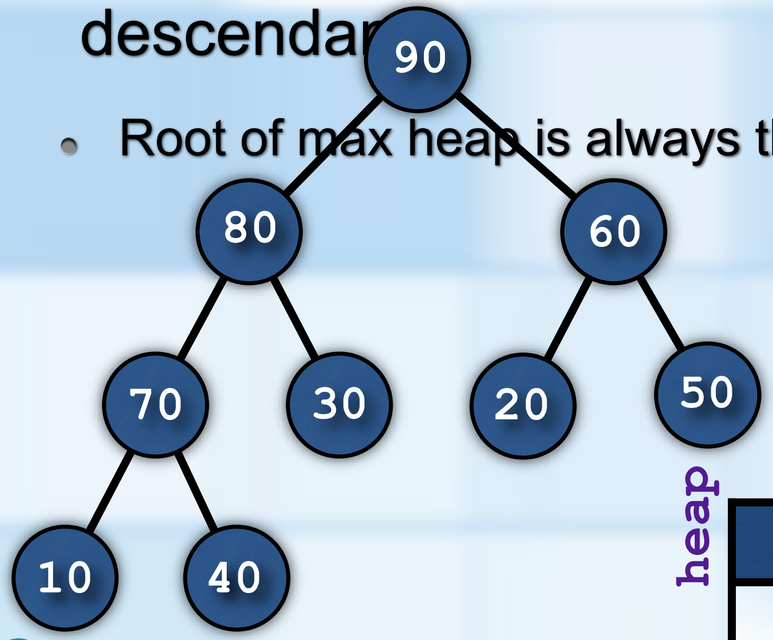
THE ADT PRIORITYQUEUE

```
template<class ItemType>
class PriorityQueueInterface
{
public:
   /** Adds a new entry to this queue. */
   virtual bool add(const ItemType& someItem) = 0;

   /** Removes high priority item from this queue. */
   virtual bool dequeue() = 0;

   /** Returns high priority item from this queue. */
   virtual ItemType peek() const = 0;

  /** Sees whether this queue is empty. */
   virtual bool isEmpty() const = 0;
}; // end PriorityQueueInterface
```

THE ADT HEAP

HEAPS

- **A complete binary tree whose nodes contain objects that can be compared**
  - Each node contains an object that is larger than the objects in its descendants
    - Root of max heap is always the largest entry




```
template<class ItemType>
class HeapInterface
{
public:
  /** Sees whether this heap is empty. */
  virtual bool isEmpty() const = 0;
  /** Gets the number of nodes in this heap. */
  virtual int getNumberOfNodes() const = 0;
  /** Gets the height of this heap. */
  virtual int getHeight() const = 0;
  /** Gets data in root (top) of this heap. */
  virtual ItemType peekTop() const = 0;
  /** Adds a new node containing to this heap. */
  virtual bool add(const ItemType& someItem) = 0;
  /** Removes the root node from this heap. */
  virtual bool remove() = 0;
  /** Removes all nodes from this heap. */
  virtual void clear() = 0;
  /** Destroys object & frees memory */
  virtual ~HeapInterface() { }
}; // end HeapInterface
```

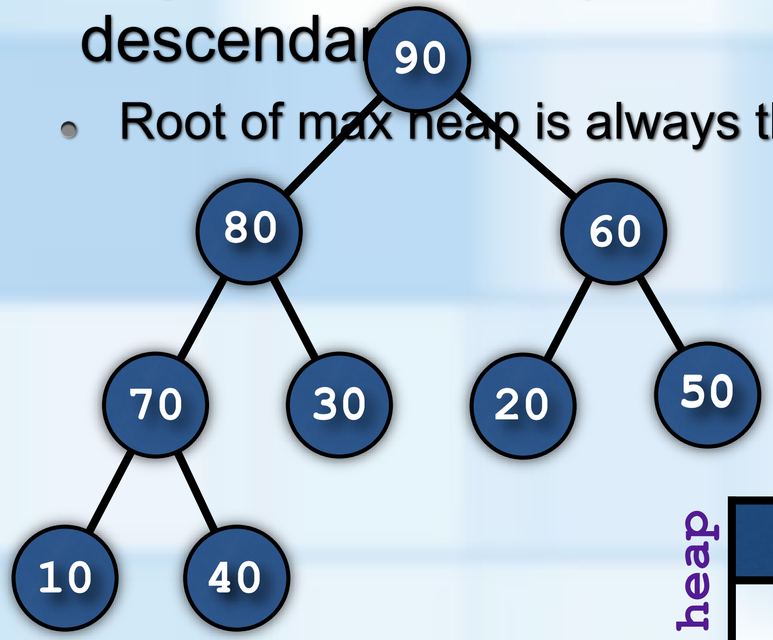
heap

0	1	2	3	4	5	6	7	8	9	10

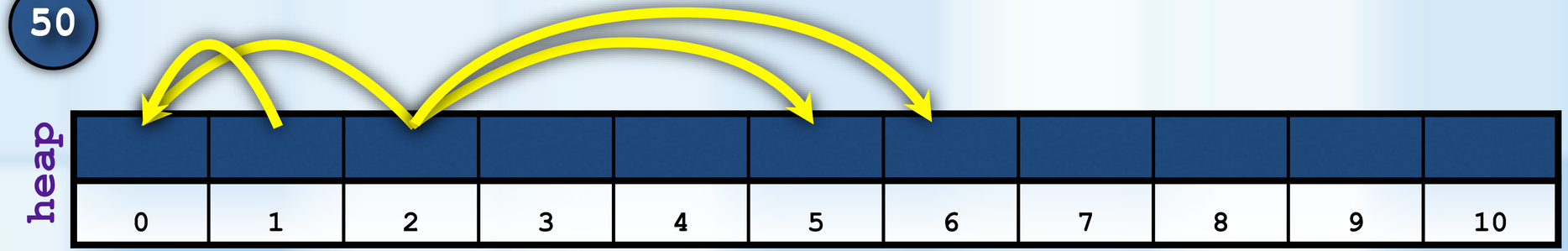
Pearson

HEAPS

- **A complete binary tree whose nodes contain objects that can be compared**
  - Each node contains an object that is larger than the objects in its descendants
    - Root of max heap is always the largest entry



- **Using an array to represent a Heap**
  - Root is stored in element `[0]`
  - Index of left child is `2 * i + 1`
  - Index of right child is `2 * (i + 1)`
  - Parent is at index `(i - 1) / 2`
  - Array represents the level order traversal of the heap (a complete tree)

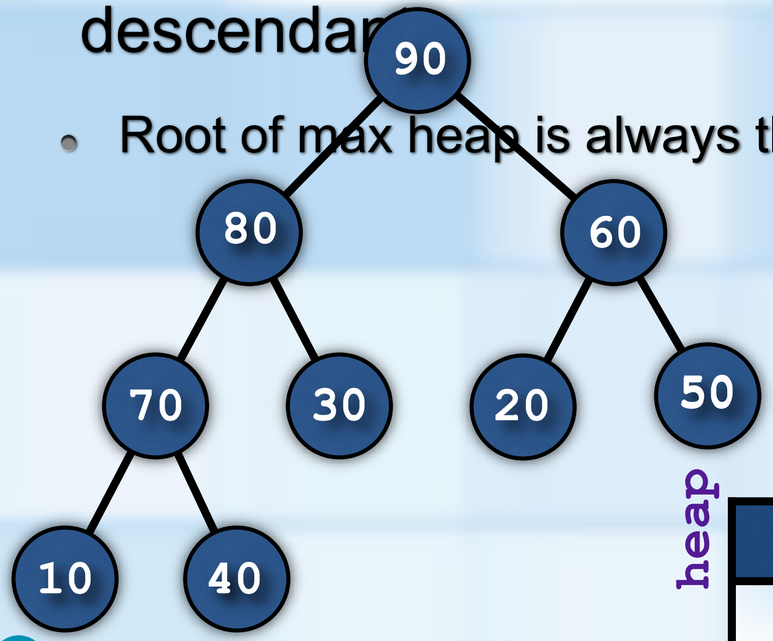


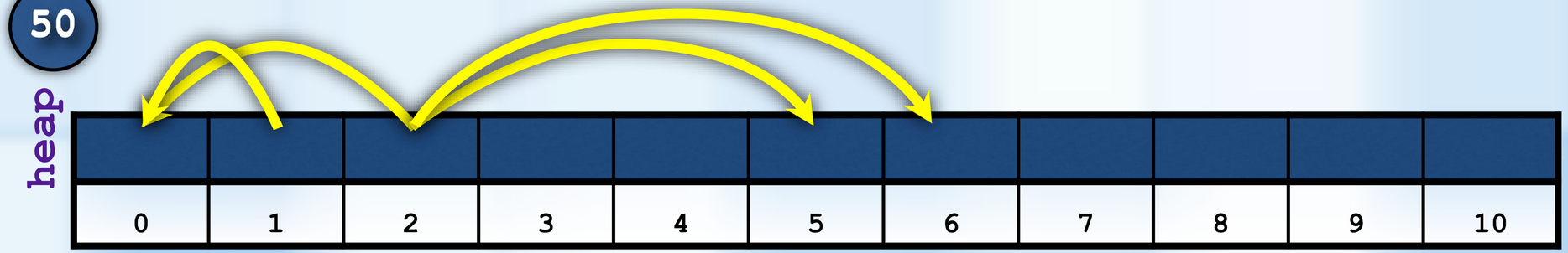
Pearson

HEAPS

- **A complete binary tree whose nodes contain objects that can be compared**
  - Each node contains an object that is larger than the objects in its descendants
    - Root of max heap is always the largest entry

```
template<class ItemType>
class ArrayMaxHeap : public HeapInterface<ItemType>
{
private:
   // Helps with readability
   static const int ROOT_INDEX = 0;
    // Small capacity to test for a full heap
   static const int DEFAULT_CAPACITY = 21;
   // Array of heap items
     ItemType* items;
   // Current count of heap items
   int itemCount;
```

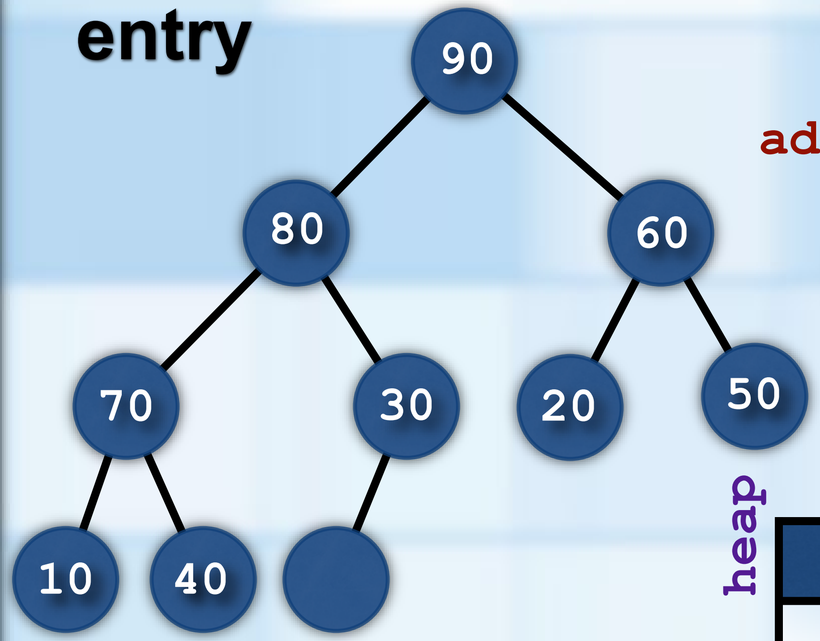




Pearson

Adding To a Heap

- Add a node at next location to keep a complete tree
- Compare entry to parent
  - Swap with parent if entry is larger
- Stop when parent is larger than entry



add(85)

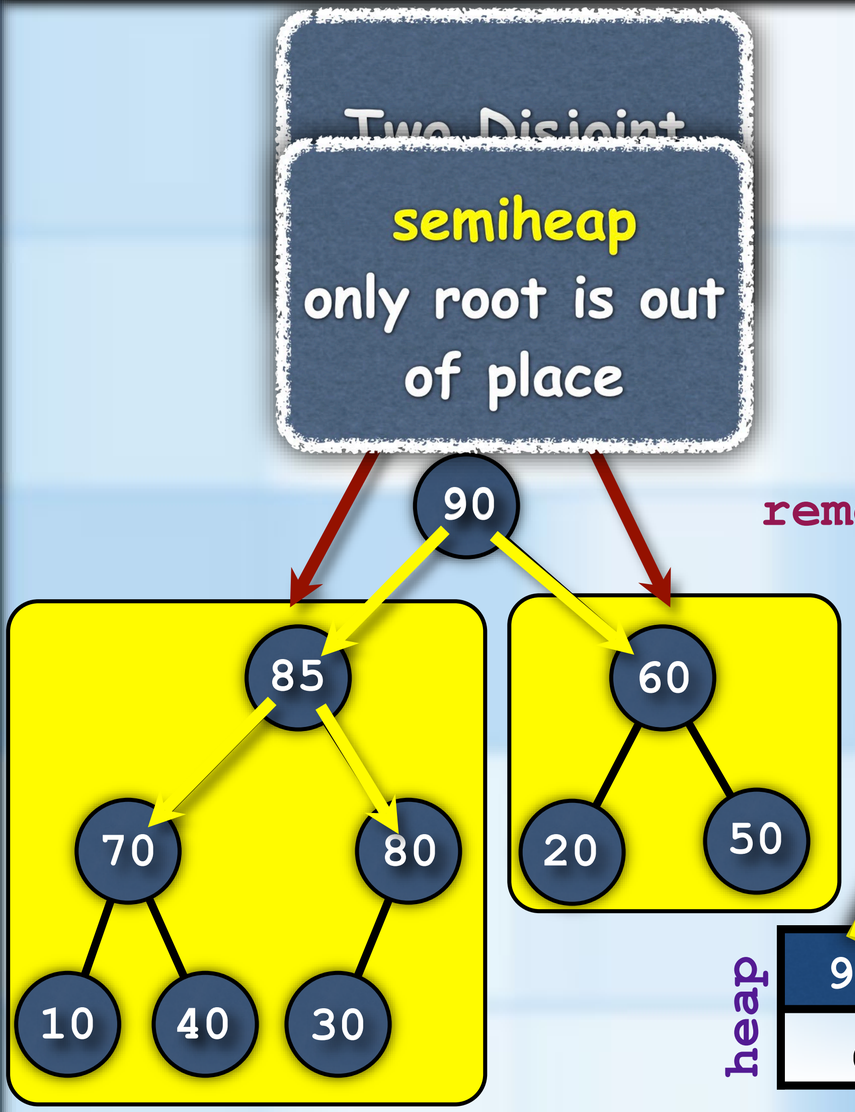
```
template<class ItemType>
bool ArrayMaxHeap<ItemType>::add(const ItemType& someItem)
{
  bool isSuccessful = false;
  if (itemCount < DEFAULT_CAPACITY)
  {
    items[itemCount] = someItem;

    bool inPlace = false;
    int someItemIndex = itemCount;
    while ((someItemIndex > 0) && !inPlace)
    {
      int parentIndex = (someItemIndex - 1) / 2;
      if (items[someItemIndex] < items[parentIndex])
      {
        inPlace = true;
      }
      else
      {
        std::swap(items[someItemIndex], items[parentIndex]);
        someItemIndex = parentIndex;
      } // end if
    } // end while

    itemCount++;
    isSuccessful = true;
  } // end if

  return isSuccessful;
} // end add
```

heap

90	80	60	70	30	20	50	10	40	85	
0	1	2	3	4	5	6	7	8	9	10

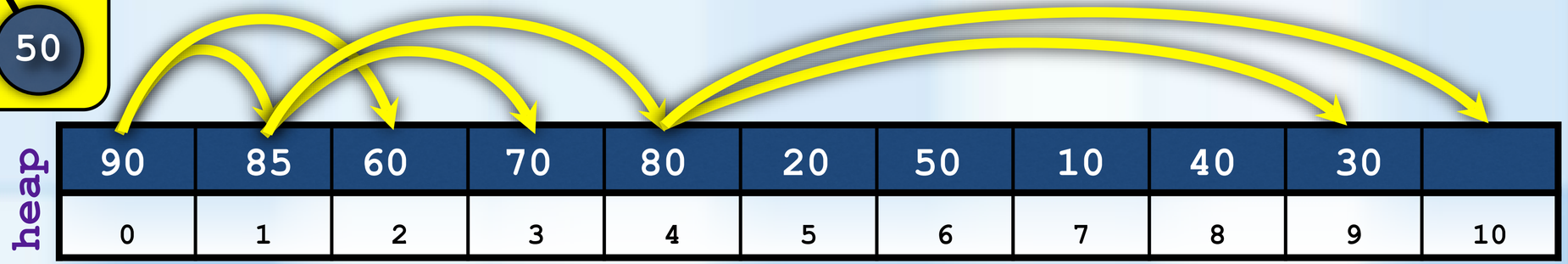
Pearson

Removing From a Heap



`removeMax()`

- **Remove entry at root**
- **Replace with entry in last node**
- **Trickle down value to correct position (`reheap` or `rebuildHeap`)**
  - If at least one child has a larger entry
    - Swap with larger child
  - Stop when
    - entry is larger than both children **or**
    - entry is in a leaf



Pearson

Creating a Heap

- **If beginning with an empty heap**
  - Repeatedly add entries using `add()`
  - *O(n log n)*
- **If given an an array of randomly ordered entries**
  - Repeatedly use `reheap()`
    - beginning with element `[n/2]` and
    - working back to root (element `[0]`)
  - *O(n)*