CMPT 225

Priority Queues and Heaps

Priority Queues

- Items in a priority queue have a priority
  - The priority is usually numerical value
  - Could be lowest first or highest first
- The highest priority item is removed first
- Priority queue operations
  - Insert
  - Remove in priority queue order (not a FIFO!)

Using a Priority Queue

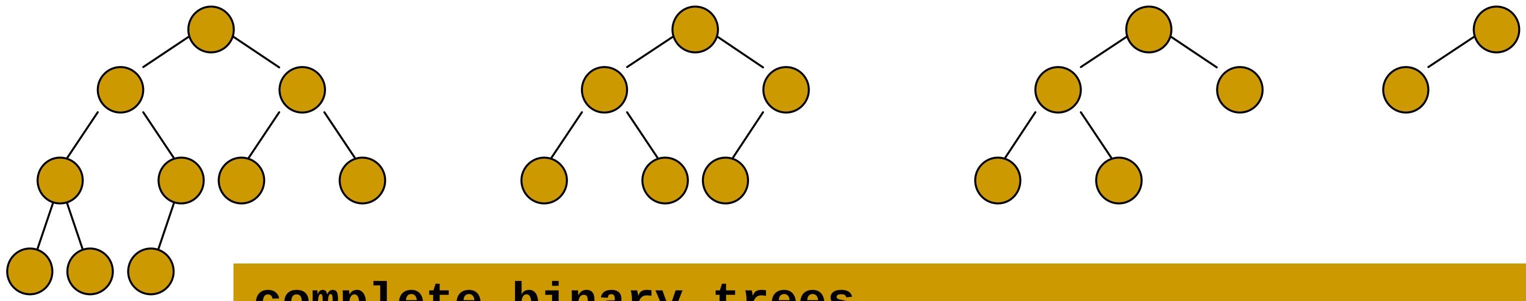
Implementing a Priority Queue

- Items have to be removed in priority order
- This can only be done efficiently if the items are ordered in some way
  - A balanced binary search (e.g., red-black) tree is an efficient and ordered data structure but
    - Some operations (e.g. removal) are complex to code
    - Although operations are O(log *n*) they require quite a lot of structural overhead
- There is another binary tree solution – *heaps*
  - **Note:** We will see that search/removal of the maximum element is efficient, but it's not true for other elements

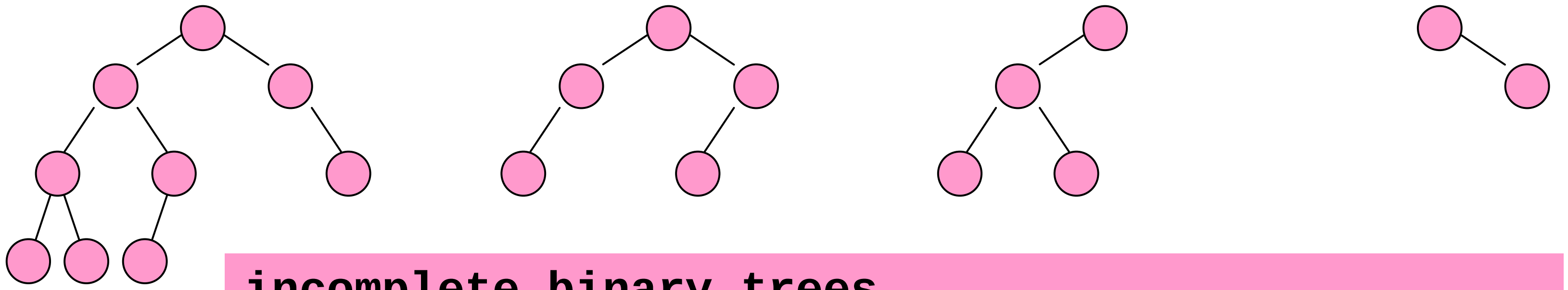
Heaps

- A heap is binary tree with two properties
- Heaps are complete
  - All levels, except the bottom, must be completely filled in
  - The leaves on the bottom level are as far to the left as possible.
- Heaps are partially ordered ("heap property"):
  - **The value of a node is at least as large as its children's values, for a *max heap*** or
  - The value of a node is no greater than its children's values, for a *min heap*

Complete Binary Trees

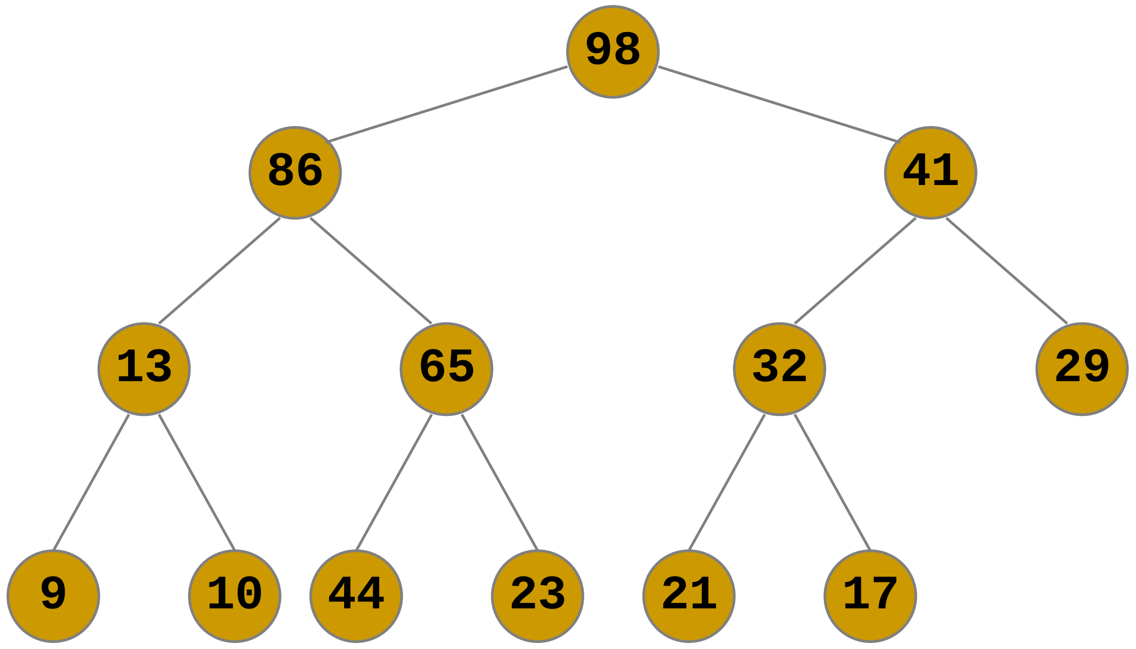


complete binary trees



incomplete binary trees

Partially Ordered Tree – max heap



Note: an inorder traversal would result in:

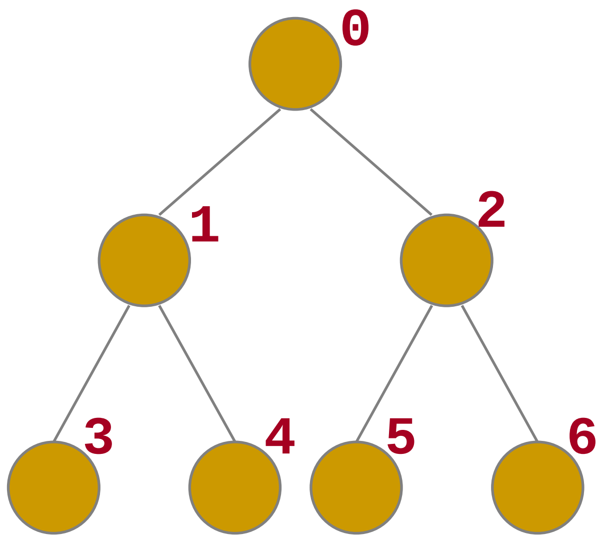
9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 17, 41, 29

Priority Queues and Heaps

- A heap can be used to implement a priority queue
- Because of the partial ordering property the item at the top of the heap must always the largest value
- Implement priority queue operations:
  - Insertions – insert an item into a heap
  - Removal – remove and return the heap's root
  - For both operations preserve the heap property

Heap Implementation

- Heaps can be implemented using arrays
- There is a natural method of indexing tree nodes
  - Index nodes from top to bottom and left to right as shown on the right (by levels)
  - Because heaps are complete binary trees there can be no gaps in the array



Array implementations of heap

```
public class Heap<T extends KeyedItem> {
  private int HEAPSIZE=200;
  // max. number of elements in the heap
  private T items[];        // array of heap items
  private int num_items;    // number of items

  public Heap() {
    items = new T[HEAPSIZE];
    num_items=0;
  }  // end default constructor
```

- We could also use a **dynamic array** implementation to get rid of the limit on the size of heap.
- We will assume that priority of an element is equal to its key. So the elements are partially sorted by their keys. They element with the biggest key has the highest priority.

Referencing Nodes

- It will be necessary to find the indices of the parents and children of nodes in a heap's underlying array
- The children of a node `i`, are the array elements indexed at `2i+1` and `2i+2`
- The parent of a node *i*, is the array element indexed at `floor[(i–1)/2]`

Helping methods

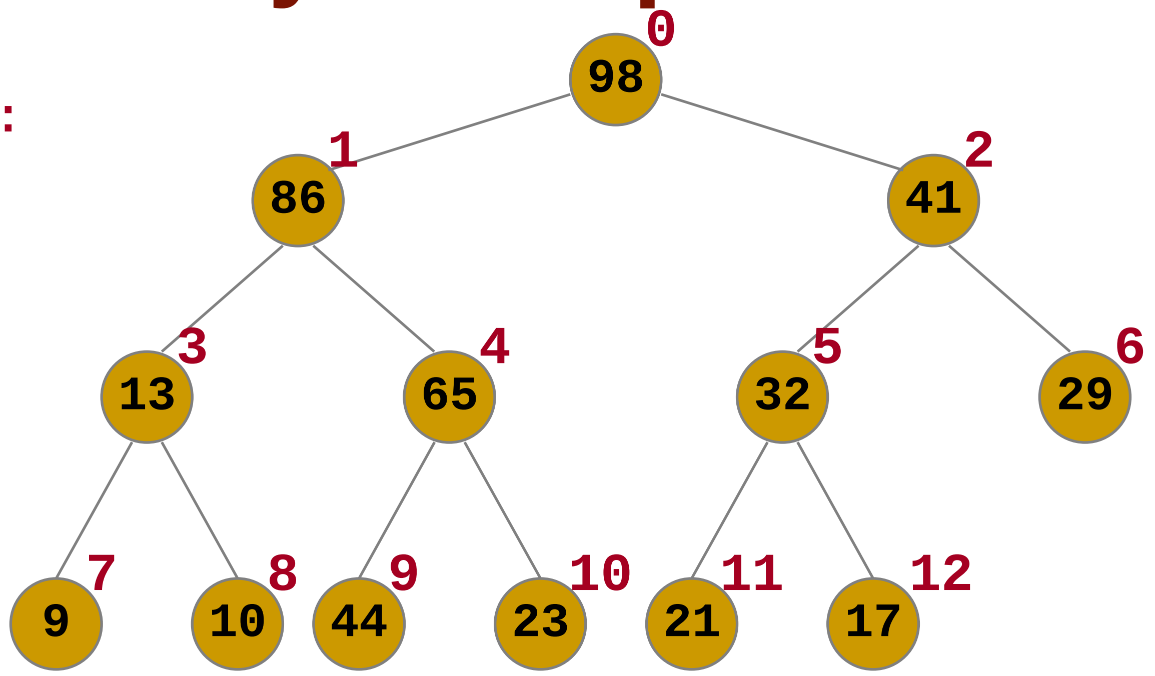
```
private int parent(int i)
{ return (i-1)/2; }

private int left(int i)
{ return 2*i+1; }

private int right(int i)
{ return 2*i+2; }
```

Heap Array Example

Heap:



Underlying Array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Insertion

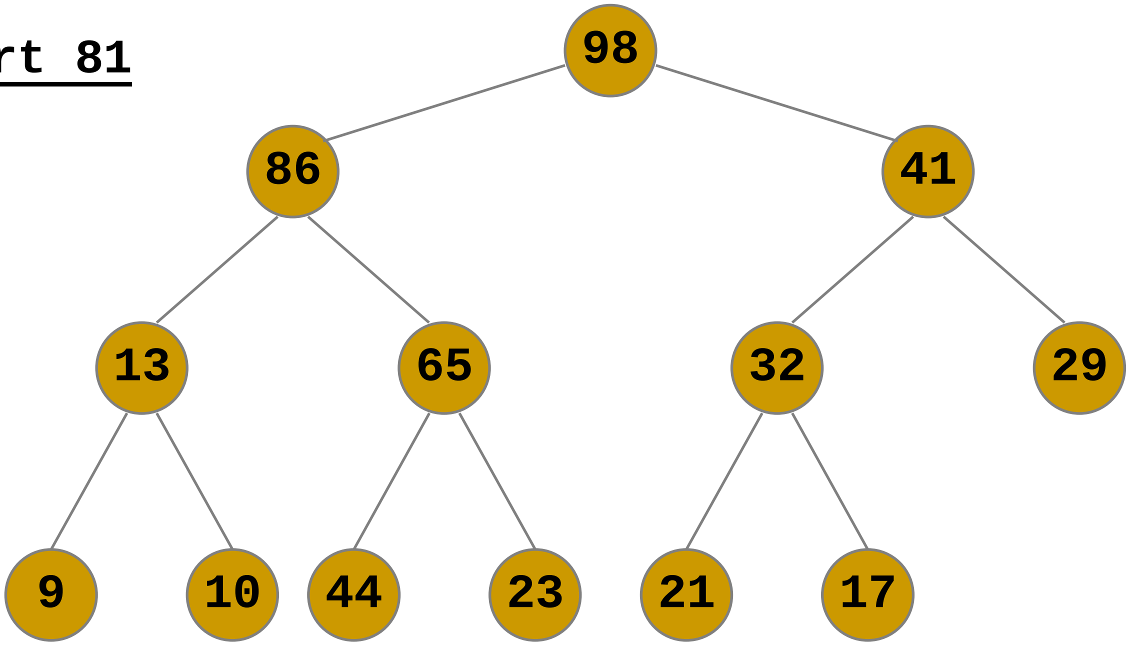
- On insertion the heap properties have to be maintained; remember that
  - A heap is a complete binary tree and
  - A partially ordered binary tree
- There are two general strategies that could be used to maintain the heap properties
  - Make sure that the tree is complete and then fix the ordering, or
  - Make sure the ordering is correct first
  - Which is better?

Heap Insertion algorithm

- The insertion algorithm first ensures that the tree is complete
  - Make the new item the first available (left-most) leaf on the bottom level
  - i.e. the first free element in the underlying array
- Fix the partial ordering
  - Repeatedly compare the new value with its parent, swapping them if the new value is greater than the parent (for a max heap)
  - Often called “bubbling up”, or “trickling up”

Heap Insertion Example

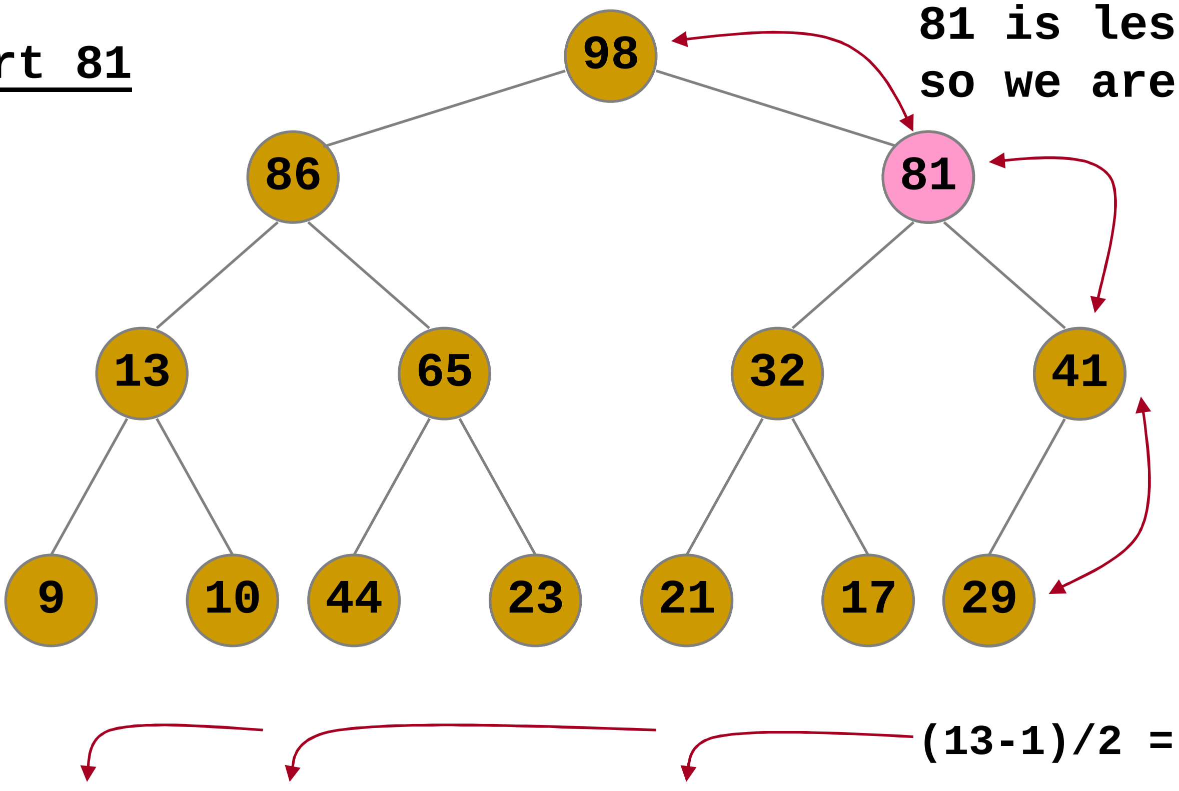
Insert 81



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	41	13	65	32	29	9	10	44	23	21	17	

Heap Insertion Example

Insert 81



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	81	13	65	32	41	9	10	44	23	21	17	29

Heap Insertion algorithm

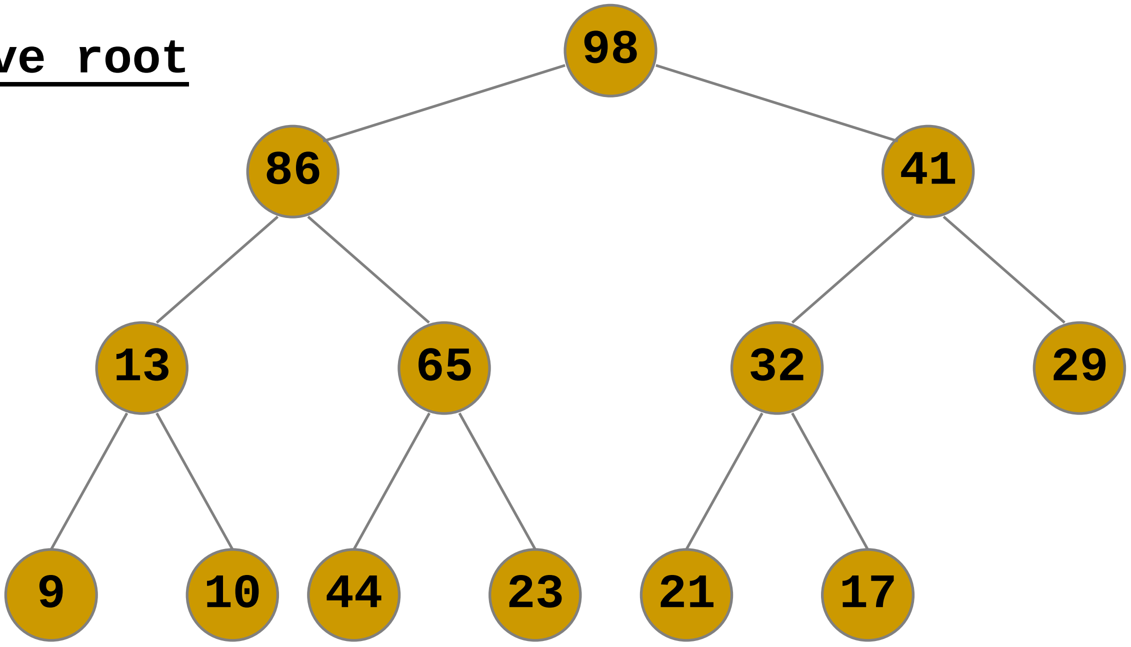
```
public void insert(T newItem) {
  // TODO: should check for the space first
  num_items++;
  int child = num_items-1;
  while (child > 0 &&
         item[parent(child)].getKey() <
         newItem.getKey()) {
    items[child] = items[parent(child)];
    child = parent(child);
  }
  items[child] = newItem;
}
```

Heap Removal algorithm

- Make a temporary copy of the root’s data
- Similar to the insertion algorithm, ensure that the heap remains complete
  - Replace the root node with the right-most leaf on the last level
  - i.e. the highest (occupied) index in the array
- Repeatedly swap the new root with its largest valued child until the partially ordered property holds
- Return the root’s data

Heap Removal Example

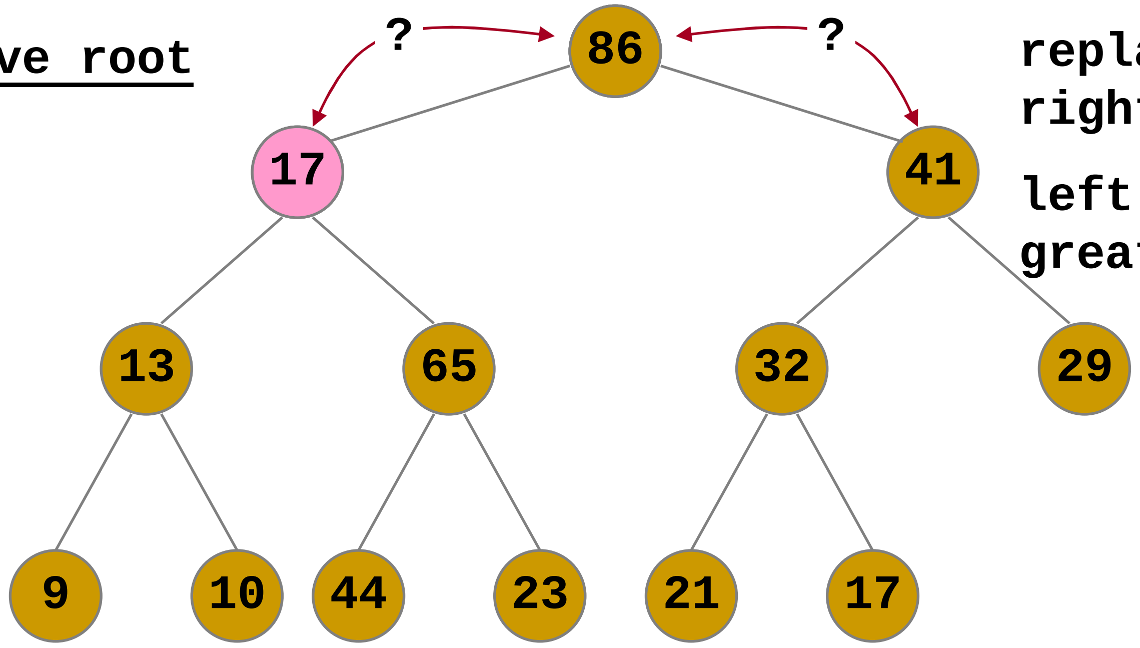
<u>Remove root</u>



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Removal Example

Remove root



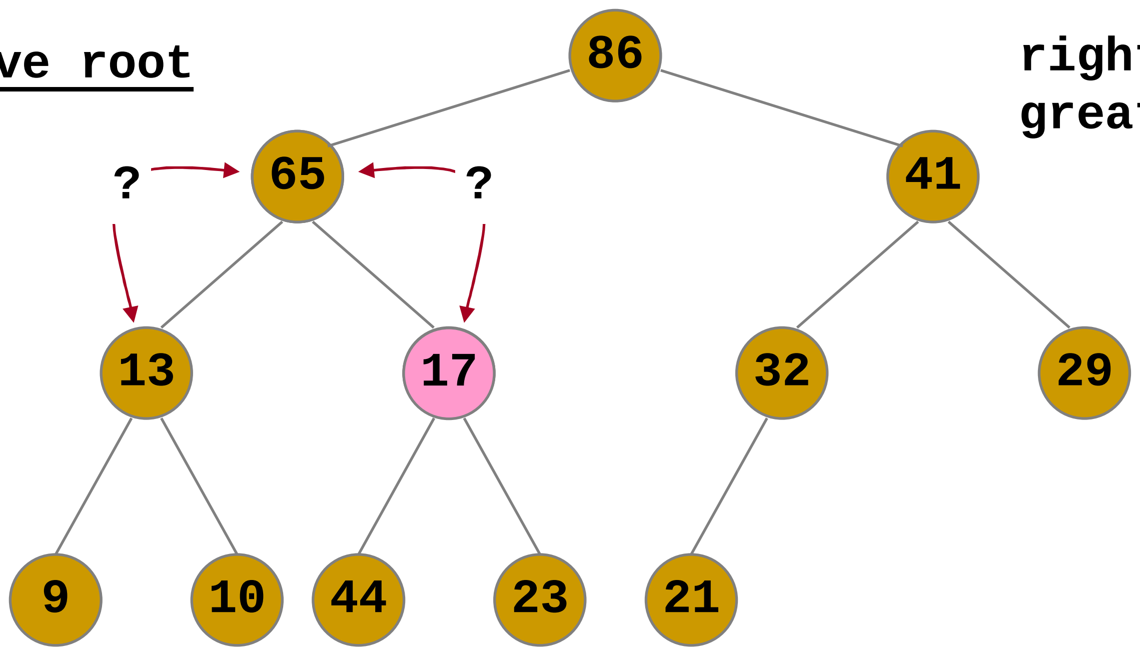
replace root with right-most leaf

left child is greater

children of root: 2*0+1, 2*0+2 = 1, 2

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	17	41	13	65	32	29	9	10	44	23	21	

Heap Removal Example



right child is greater

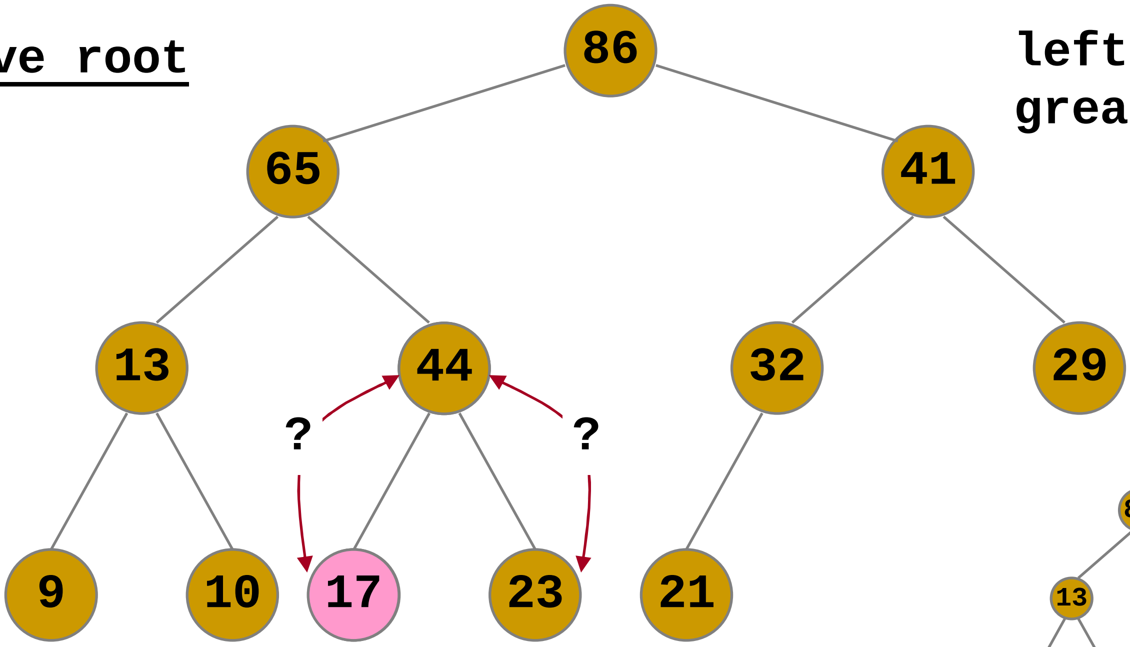
children: 2*1+1, 2*1+2 = 3, 4

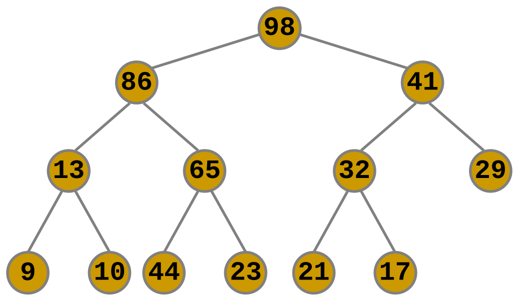
index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	17	32	29	9	10	44	23	21	

Heap Removal Example

Remove root

left child is greater





children: 2*4+1, 2*4+2 = 9, 10

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	44	32	29	9	10	17	23	21	

Heap Removal algorithm

```
public T remove()
  // remove the highest priority item
  {
    // TODO: should check for empty heap
    T result = items[0]; // remember the item
    T item = items[num_items-1];
    num_items--;
    int current = 0; // start at root
    while (left(current) < num_items) { // not a leaf
      // find a bigger child
      int child = left(current);
      if (right(current) < num_items &&
          items[child].getKey() < items[right(current)].getKey()) {
        child = right(current);
      }
      if (item.getKey() < items[child].getKey()) {
        items[current] = items[child];
        current = child;
      } else
        break;
    }
    items[current] = item;
    return result;
  }
```

Heap Efficiency

- For both insertion and removal the heap performs at most *height* swaps
  - For insertion at most *height* comparisons
  - For removal at most *height**2 comparisons
- The height of a complete binary tree is given by $\lfloor \log_2(n) \rfloor + 1$
  - Both insertion and removal are O(log*n*)

Remark: but removal is only implemented for the element with the highest key!