CS2040 Data Structures and Algorithms
Lecture Note #10

Priority Queue and Heap

Examples

- A “to-do” list with priorities
- Scheduling jobs in OS

- Go to Takashimaya (6)
- Work out in gym (5)
- Prepare CS2040 lecture slides (1)
- Go to department tea (4)
- …

Priority queue operations

- Insert an item with given key
- Remove the item with maximum key

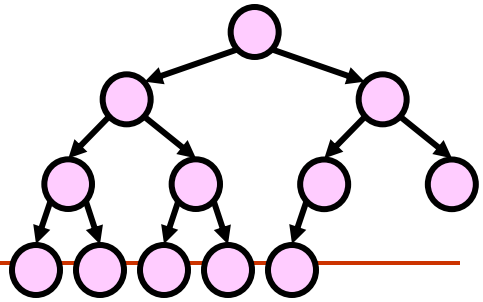
Unsorted list implementation

- **Insertion**: add the element to end of a list O(1)
- **Deletion**: traverse the list to find the element of maximum key and remove it O(n)

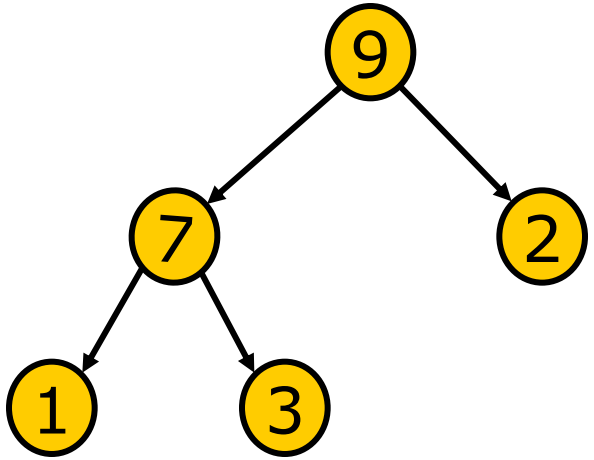
What are the running times if we use **sorted** list?

Heap

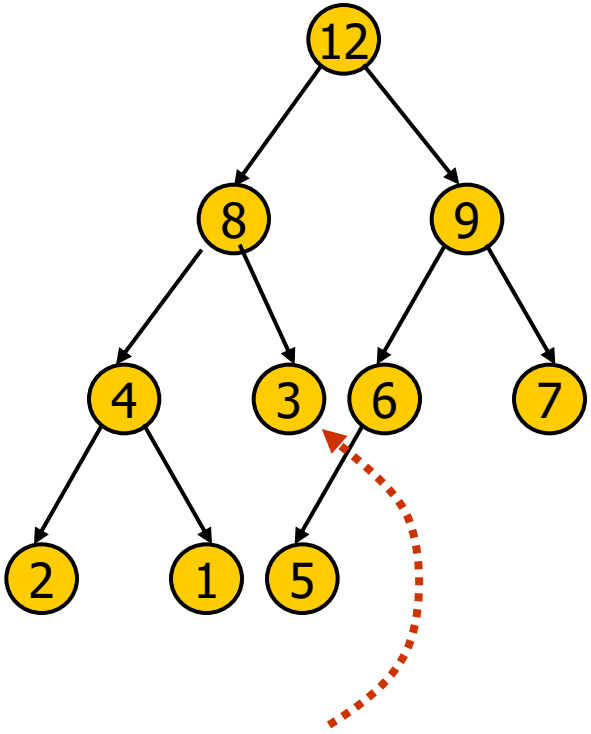
Definition



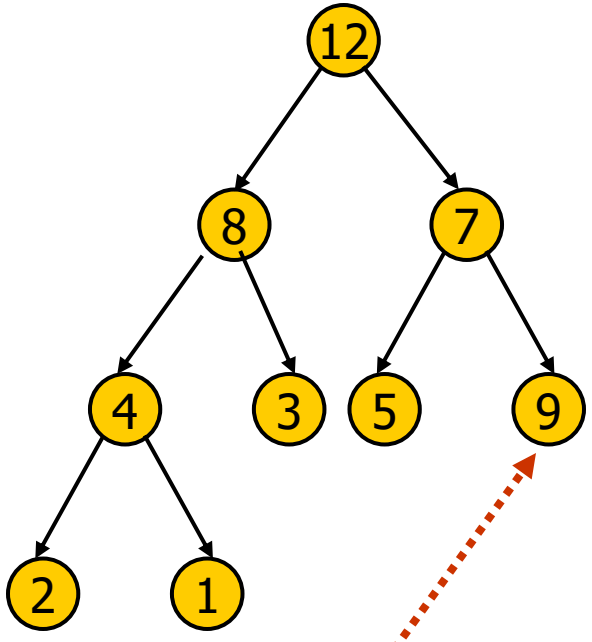
- A (binary) **heap** is a complete binary tree
  - either is empty,
  - or satisfies the **heap property**: for every node *v*, the search key in *v* is greater or equal to those in the children of *v*.



Negative examples



not complete



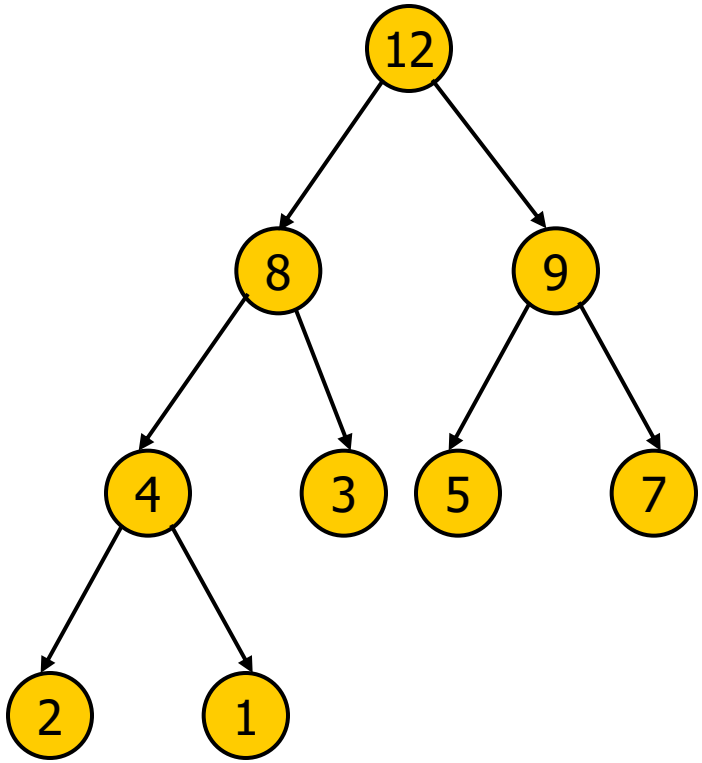
fail heap property

Note: usually we talk about a *max* heap (root is largest element)

Compare heap with BST

- Both are binary trees.
- Difference
  - Heap maintains **heap property**.
  - BST maintains **BST property**.

Representation using arrays



0	12
1	8
2	9
3	4
4	3
5	5
6	7
7	2
8	1

left(i) = 2*i+1

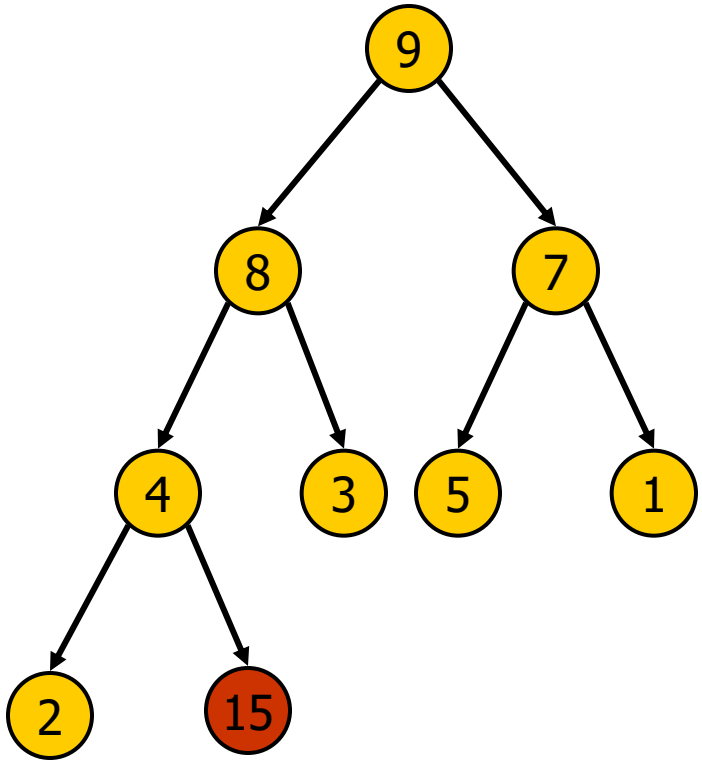
right(i) = 2*i+2

parent(i) =

floor((i-1)/2)

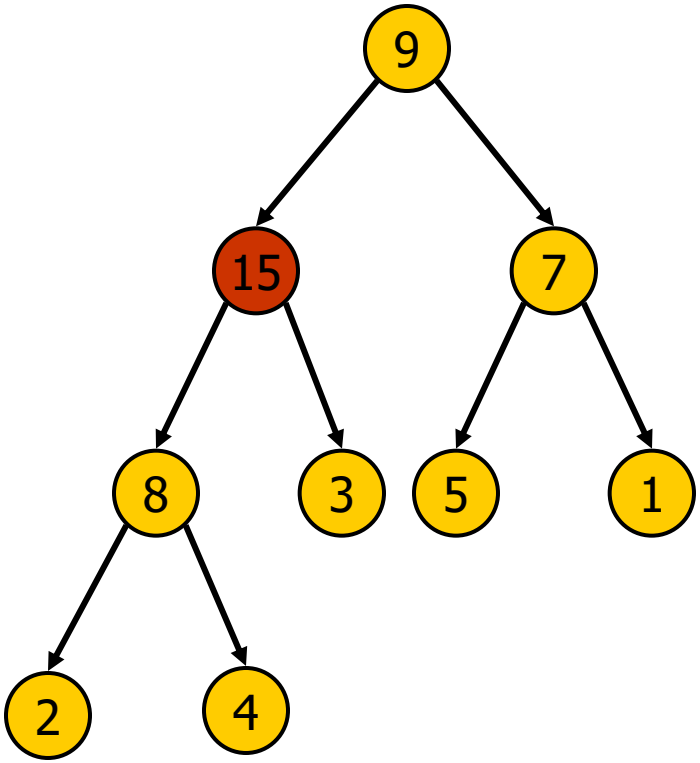
Note: For level-by-level traversal, when a tree is built with nodes and edges, we use a queue.

Insert an item

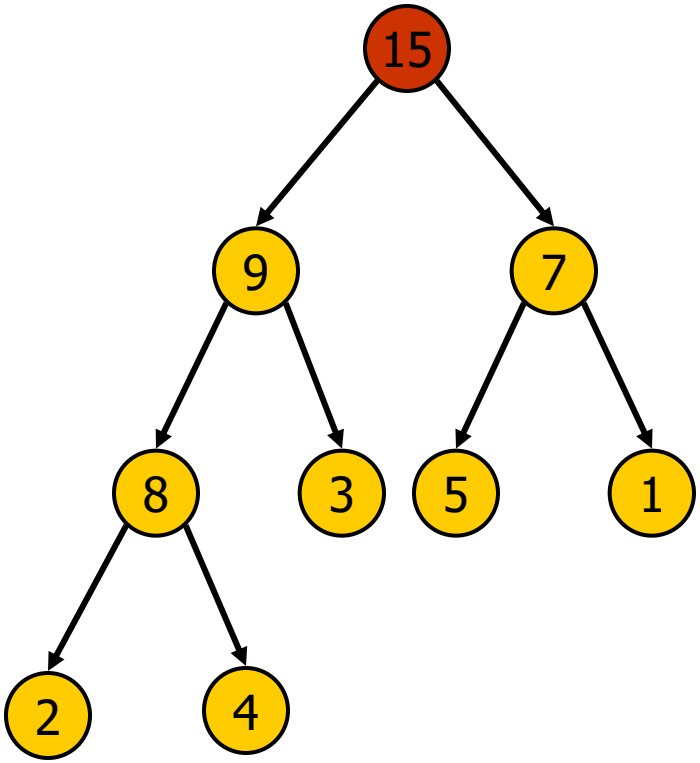


0	12
1	8
2	9
3	4
4	3
5	5
6	7
7	2
8	1
9	15

Re-establish heap property

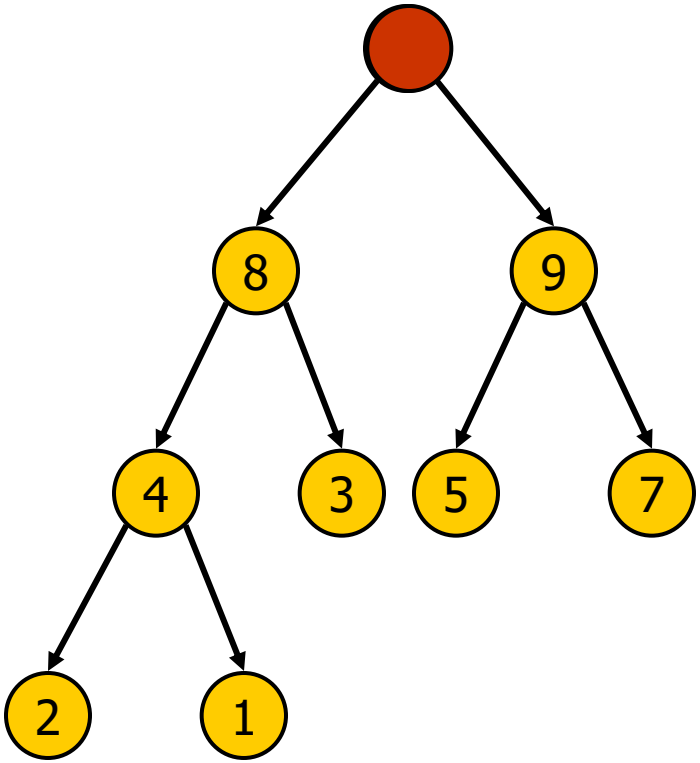


Re-establish heap property

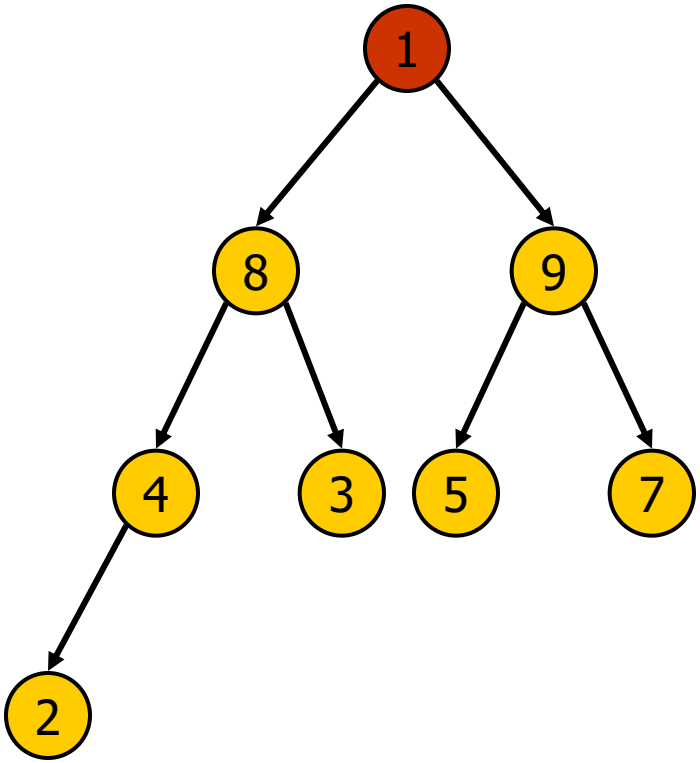


bubble up

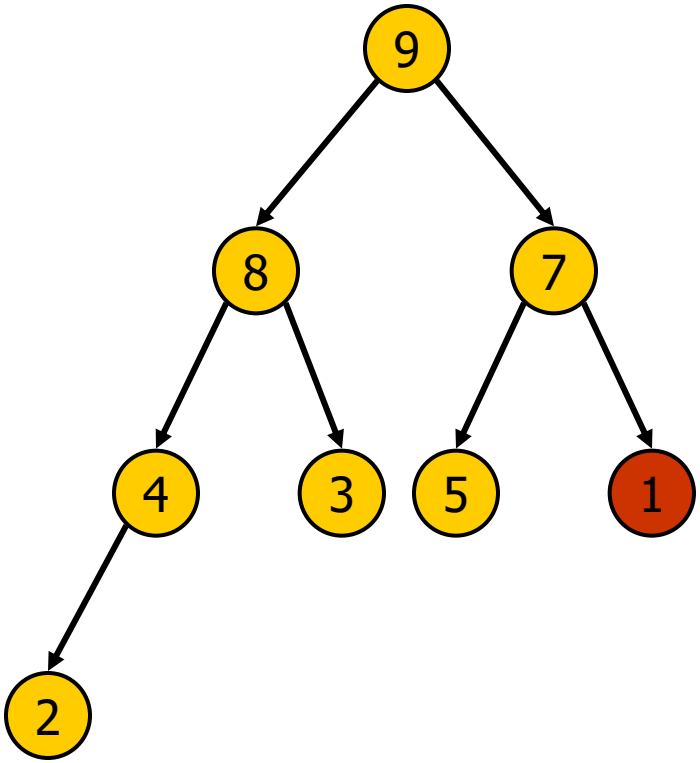
Remove the max item



Re-establish heap property



Re-establish heap property

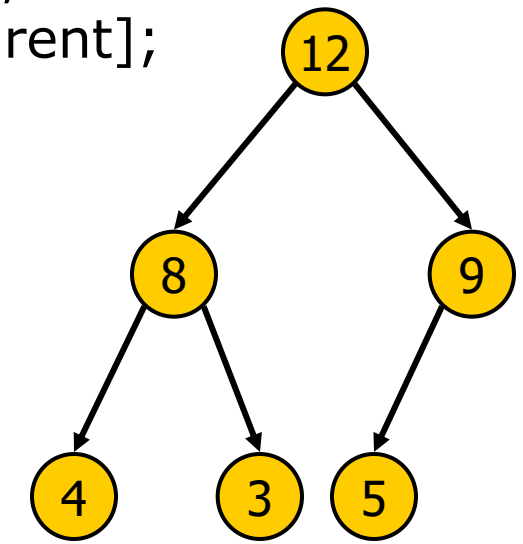


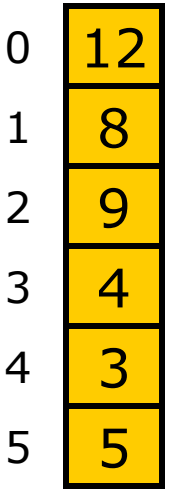
bubble down

Code

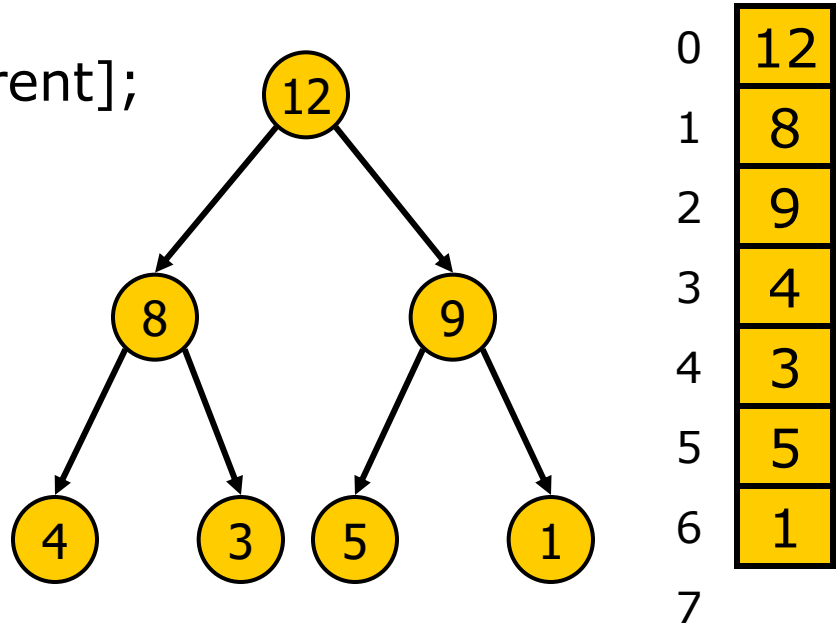
```
Public class Heap {
    private int MAX_HEAP = 100;
    private int [] items;
    private int size;

    public Heap() {
       item = new int[MAX_HEAP];
       size = 0;
    }

    public boolean heapIsEmpty() {
       return size == 0;
    }
```

```
public void heapInsert(int newItem) throws HeapException {
    if (size < MAX_HEAP) {
        items[size] = newItem;
        int place = size;
        int parent = (place - 1)/2;
        while ( (parent >= 0) && (items[place] > items[parent]) {
            int temp = items[place];
            items[place] = items[parent];
            items[parent] = temp;

            place = parent;
            parent = (place - 1)/2;
        }
        ++size;
    }
    else
        throw new HeapException("HeapException: Heap full");
}
```

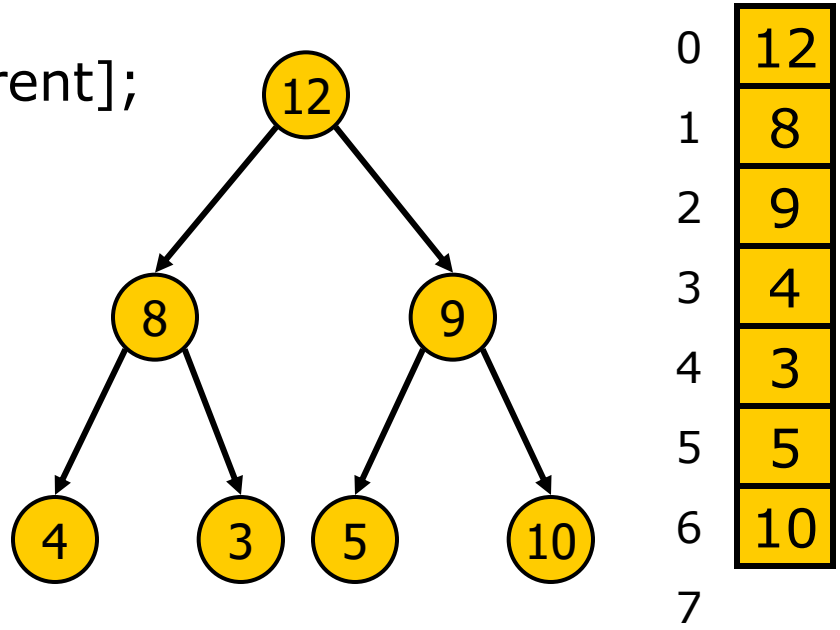




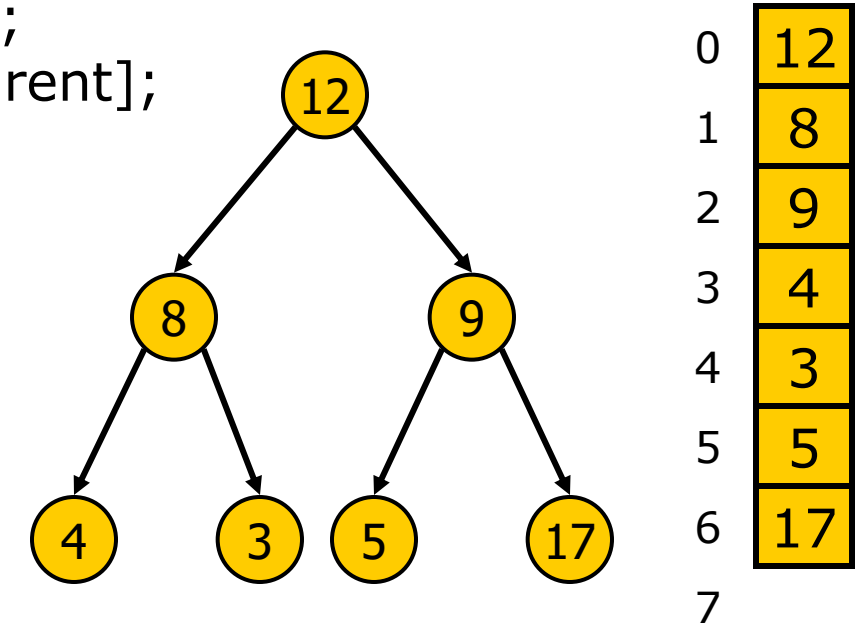
```
public void heapInsert(int newItem) throws HeapException {
    if (size < MAX_HEAP) {
        items[size] = newItem;
        int place = size;
        int parent = (place - 1)/2;
        while ( (parent >= 0) && (items[place] > items[parent]) {
            int temp = items[place];
            items[place] = items[parent];
            items[parent] = temp;

            place = parent;
            parent = (place - 1)/2;
        }
        ++size;
    }
    else
        throw new HeapException("HeapException: Heap full");
}
```



```
public void heapInsert(int newItem) throws HeapException {
    if (size < MAX_HEAP) {
        items[size] = newItem;
        int place = size;
        int parent = (place - 1)/2;
        while ( (parent >= 0) && (items[place] > items[parent]) {
            int temp = items[place];
            items[place] = items[parent];
            items[parent] = temp;

            place = parent;
            parent = (place - 1)/2;
        }
        ++size;
    }
    else
        throw new HeapException("HeapException: Heap full");
}
```

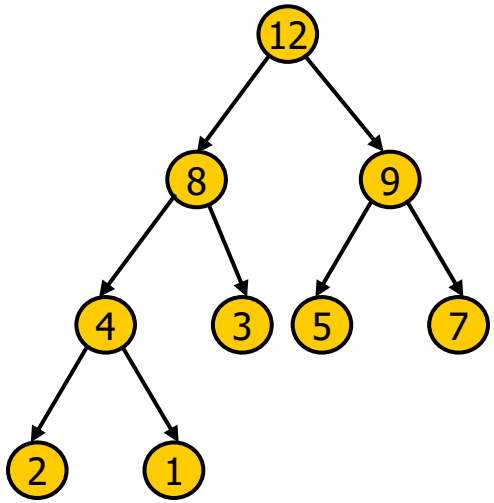


```
public void heapInsert(int newItem) throws HeapException {
    if (size < MAX_HEAP) {
        items[size] = newItem;
        int place = size;
        int parent = (place - 1)/2;
        while ( (parent >= 0) && (items[place] > items[parent]) {
            int temp = items[place];
            items[place] = items[parent];
            items[parent] = temp;

            place = parent;
            parent = (place - 1)/2;
        }
        ++size;
    }
    else
        throw new HeapException("HeapException: Heap full");
}
```



```
public int heapDelete() {
        int rootItem = 0;
        if (!heapIsEmpty()) {
            rootItem = items[0];
            items[0] = items[--size];
            heapRebuild(0);
        }
        return rootItem;
    }
```

```
protected void heapRebuild(int root) {
        int child = 2 * root + 1;        // left child

        if (child < size) {                    // there is a left child
            int rightChild = child + 1;  // right child

            if ( (rightChild < size) &&  // there is a right child
                 (items[rightChild] > items[child]) )
                child = rightChild;      // choose child with bigger value

            if ( items[root] < items[child] ) {     // trickle down
                int temp = items[root];
                items[root] = items[child];
                items[child] = temp;
                heapRebuild(child);
            }
        }
    }
```



Running time

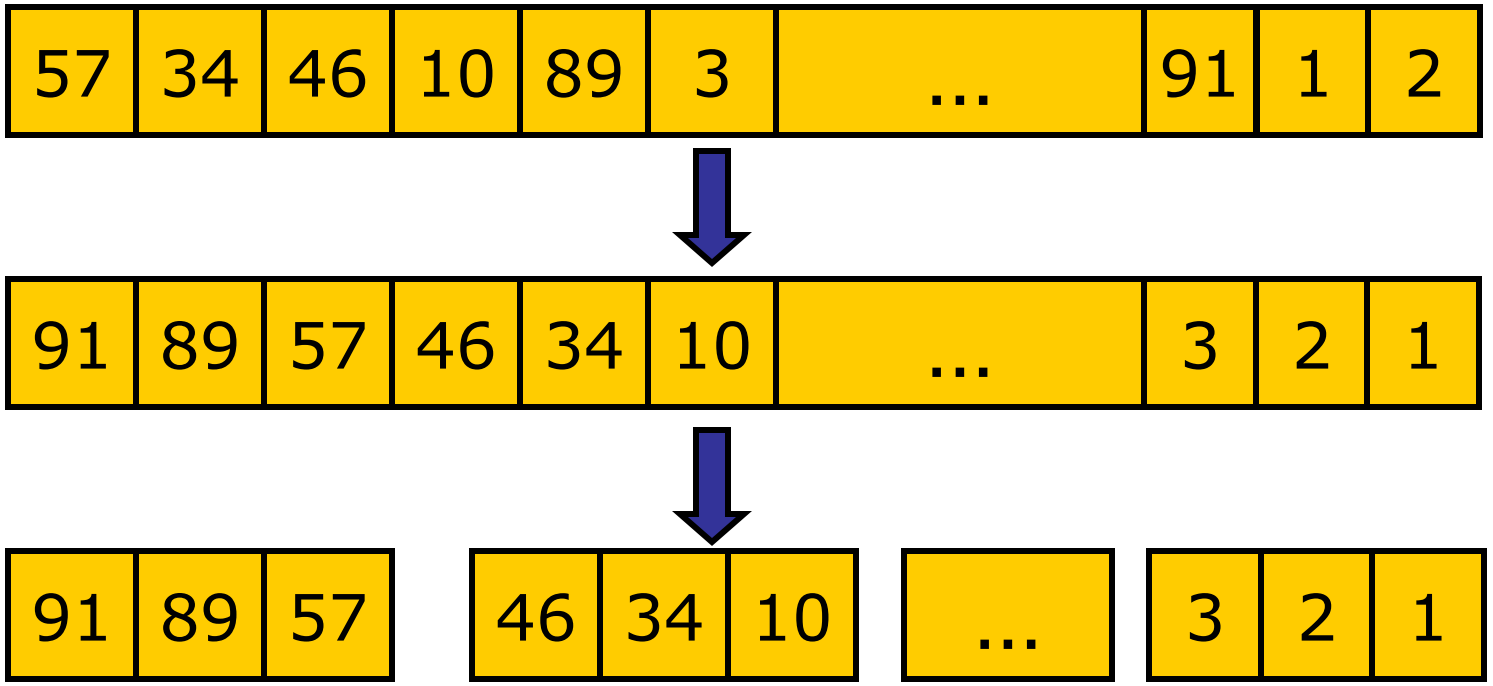
- How many calls to **heapRebuild**?
- Go down 1 level after each call to **heapify.**
- number of calls = height of tree
- Worst case running time $O(h)=O(\log n)$

Heap Construction

Display ranked web pages

- Display 10 pages at a time in order of decreasing page rank scores

Sort the scores



- Sort the web pages according to their scores
- Traverse the sorted list

Running times

- Sorting $O(n \log n)$
  n: total number of pages
- Traversing $O(k)$
  k: number of pages requested
- Total running time
  $O(n \log n) + O(k)$
  $<= O(n \log n) + O(n)$, since $k<n$
  $= O(n \log n)$

Can we do better? Maybe

Idea

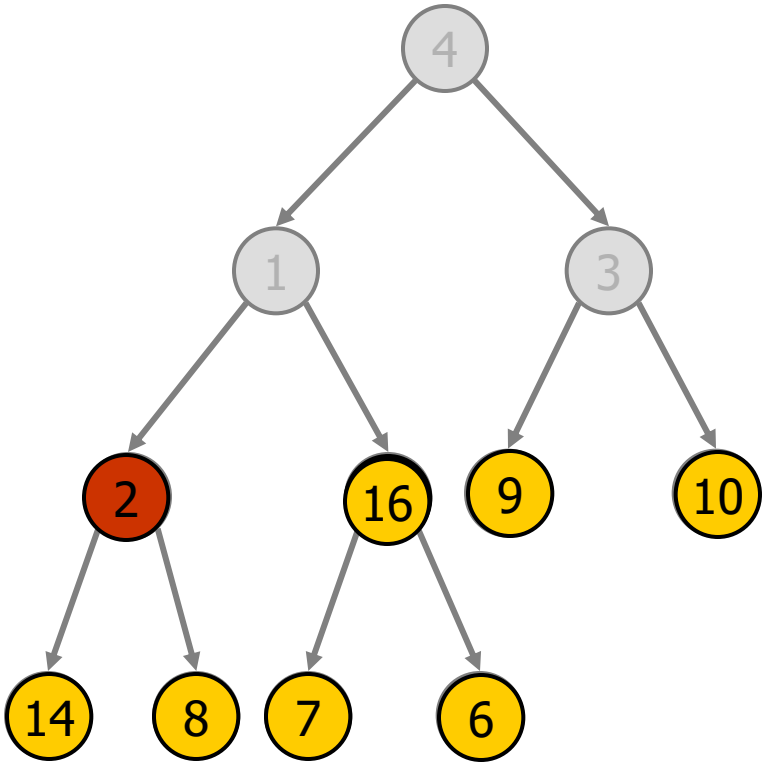
- Build a heap of scores
- Remove the top 10 pages at a time

Heap construction

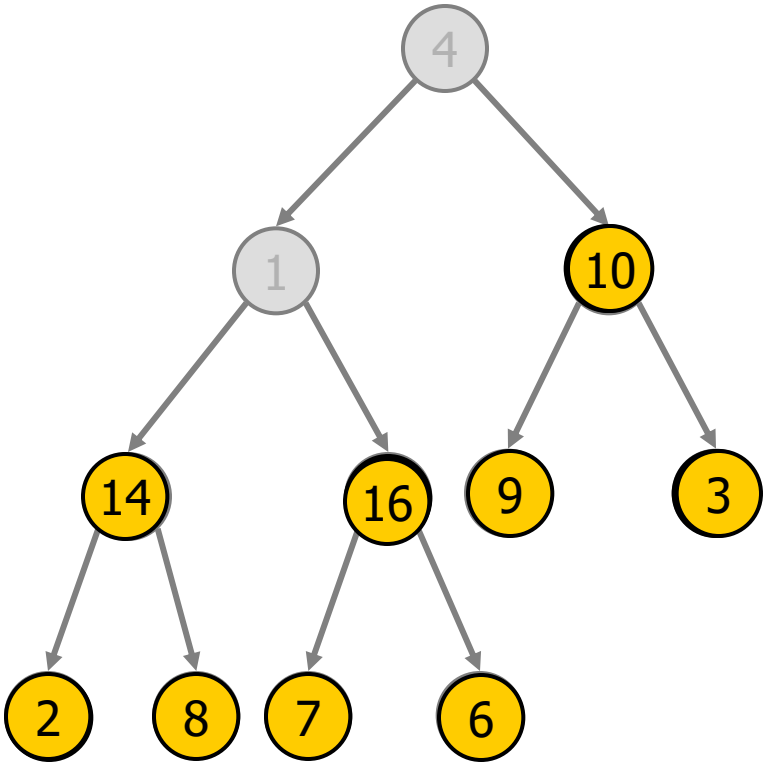
- **heap property**: for every node *v*, the search key in *v* is greater than those in the children of *v*
- Build the heap recursively from bottom up

Heap construction example

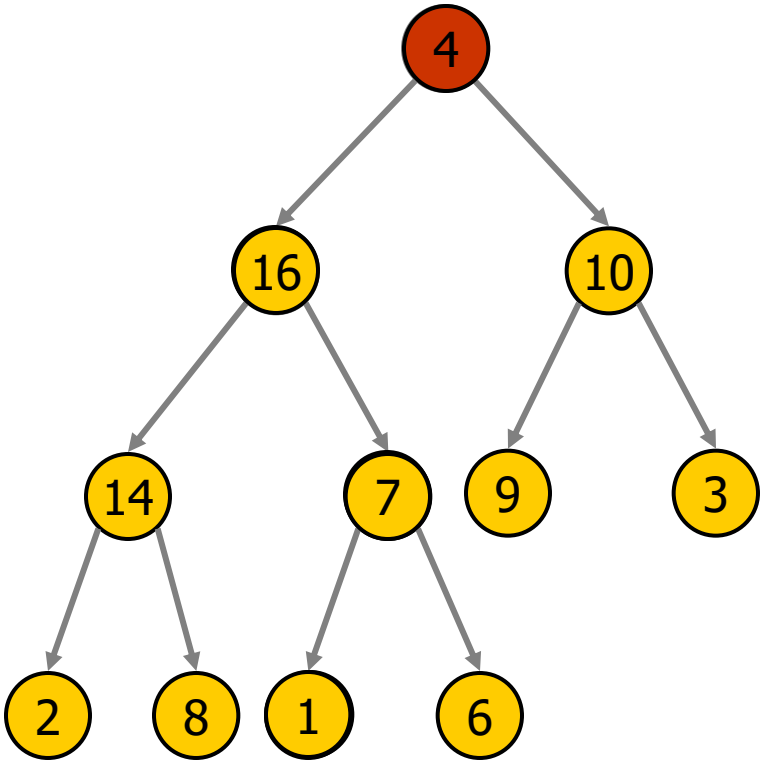
0	4
1	1
2	3
3	2
4	16
5	9
6	10
7	14
8	8
9	7
10	6



Heap construction example

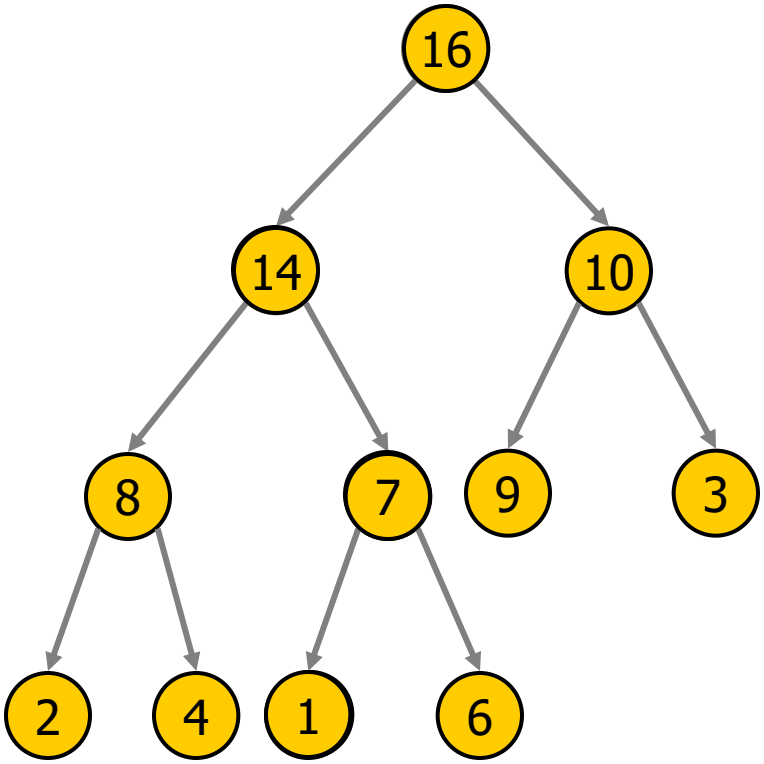


Heap construction example



Heap construction example

0	16
1	14
2	10
3	8
4	7
5	9
6	3
7	2
8	4
9	1
10	6



Code

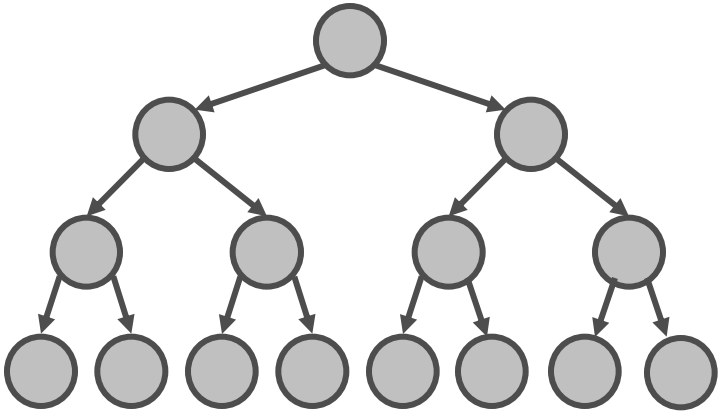
```
Protected void heapify() {
    for (int i = size/2; i >= 0; i--)
        heapRebuild(i);
}
```

Running time

- Rough count

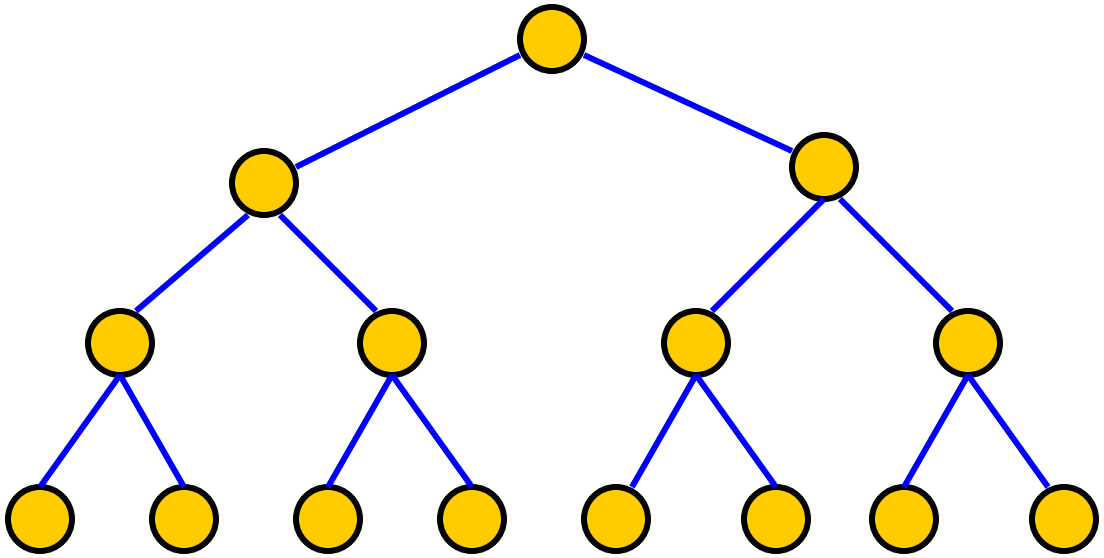
$$n \cdot O(\log n) = O(n \log n)$$

- More careful count



Running time: tight bound

- $O(n)$



Total number of nodes: $n = 2^{h+1} - 1$

Total number of bubbling down: $n - h - 1$

Web page ranking again

- Build a heap *O(n)*
- Retrieve top k pages *O(k log n)*
- Total running time
  O(n) + O(k log n)
  - If k=n, then O(n log n)
  - If k=20, O(n) + O(20 log n) = O(n)

Heapsort

- Uses a heap to sort an array of items.
  - Transform the array into a heap using the heap construction method discussed before.
  - Partition the array into two regions:
    - The heap region and the sorted region

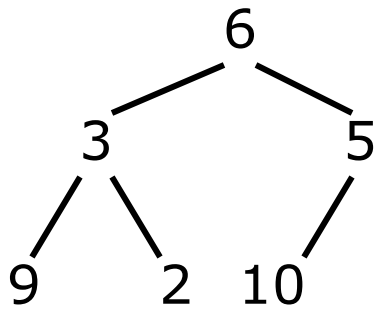
Heapsort

- Each step moves an item I from the heap region to the sorted region such that
  - After step k, the sorted region contains the k largest values in the array and they are in sorted order.
  - The items in the heap region form a heap.

Heapsort - Example

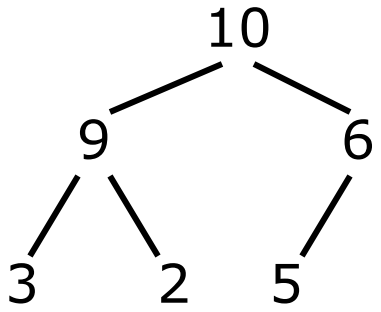
Original array

6	3	5	9	2	10



After heap construction

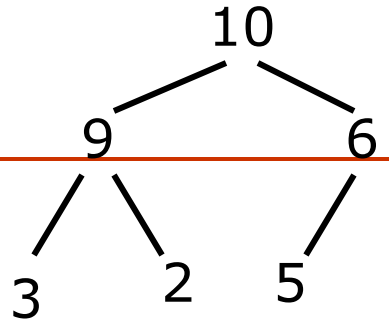
10	9	6	3	2	5



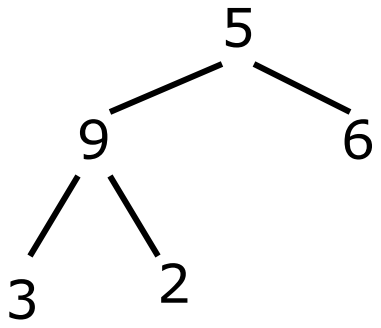
Heapsort - Example

Sorting Step

10	9	6	3	2	5

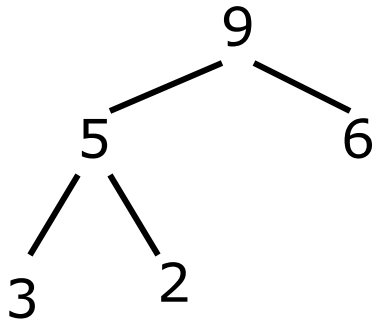


heap					sorted
5	9	6	3	2	10

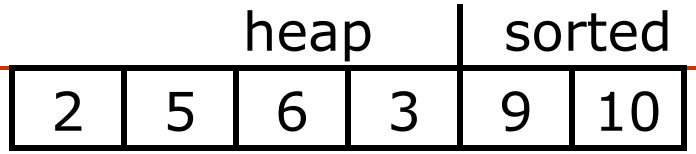


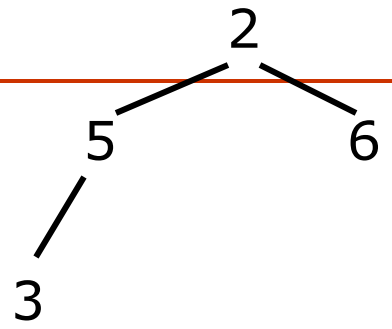
Heapify

heap					sorted
9	5	6	3	2	10

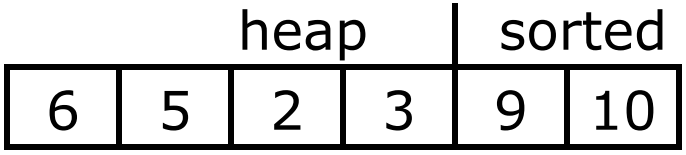


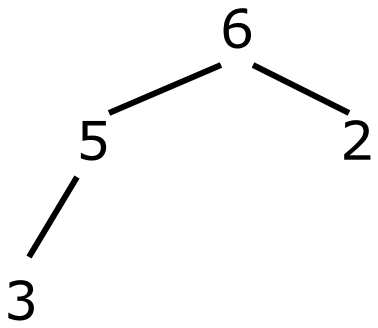
Heapsort - Example



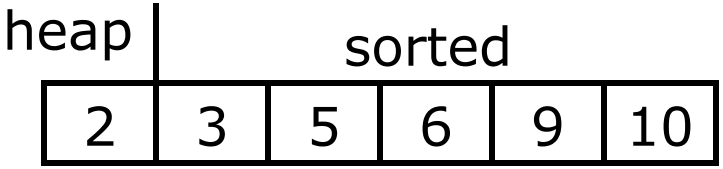


Heapify





Sorted



Is it in place?

Is it stable?

Complexity?