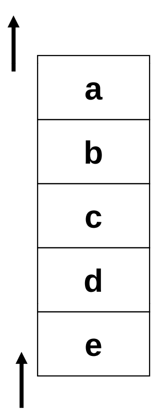
1.204 Lecture 8

Data structures: heaps

Priority Queues or Heaps



Top

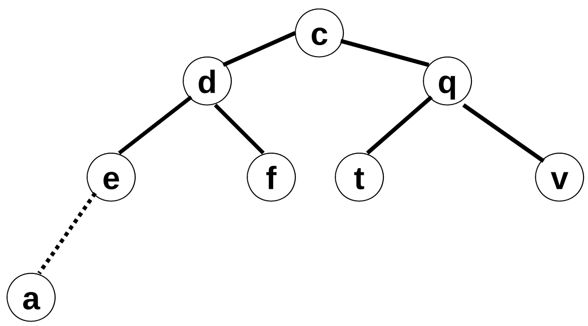
Bottom

- Highest priority element at top
- “Partial sort”
- All enter at bottom, leave at top

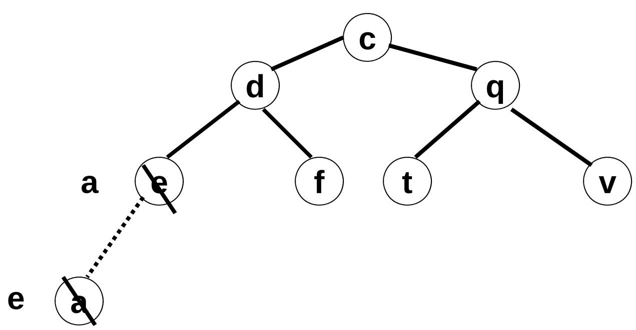
Applications:
1. Simulations: event list
2. Search, decision trees
3. Minimum spanning tree
4. Shortest path (label setting)
5. And many others…

Complexity:
1. Insertion, deletion: O(lg n)

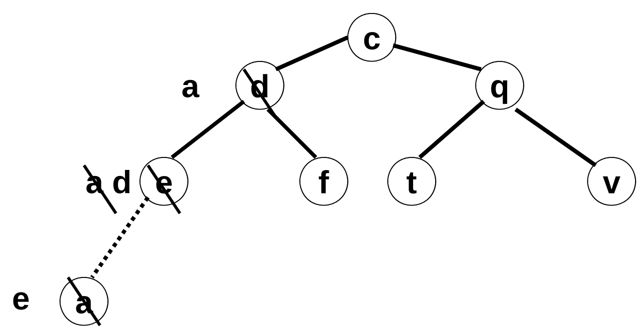
Min Heap Modeled as Binary Tree



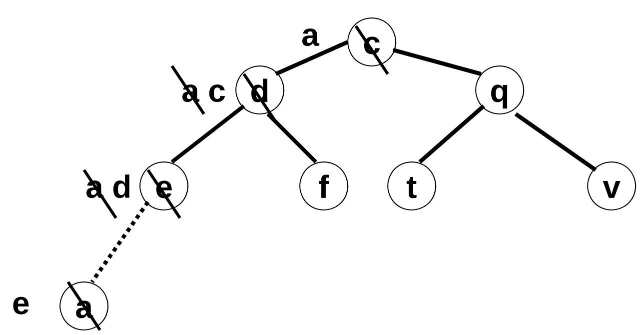
Min Heap Modeled as Binary Tree



Min Heap Modeled as Binary Tree



Min Heap Modeled as Binary Tree



Heap: constructors

```
public class Heap {    // Max heap: largest element at top
   private Comparable[] data;
   private int size;   // Actual number of elements in heap
   private int capacity;
   private static final int DEFAULT_CAPACITY= 30;

   public Heap(int capacity) {
        data = new Comparable[capacity];
        this.capacity= capacity;
   }

   public Heap() {
        this(DEFAULT_CAPACITY);
   }

   public Heap(Comparable[] c) {
        data= c;
        heapify(data);
        capacity= size= data.length;
   }
```

(Max) Heap insertion

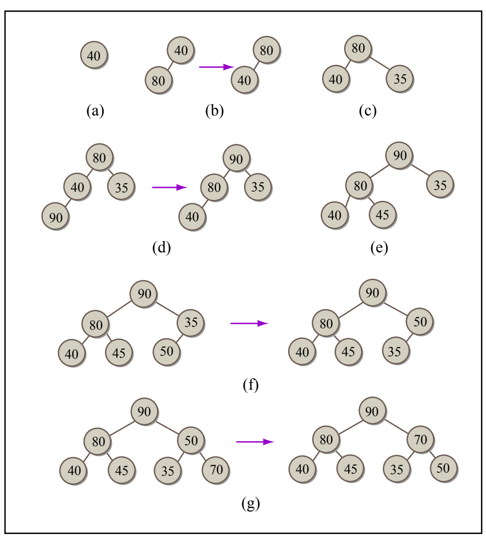


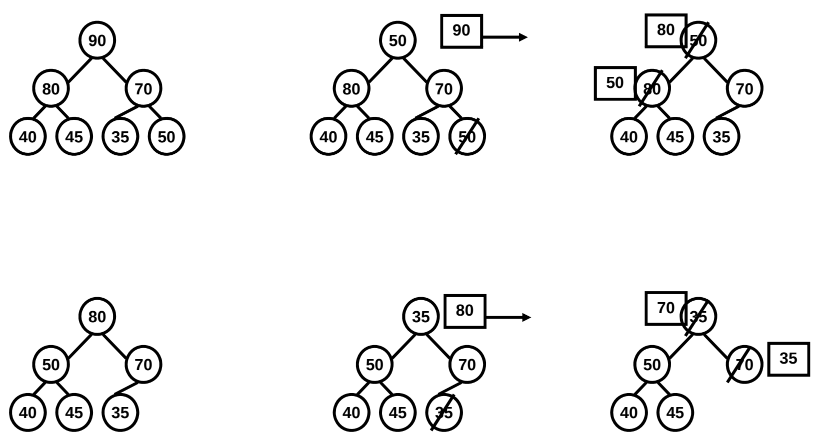
Figure by MIT OpenCourseWare.

Heap: insert()

```
public void insert(Comparable item) {
  if (size == 0) { // Empty heap, first element being added
    size= 1;
    data[0]= item;

  } else {
    if (size == data.length)
      grow();
    int i = size++; // Increase no of elements
    while (i > 0 && (data[(i-1)/2].compareTo(item) < 0)) {
      data[i] = data[(i-1)/2]; // Move parent item down
      i = (i-1)/2;             // Go up one level in heap
    }
    data[i] = item; // Drop item into correct place in heap
  }
}                    // See download for grow() code
```

(Max) Heap deletion



Heap: delete()

```
public Comparable delete() throws NoSuchElementException {
  if (size == 0)
    throw new NoSuchElementException();

  Comparable retValue = data[0];   // Top removed and returned
  // Put last element at top (element 0) and bubble it down
  Comparable item = data[0] = data[--size];
  int j = 1;     // Look at right and left children of top node

  while (j < size) {
    // Compare left and right child and let j be larger child
    if ((j+1 < size) && (data[j].compareTo(data[j + 1]) < 0))
      j++;
    if (item.compareTo(data[j]) > 0)
      break;                        // Position for item found
    data[(j-1) / 2] = data[j];    // Else put in parent node
    j = 2*j+1;             // Move down to next level of heap
  }

  data[(j-1) / 2] = item;  // Drop last element in place
  return retValue;
}
```

(Max) Heapify

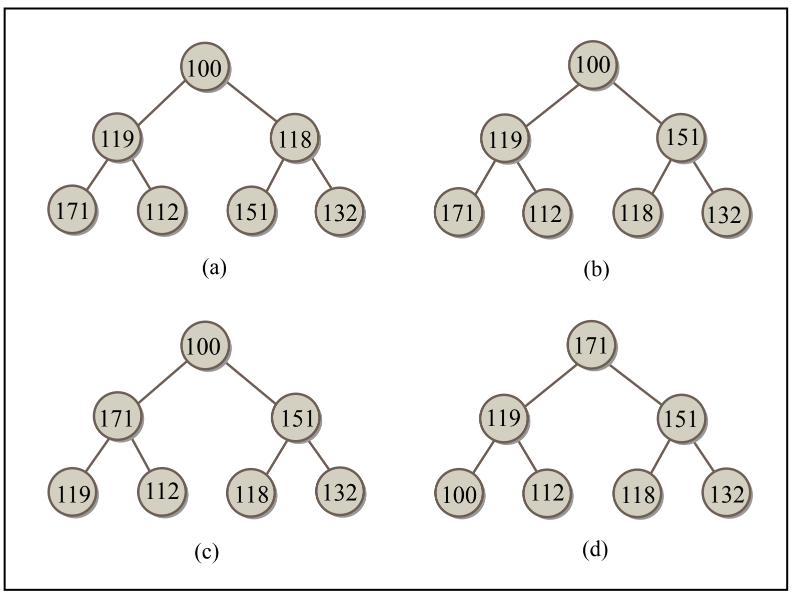
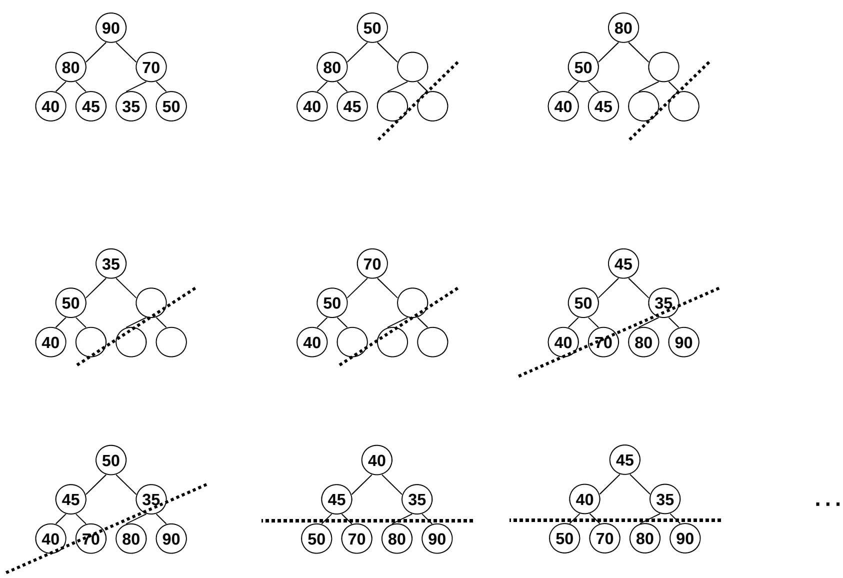


Figure by MIT OpenCourseWare.

Heap: heapify()

```
private static void heapify(Comparable[] c) {
  Comparable item;
  int size= c.length;
  for (int i= size/2 - 1; i >= 0; i--) {  // Start at mid-tree node
    int j= 2*i + 1;        // Left child
    item= c[i];
    while (j < size) {     // While loop same as delete()
    // Compare left and right child and let j be larger child
      if ((j+1 < size) && (c[j].compareTo(c[j + 1]) < 0))
        j++;
      if (item.compareTo(c[j]) > 0)
        break;             // Position for item found
      c[(j-1) / 2] = c[j]; // Else put child data in parent node
      j = 2*j+1;           // Move down to next level of heap
    }
    c[(j-1) / 2] = item;   // Drop last element into correct place
  }
}
```

(Max) Heapsort



Heap: heapsort()

```
public static Comparable[] heapsort(Comparable[] c) {
  heapify(c);
  Comparable item;
  int size= c.length;
  for (int i= size-1; i > 0; i--) {
    Comparable t= c[i]; // Swap top element with ith element
    c[i]= c[0];
    c[0]= t;
    int j= 1;              // Left child
    item= c[0];

    while (j < i) {
    // Compare left and right child and let j be larger child
      if ((j+1 < i) && (c[j].compareTo(c[j + 1]) < 0))
        j++;
      if (item.compareTo(c[j]) > 0)
        break;             // Position for item found
      c[(j-1) / 2] = c[j]; // Else put data in parent node
      j = 2*j+1;           // Move down to next level of heap
    }
    c[(j-1) / 2] = item;   // Drop element into correct place
  }
  return c;
}
```

Heap: example

```
public static void main(String[] args) {  // Max heap
     System.out.println("Heap");
     Heap h= new Heap(10);
     h.insert("b");
     h.insert("d");
     h.insert("f");
     h.insert("a");
     h.insert("c");
     h.insert("e");
     h.insert("g");
     h.insert("h");
     h.insert("i");
     String top = null;
     while (h.getSize() > 0) {
             top= (String) h.delete();
             System.out.println(" "+ top);
     }
```

Heap: example, p.2

```
        System.out.println("\nHeapify");

        String[] s= {"a", "b", "c", "d", "e", "f", "g", "h",
                "i", "j", "k", "l", "m"};
        Heap h2= new Heap(s);
        while (!h2.isEmpty()) {
               top= (String) h2.delete();
               System.out.println(" "+ top);
        }
        System.out.println("\nHeapsort");

        String[] s2= {"z", "b", "x", "d", "y", "f", "w", "h",
                "v", "j", "u", "l", "t"};
        String[] answer2= (String[]) Heap.heapsort(s2);
        for (String ss : answer2)
               System.out.println(" "+ss);
        System.out.println();
    }
} // Download also has MinHeap class; Heap is max heap
```

Heap performance

- **Heap insert**
  - **Maximum number of operations= number of levels in tree = O(lg n), where n is number of nodes**
- **Heap delete**
  - **Same as insert**
- **Heapsort**
  - **First execute heapify (analyzed below)**
  - **Number of operations= number of nodes * adjustments/node, which are O(lg n) deletions**
  - **Thus heapsort is O(n lg n)**
    - **Similar to quicksort, but quicksort tends to be twice as fast**

Heapify performance

n= number of nodes in heap

K= levels in heap

(0 is bottom, K is top)

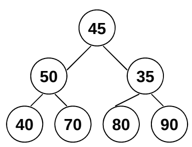
K= floor(lg n) +1

(n+1)/(2^{k+1}) nodes in each level

Heapify moves a node at level k a maximum of k steps

The total number of steps=

(number of nodes at each level) * (maximum moves for that level)



k=2

k=1

k= 0

$$\sum_{k=0}^{\lg n} \frac{n}{2^{(k+1)}} * k = n\sum_{k=0}^{\lg n} \frac{k}{2^{(k+1)}} \cong n\sum_{k=0}^{\lg n} k\left(\frac{1}{2}\right)^k$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad for\ |x|<1$$

CLRS, page 1061, (A.8)

Thus

$$\sum_{k=0}^{\lg n} k\left(\frac{1}{2}\right)^k \le \frac{1/2}{(1-1/2)^2} = 2$$

$$n\sum_{k=0}^{\lg n} k\left(\frac{1}{2}\right)^k = O(2n) = O(n)$$