

## Revision

These are the data structures we covered:

- stacks
- queues
- linked lists
- binary search trees
- priority queues
- binary heaps
- hash tables

## Stacks

- **last-in-first-out** data structures (LIFO)
- the interface

```
public interface Stack {  
  
    // Indicates the status of the stack.  
    public boolean isEmpty();  
  
    // Pushes an item onto the stack.  
    public void push(Object item);  
  
    // Pops an item off the stack.  
    public Object pop();  
  
}
```

- can be implemented using arrays or lists

## Queues

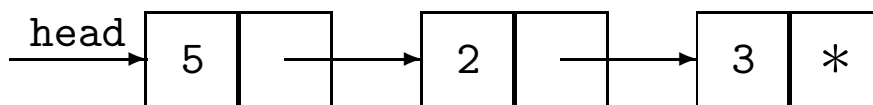
- **first-in-first-out** data structures (FIFO)
- the interface

```
public interface Queue {  
  
    // Indicates the status of the queue.  
    public boolean isEmpty();  
  
    // Attaches an item to the queue.  
    public void enqueue(Object item);  
  
    // Detaches an item from the queue.  
    public Object dequeue();  
  
}
```

- can be implemented using arrays or lists

## Linked Lists

- one-dimensional **dynamic** data structure



- singly linked lists can only be directly accessed at front
- general access only sequentially from front
- based on the class

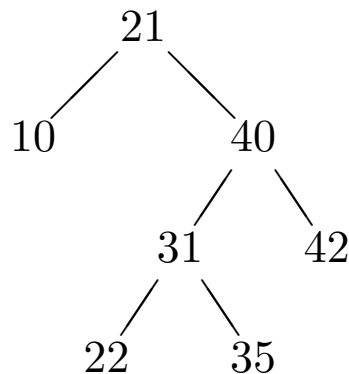
```
class ListNode {  
    Object data;  
    ListNode next;  
    ....  
}
```

```
public class LinkedList {
    private ListNode head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }
    public boolean isEmpty();
    public int size();
    public void addFirst(Object item);
    public Object firstItem();
    public void removeFirst();
    public int remove(Object item);
    public boolean contains(Object item);
    public void reverse();
    public Iterator iterator();
    public boolean equals(Object other);
    public String toString();
}
```

## Binary search trees

- linked data structures with the following property
  - (B) a **binary tree** is either empty, or else consists of some data item together with a left and right descendant binary tree
- one of the most useful types of binary tree is a binary search tree, for example



- (S) a **binary search tree** is a binary tree in which the value stored at each node of the tree is greater than the values stored in its left subtree and less than the values stored in its right subtree
- binary search trees can be constructed for any type of data; all that matters is that the data items can be compared with regard to order

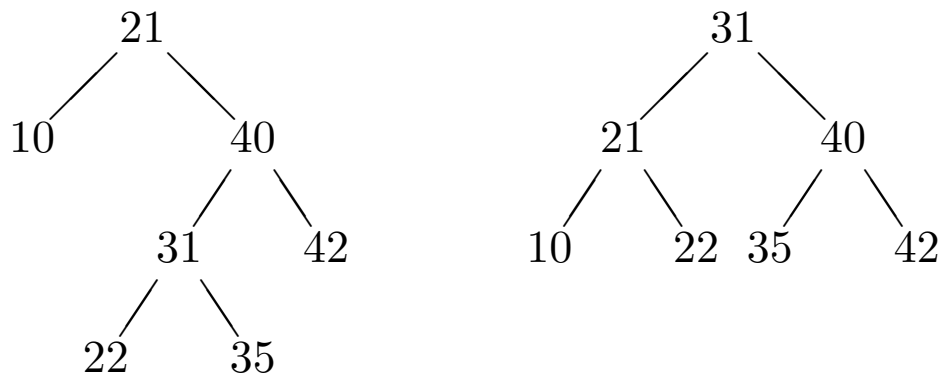
## The abstract SearchTree class

```
public abstract class SearchTree {  
  
    public static SearchTree empty() {  
        return new EmptyTree();  
    }  
  
    public abstract boolean isEmpty();  
    public abstract int nrNodes();  
    public abstract boolean contains(Comparable item);  
    public abstract SearchTree add(Comparable item);  
    public abstract SearchTree remove(Comparable item);  
    public abstract Enumeration elementsInOrder();  
    public abstract Enumeration elementsLevelOrder();  
    public abstract String toString();  
}
```

- implement EmptyTree and NodeTree classes as extensions of SearchTree class:

```
        SearchTree  
        /   \  
    EmptyTree NodeTree
```

- the two methods `elementsInOrder` and `elementsLevelOrder` return an `Enumeration`
- the *in-order* enumeration enumerates the data elements of a binary search tree in increasing order
- both of the trees



have in-order enumerations as

10, 21, 22, 31, 35, 40, 42

- the *level-order* enumeration enumerates the elements of each level in turn: the first tree is enumerated in level-order as

21, 10, 40, 31, 42, 22, 35

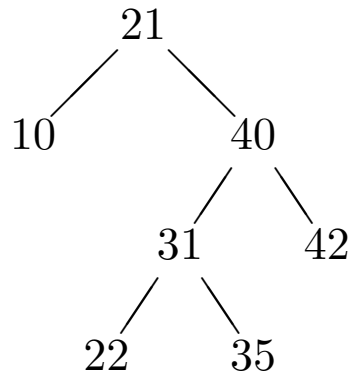
while the second tree is enumerated as

31, 21, 40, 10, 22, 35, 42

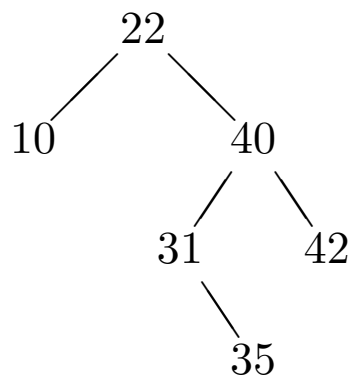
add

remove

- suppose that we wish to remove the item 21 from the binary search tree



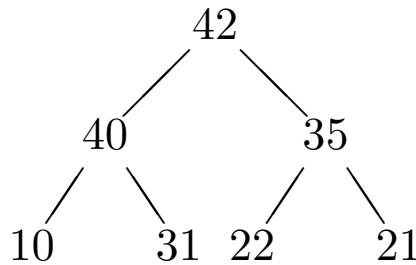
- some item other than 21 will have to be at the root of the new tree, and it will have to be larger than all elements in the current left sub-tree of 21 and smaller than all elements in the current right sub-tree of 21
- the simple solution we adopt is to replace 21 by the smallest item in the right sub-tree, and then to delete that item from the right sub-tree, leaving



- this is sure to preserve the ordering property

## Binary heaps

- an alternative to the ordering of binary trees is for the items to be ordered from bottom to top



- (H) the value stored at each node of the tree is greater than or equal to the values stored in its left subtree, and greater than or equal to the values stored in its right subtree
- trees satisfying (H) are particularly useful if they are well balanced, so we want binary trees in which every level is fully occupied except, possibly, for the bottom level, which is filled from left to right
  - binary trees of this type are called **complete**
  - a complete binary tree which also satisfies the ordering property (H) is called a **binary heap**

## Priority queues

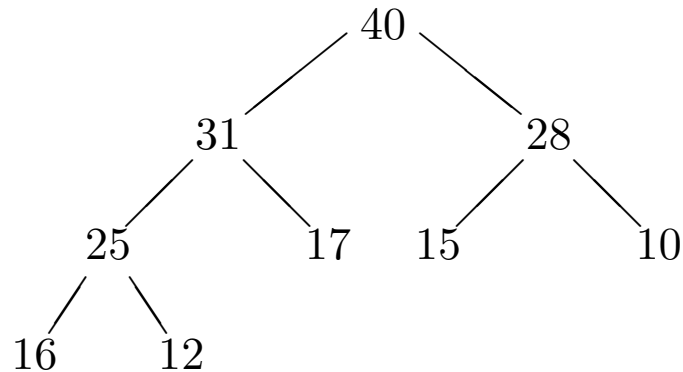
- frequently useful to modify the **queue** concept by attaching a **priority** to items in the queue—items of highest rank in the queue should be served first
- simple interface for a priority queue

```
public interface PriorityQueue {  
    public boolean isEmpty();  
    public int size();  
    public void add(Comparable item);  
    public Comparable remove();  
}
```

- items in a priority queue need to be `Comparable` with respect to order
- implement a `PriorityQueue` by means of a `BinaryHeap`

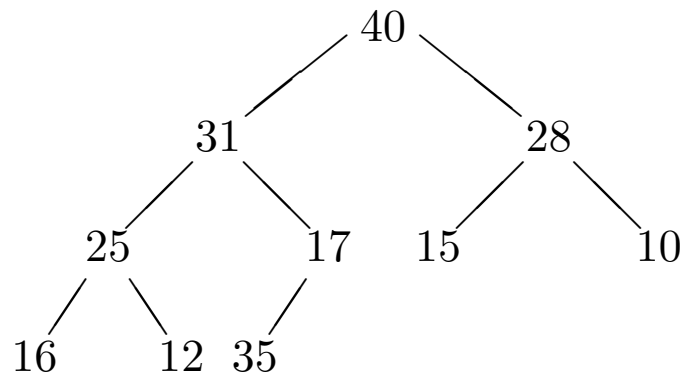
## Insertion algorithm

- suppose we have the binary heap



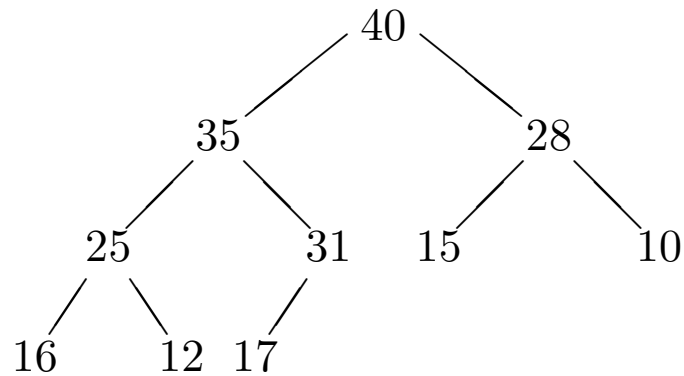
and we have to insert the number 35

- the structure of the new heap is fixed by the need for it to be a complete binary tree; it must have the form



- the 35 has been placed in the next vacant place, although this now violates the heap ordering property
- now bubble the 35 up the tree until the order of this path is correct, by the sequence of transformations  $(40, 31, 17, \mathbf{35}) \mapsto (40, 31, \mathbf{35}, 17) \mapsto (40, \mathbf{35}, 31, 17)$

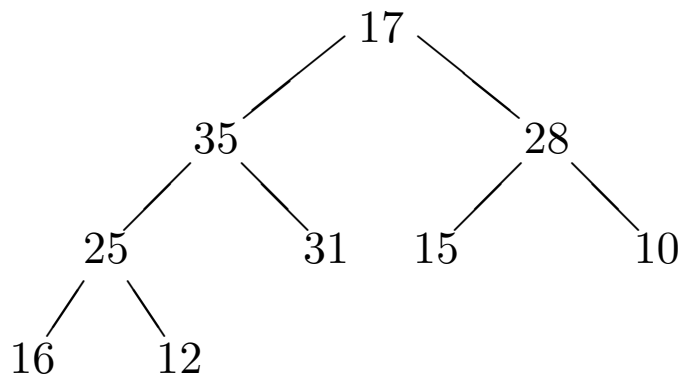
- after making these two exchanges, we obtain the heap



- note that 35 is also no less than its left descendant 25

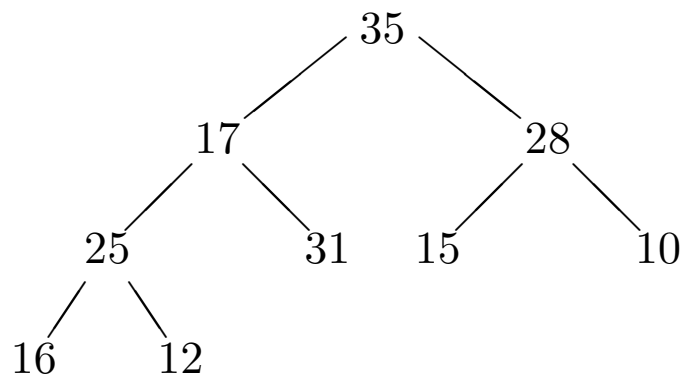
### Removal algorithm

- suppose we want to remove an item from this heap—this will be the one at the top, namely 40
- place the last time 17 at the top of the heap

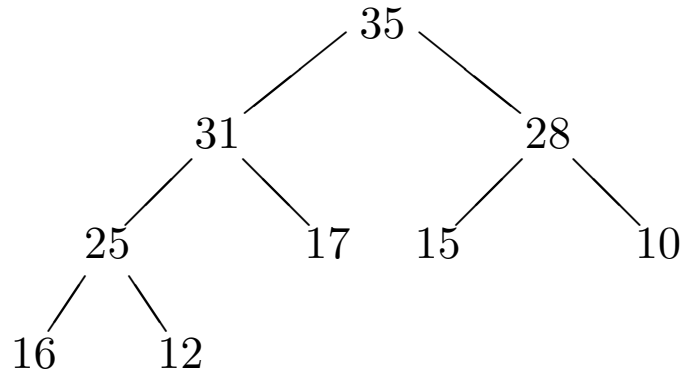


- this is now a complete binary tree, but it is not a heap because the 17 is smaller than both its children

- demote the 17 from the top by successively exchanging it with the **larger** of its two children (if they are the same, it doesn't matter which is chosen)
- on the first exchange we get



- 17 must now be compared with its children, 25 and 31
- using the larger of the two we obtain



- the algorithm is simple: put the last item at the top of the heap and then keep exchanging it with the larger of its children until heap order is restored

## Hash Tables

- implements a lookup-table or map

```
public interface LookupTable {
    public boolean isEmpty();
    public int size();
    public boolean contains(Object key);
    public void add(Object key, Object value);
    public void remove(Object key);
    public Object lookup(Object key);
    public interface Entry {
        public Object key();
        public Object value();
    }
    public Iterator iterator();
}
```

### Hashing

- aim of the **hash table** is access in constant time
- need a function which associates an integer with every data item of interest; this is the **hash code** of the object, and the process of forming it is called **hashing**
- the hash code determines the proper place of the data item in the underlying array, which is now to be called the **hash table**

## Collisions

- the problem where two objects initially map to the same array index, is called the problem of **collision resolution**
- there are several strategies for resolving collisions; we deal here with **open addressing**
- we want to insert a data object  $d$  into the table; let the hashing function be  $h$ , so the hash code for  $d$  is  $h(d)$
- since index must lie between 0 and `table.length-1` use
$$h(d) \% \text{table.length}$$
as the index at which to insert the item  $d$
- but suppose that, for two items  $m$  and  $n$  we have
$$h(m) \% \text{table.length} == h(n) \% \text{table.length}$$
- this could be because
  - $m$  and  $n$  had the same hash codes  $h(m)$  and  $h(n)$
  - they had the same remainder modulo `table.length`

## Linear probing

- the simplest approach to collision resolution is called **linear probing**
- sequence of locations probed as far as necessary is  
 $h, h+1, h+2, h+3, \dots$
- since this sequence may run off the end of the array, we wrap around, so sequence of probes is really  
 $h+i \% \text{table.length}$   
for  $i = 0, 1, 2, 3, \dots$

## Success

- using linear probing, we are bound to find a place to insert item  $n$  eventually if  $\text{size} < \text{table.length}$
- we are forced to grow the table if it is completely full, but it will be better to grow it when it is only half-full
- the proportion of the hash table that is currently occupied is called the **loadFactor**, say 50%

## Clustering

- linear probing leads to clusters of sequentially occupied locations: this is called **primary clustering**

## Quadratic probing

- an alternative to linear probing is **quadratic probing**

- explore

$h, h+1, h+4, h+9, h+16, \dots$

remembering that these increments are to be taken modulo `table.length`

## Success

- it can be shown, however, that successful insertion is guaranteed provided

- `table.length` is a prime number

- `loadFactor` is less than 0.5

neither requirement is difficult to satisfy

## Clustering

- quadratic probing is free from primary clustering, but still liable to **secondary clustering**
- this refers to the fact that if two keys collide, the same probe sequence will be followed for both

## Double hashing

- **double hashing** is another alternative to linear probing
- we now explore a sequence of probes  
 $h, h+k, h+2k, h+3k, h+4k, \dots$

where  $k$  lies between 1 and `table.length-1`

## Success

- assuming there is room for the item, successful insertion is guaranteed if `table.length` is a **prime number**

## Clustering

- double hashing is not subject to either primary or secondary clustering for well designed hash functions

## Performance

- despite theoretical differences in terms of clustering behaviour, there is not a huge difference in performance if `loadFactor` is reasonably small, e.g. less than 0.5

- for open addressing we can use
  - linear probing
  - quadratic probing
  - double hashing

