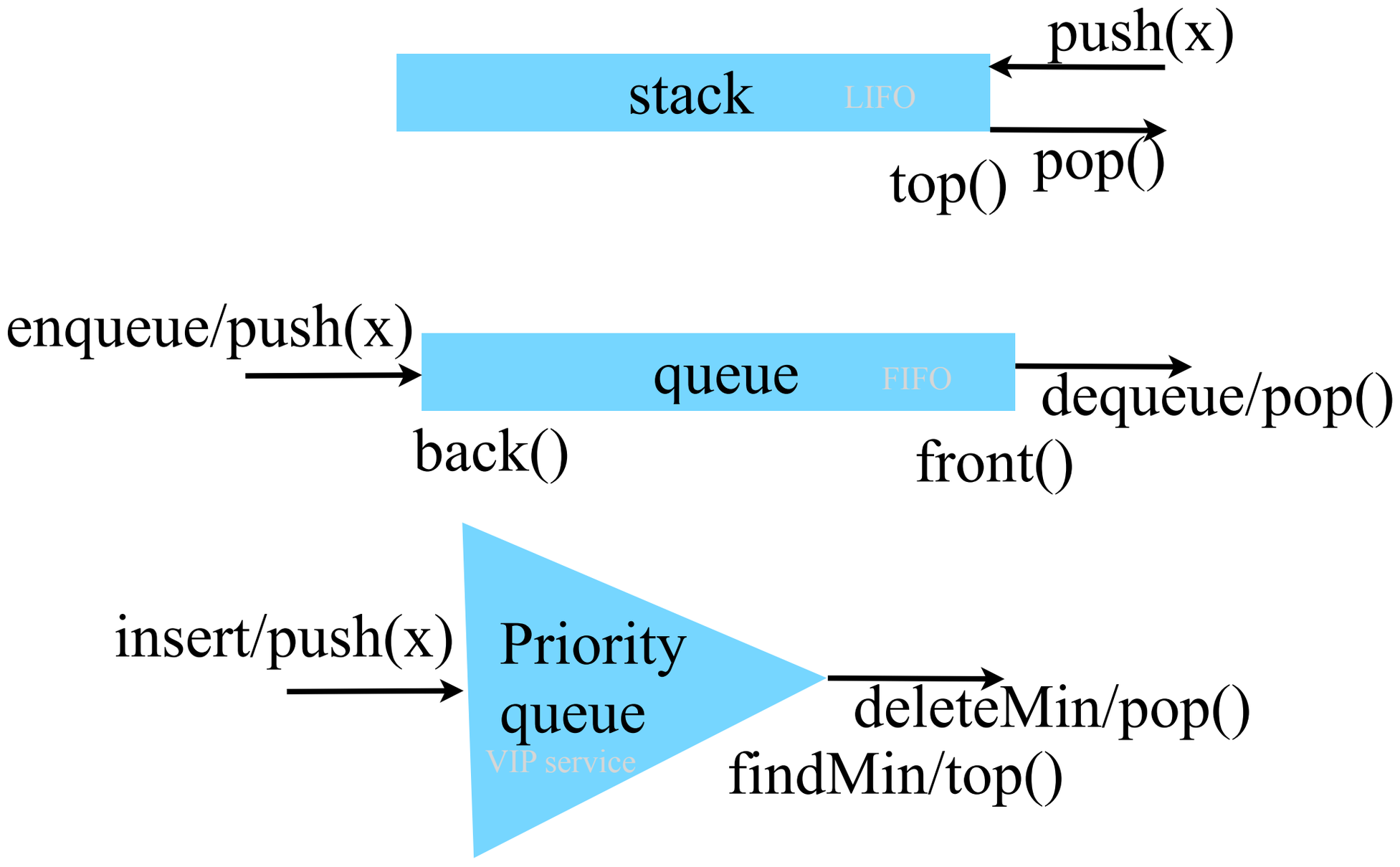
Lecture 20

Priority Queue implemented as a Heap

CS2134

Container Adapters



```
// priority_queue::push/pop
#include <iostream>
#include <queue>

using namespace std;

int main ()
{
  priority_queue<int> mypq;

  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);

  cout << "Popping out elements...";
  while (!mypq.empty())
  {
     cout << " " << mypq.top();
     mypq.pop();
  }
  cout << endl;

  return 0;
}
```

Priority Queue Example From http://www.cplusplus.com/reference/stl/priority_queue/push/

Using a Functor with a Priority Queue

```
priority_queue< int, vector<int>, greater<int> > mypq2;

mypq2.push(30);
mypq2.push(100);
mypq2.push(25);
mypq2.push(40);

cout << "Popping out elements...";
while (!mypq2.empty())
{
 cout << " " << mypq2.top();
 mypq2.pop();
}
 cout << endl;
```

This example is a slightly modified from http://www.cplusplus.com/reference/stl/priority_queue/push/

Helping Study for the Final with a Priority Queue

```
class to_do_list
{
public:
   to_do_list( int priority = 10, string job = ""): priority(priority), job(job){};

   bool operator<( to_do_list rhs ) { return get_priority() <= rhs.get_priority( ); }
   int get_priority( ) const {return priority; }
   string get_job_description( ) const { return job; }

private:
   int priority;
   string job;
};
```

```
 priority_queue< to_do_list > exam_prep;

exam_prep.push( to_do_list( 10, "get pizza" ) );
exam_prep.push( to_do_list( 8, "call mom" ) );
exam_prep.push( to_do_list( 4, "finish homework" ) );
exam_prep.push( to_do_list( 8, "buy book" ) );
exam_prep.push( to_do_list( 1, "read book" ) );

cout << "To do list:";
while (!exam_prep.empty())
{
 to_do_list item = exam_prep.top();
 cout << " " << item.get_job_description() << endl;
 exam_prep.pop();
}
cout << endl;
```

Using a functor with the Priority Queue

```
class compare_to_do_list_items
{
public:
   bool operator()(to_do_list lhs, to_do_list rhs){ return ( lhs.get_priority( ) <= rhs.get_priority
); }
};
```

container type!

```
priority_queue<to_do_list, vector<to_do_list>, compare_to_do_list_items > exam_prep;
exam_prep.push( to_do_list( 10, "get pizza" ) );
exam_prep.push( to_do_list( 8, "call mom" ) );
exam_prep.push( to_do_list( 4, "finish homework" ) );
exam_prep.push( to_do_list( 8, "buy book" ) );
exam_prep.push( to_do_list( 1, "read book" ) );
cout << "What to do! ..." << endl;
while (!exam_prep.empty())
{
 to_do_list job = exam_prep.top();
 cout << " " << job.get_priority( ) << " - " << job.get_job_description() << endl;
 exam_prep.pop();
}
cout << endl;
```

How should we implement the priority queue?

- sorted vector?
- unsorted vector?
- sorted list?
- queue?
- stack?
- set or map?

Min Priority Queues

Sometimes want non-FIFO queues in which elements with higher priority are retrieved before those with lower priority

Elements have priority numbers

low number means high priority

Operations:

findMin: returns the smallest element

deleteMin: removes the smallest element

insert: adds an element

These can be supported efficiently with a data structure called a heap

Can also be used for a nice sorting algorithm

We could also create a Max Priority Queue

“**Heap** may refer to:

Computer science

- Heap (data structure), a data structure commonly used to implement a priority queue
- Heap (or *free store*), an area of memory used for dynamic memory allocation

Mathematics

- Heap (mathematics), a generalization of a group”

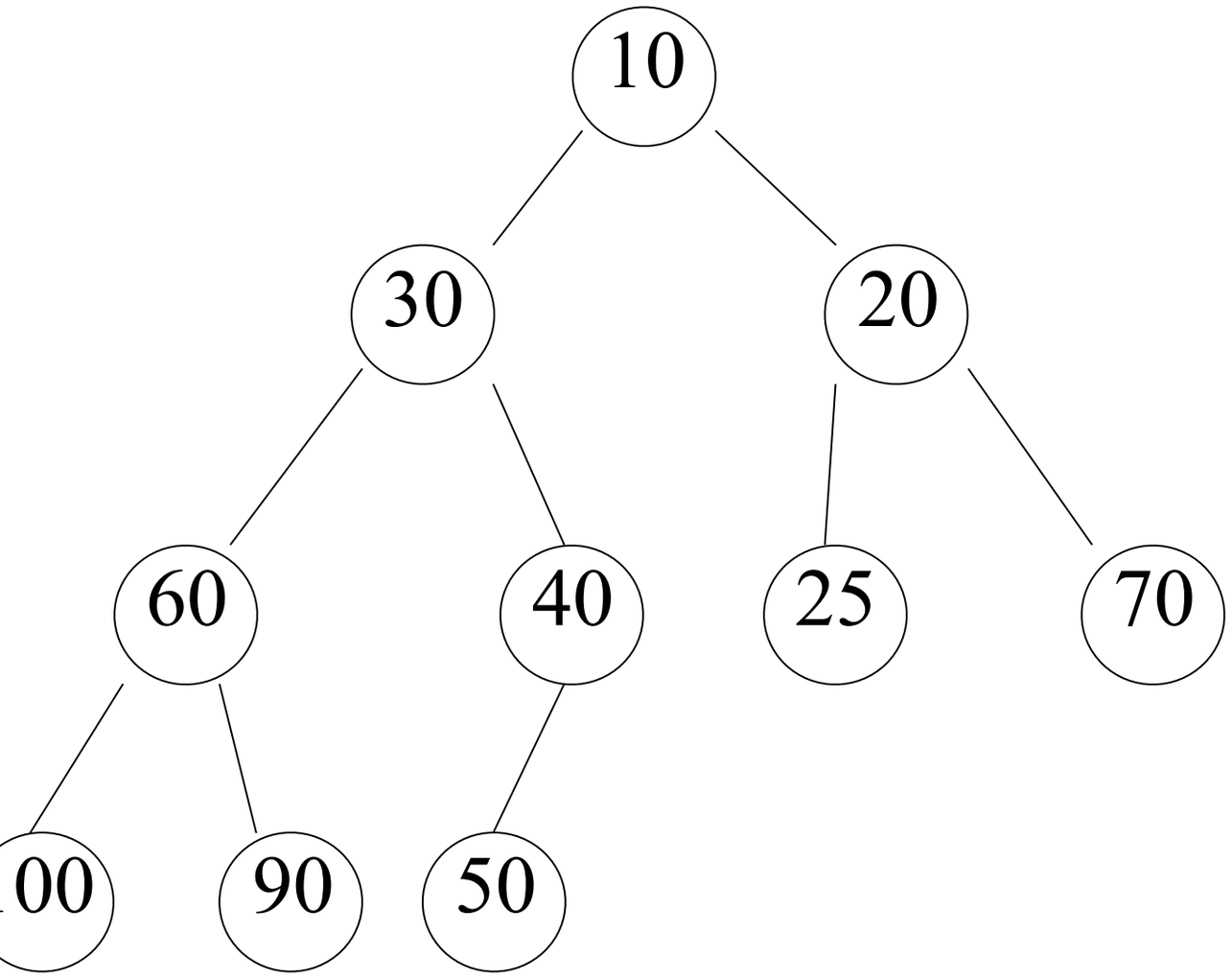
From http://en.wikipedia.org/wiki/Heap

Binary Heaps

- Complete binary tree: binary tree in which all levels are "full", except for possibly bottom level which is filled in from the left
- h = O(log n) where h is height and n is size
- A heap is a complete binary tree in which every node satisfies the heap property:
  - parent(x) <= x
- Observe
  - smallest node is at the root
  - heap is NOT Binary Search Tree

Complete Tree

Depth	max # nodes
0	2^0
1	2^1
2	2^2
3	2^3



Height?

A tree of height h has how many nodes??

A complete tree of height h has between 2^h and 2^{h+1} -1 nodes

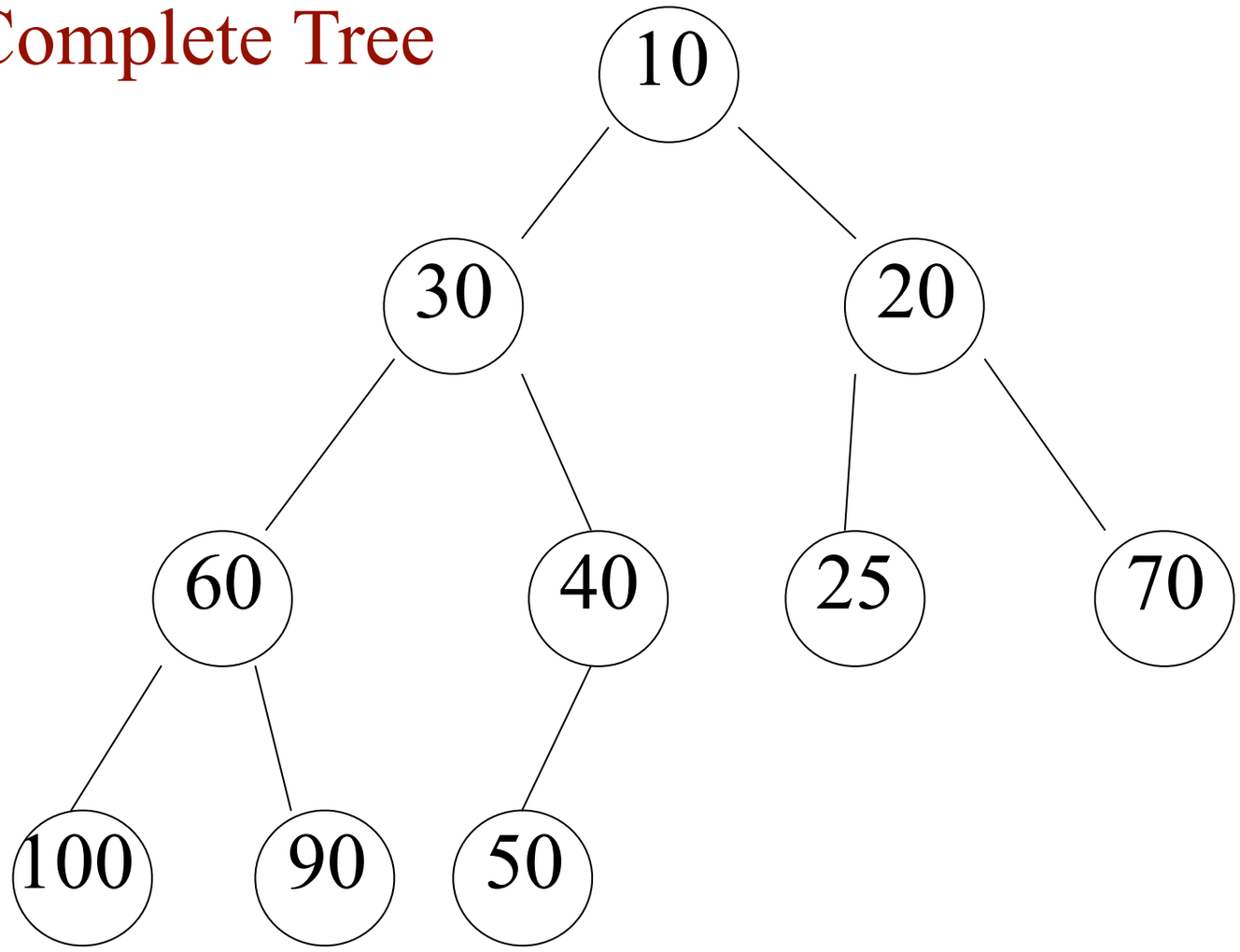
(A tree of height h has at least one node at depth h.)

The depth of the tree is the same as the height of the tree (this is not necessarily true for the height or depth of a node)

Heap-Order Property

In a Heap, for every node X with parent P, the key in P is smaller than or equal to the key in X

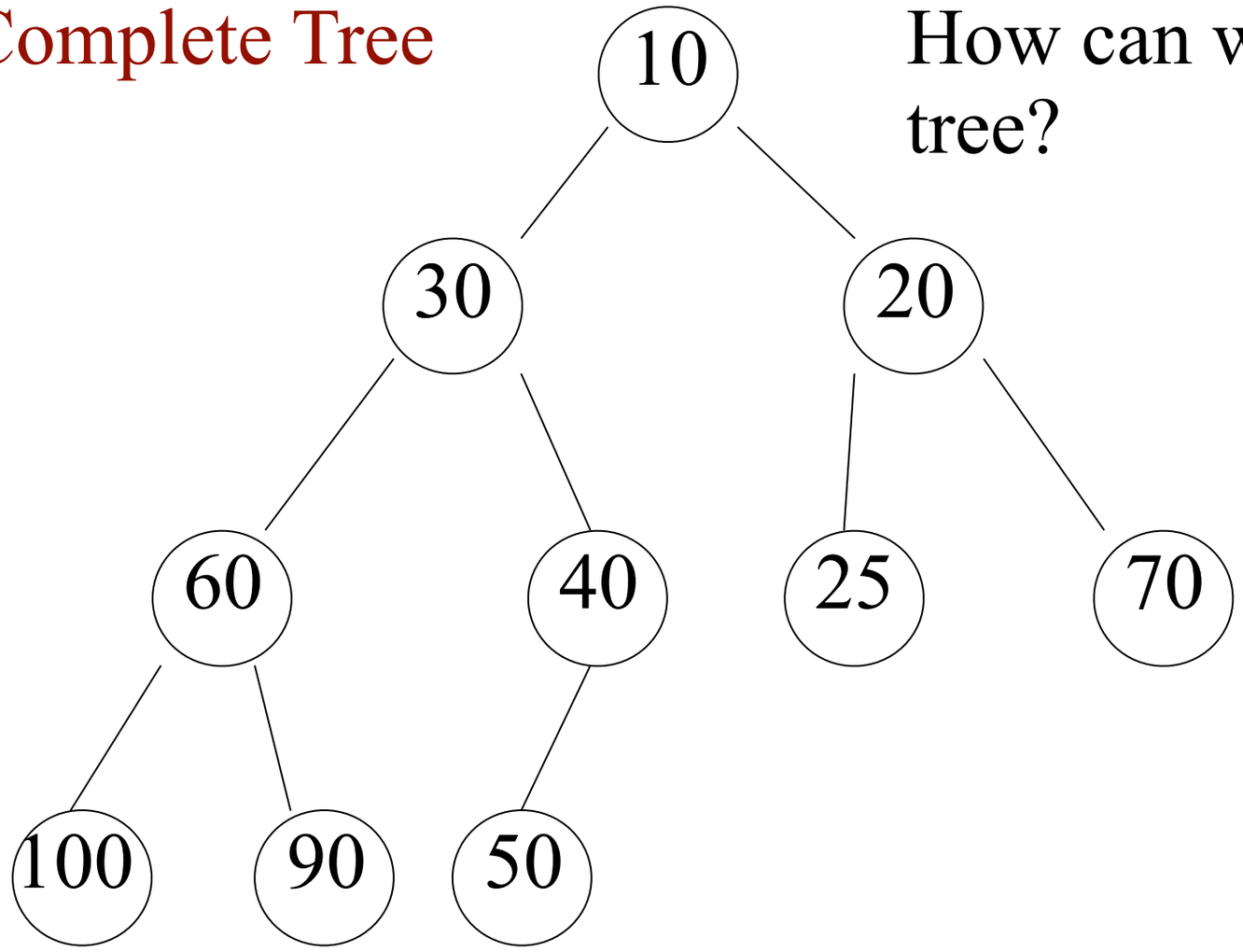
Complete Tree



Implicit Pointers

	sentinel	10	30	20	60	40	25	70	100	90	50
	0	1	2	3	4	5	6	7	8	9	10

Complete Tree



How can we implement this tree?

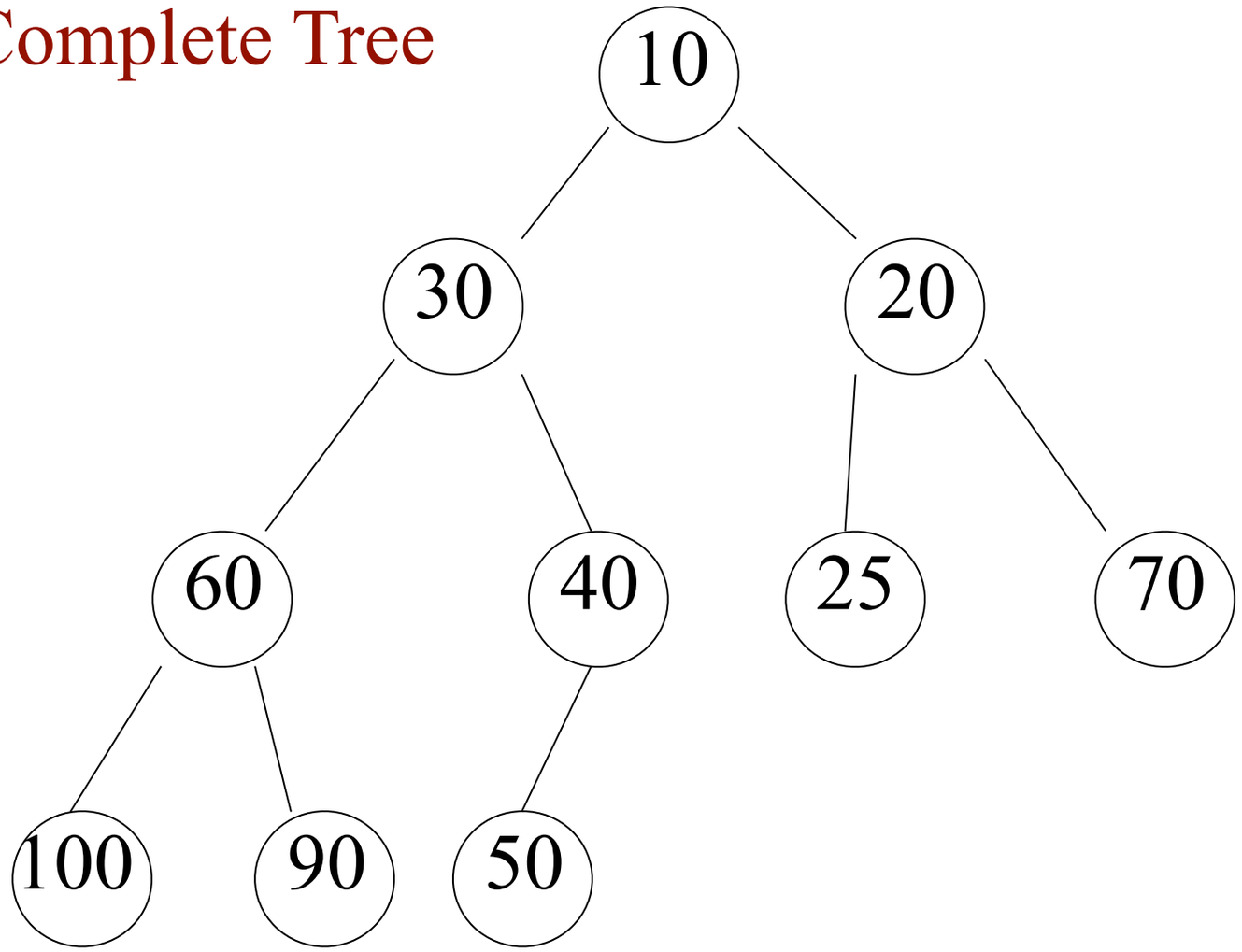
Level order traversal of an array!

sentinel	10	30	20	60	40	25	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

If 30 is found in array position 2, what position is 30's left child?

If 30 is found in array position 2, what position is 30's right child?

Complete Tree



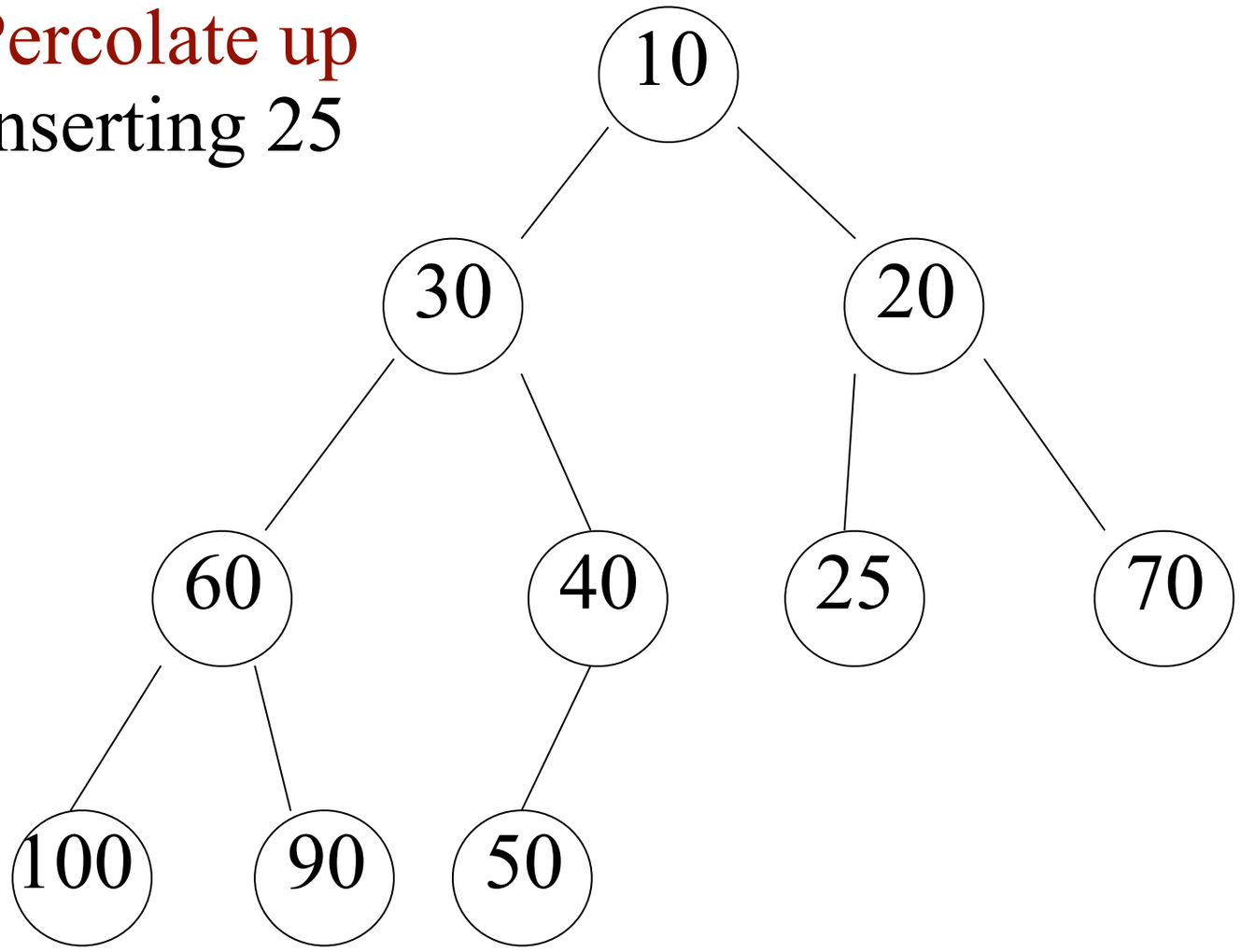
sentinel	10	30	20	60	40	25	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

PQ operations on heaps

- findMin: easy, min element is at root
- insert:
  - put element in next open leaf slot
  - this may violate heap property
  - "percolate up" to restore heap property
- deleteMin
  - clobber root with leaf and decrease size
  - this may violate heap property
  - "percolate down" to restore heap property

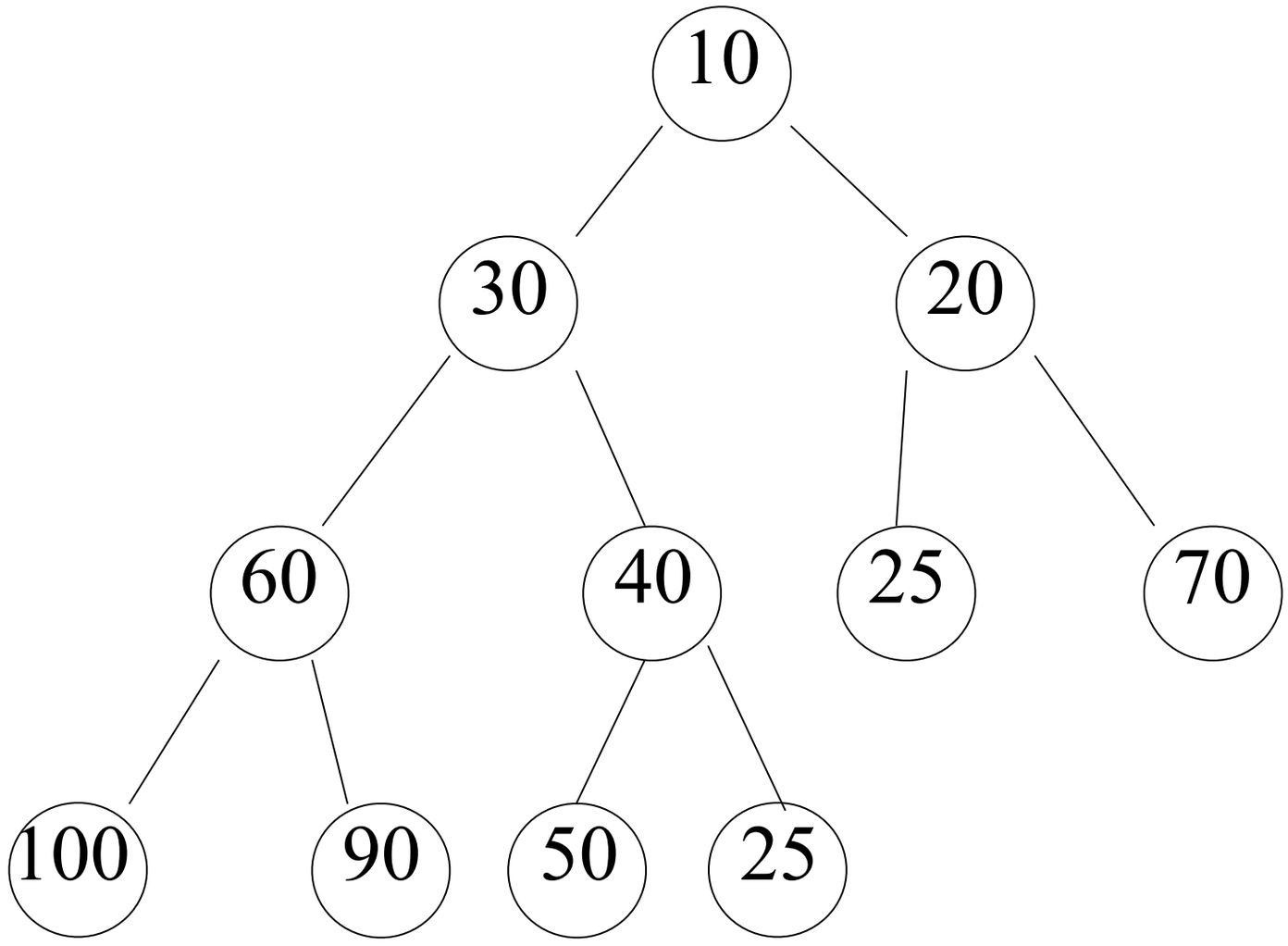
Percolate up

Inserting 25



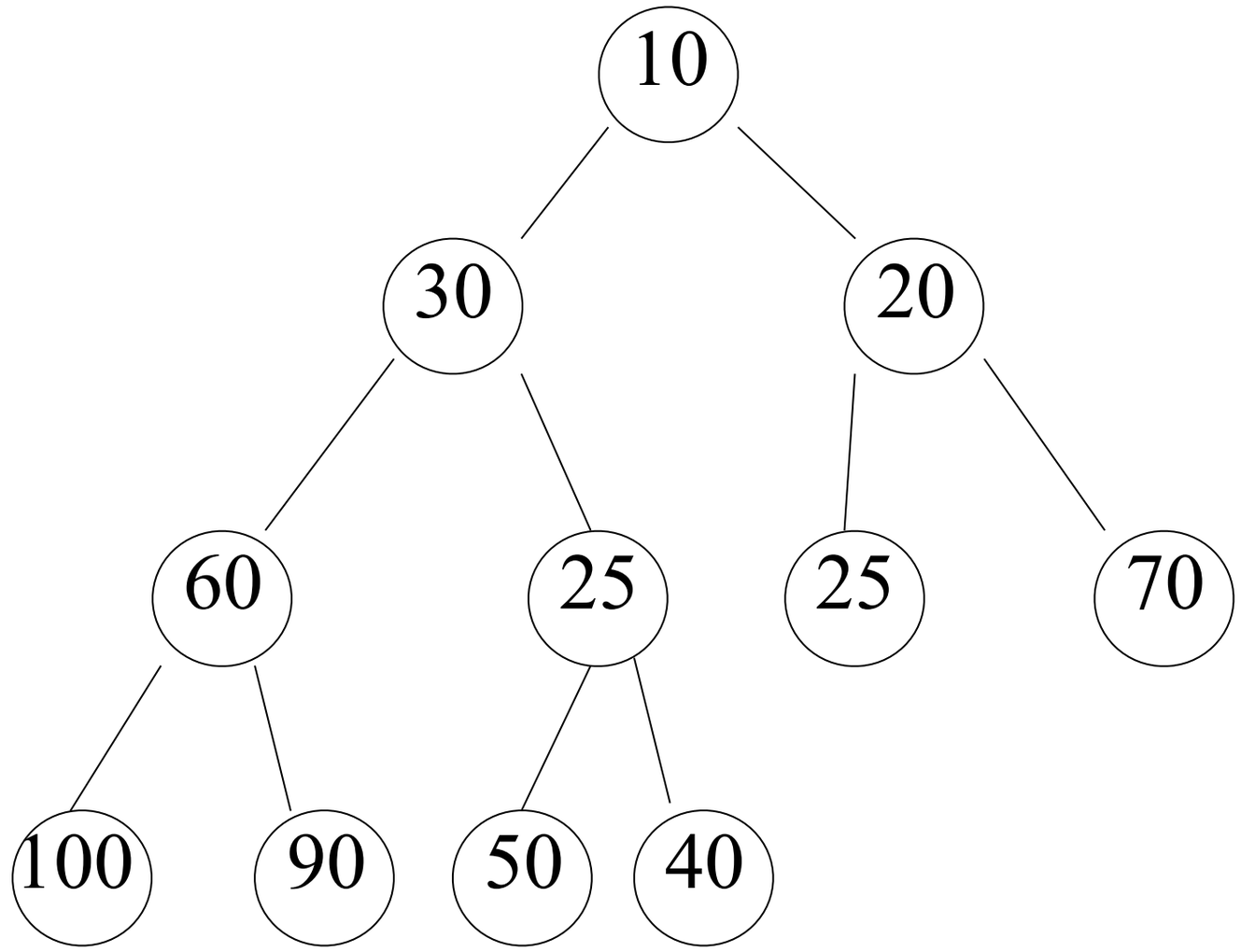
sentinel	10	30	20	60	40	25	70	100	90	50	
0	1	2	3	4	5	6	7	8	9	10	11

Percolate up



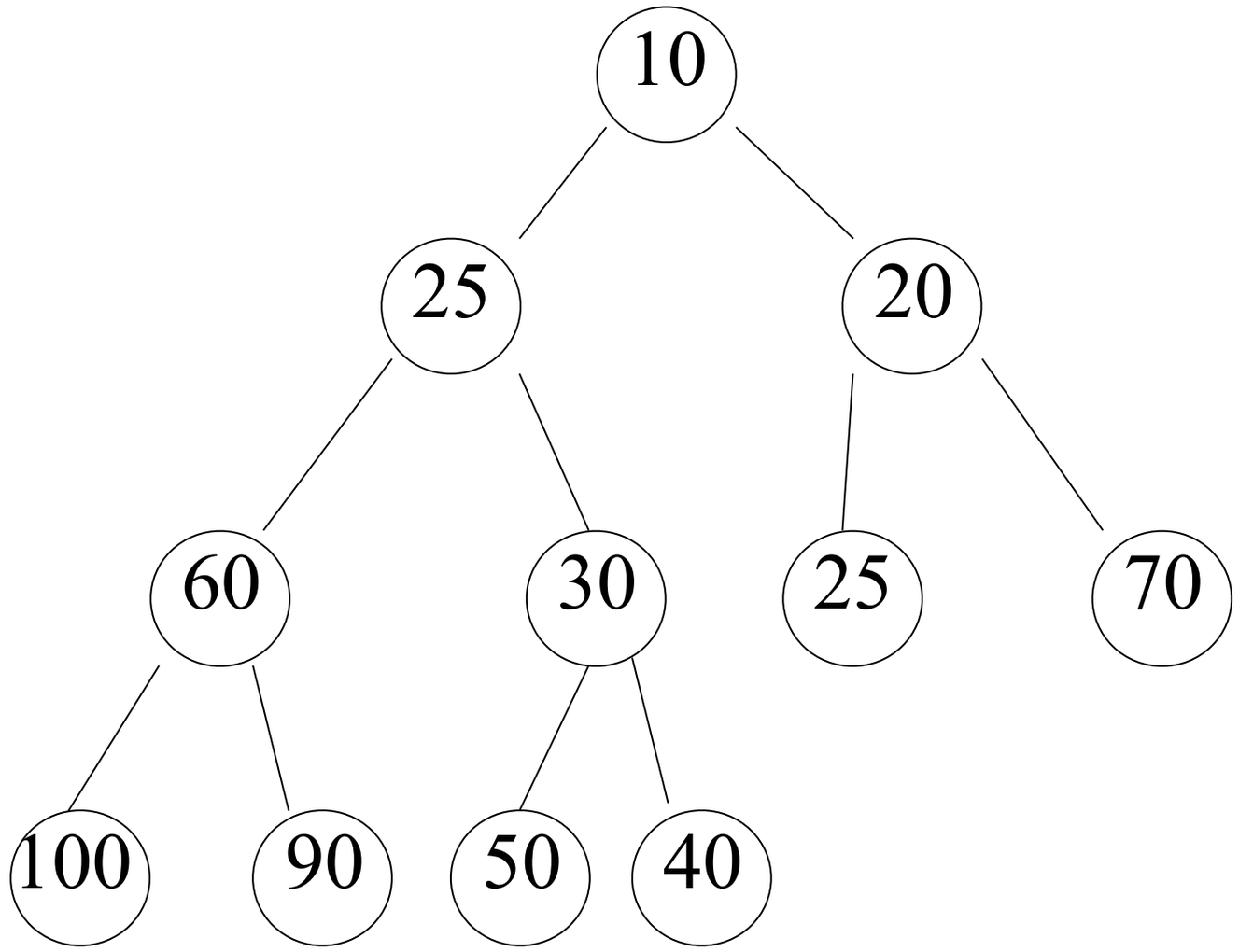
sentinel	10	30	20	60	40	25	70	100	90	50	25
0	1	2	3	4	5	6	7	8	9	10	11

Percolate up



sentinel	10	30	20	60	25	25	70	100	90	50	40
0	1	2	3	4	5	6	7	8	9	10	11

Percolate up



sentinel	10	25	20	60	30	25	70	100	90	50	40
0	1	2	3	4	5	6	7	8	9	10	11

```
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void deleteMin( )      --> Remove smallest item
// void deleteMin( min )  --> Remove and send back smallest item
// Comparable findMin( )  --> Return smallest item
// bool isEmpty( )        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************

template <class Comparable>
class BinaryHeap
{
  public:
    BinaryHeap( );
    BinaryHeap( const vector<Comparable> & v );

    bool isEmpty( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void insert( Comparable && x );

    void deleteMin( );

  private:
    int                theSize;  // Number of elements in heap
    vector<Comparable> array;    // The heap array

    void buildHeap( );
    void percolateDown( int hole );
};
```

tight loop implementation

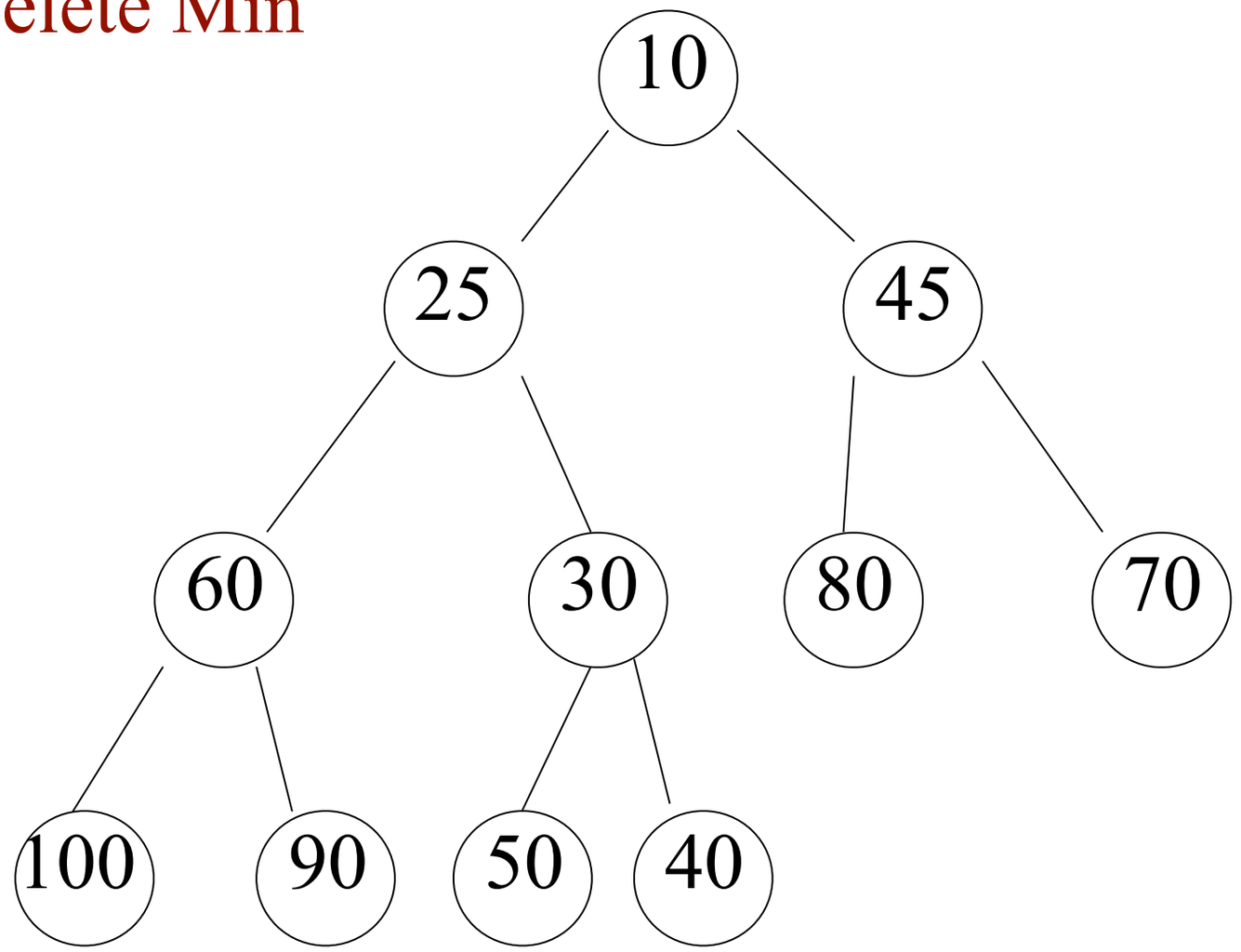
percolate up

```
template <class Comparable>
void BinaryHeap<Comparable>::insert( const Comparable & x )
{
    array[ 0 ] = x;   // initialize sentinel
    if( theSize + 1 == array.size( ) )
        array.resize( array.size( ) * 2 + 1 );

  // Percolate up
    int hole = ++theSize;
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( array[0] );

}
```

Time?
Average time:
1.6 levels
2.6 comparisons
O(1)

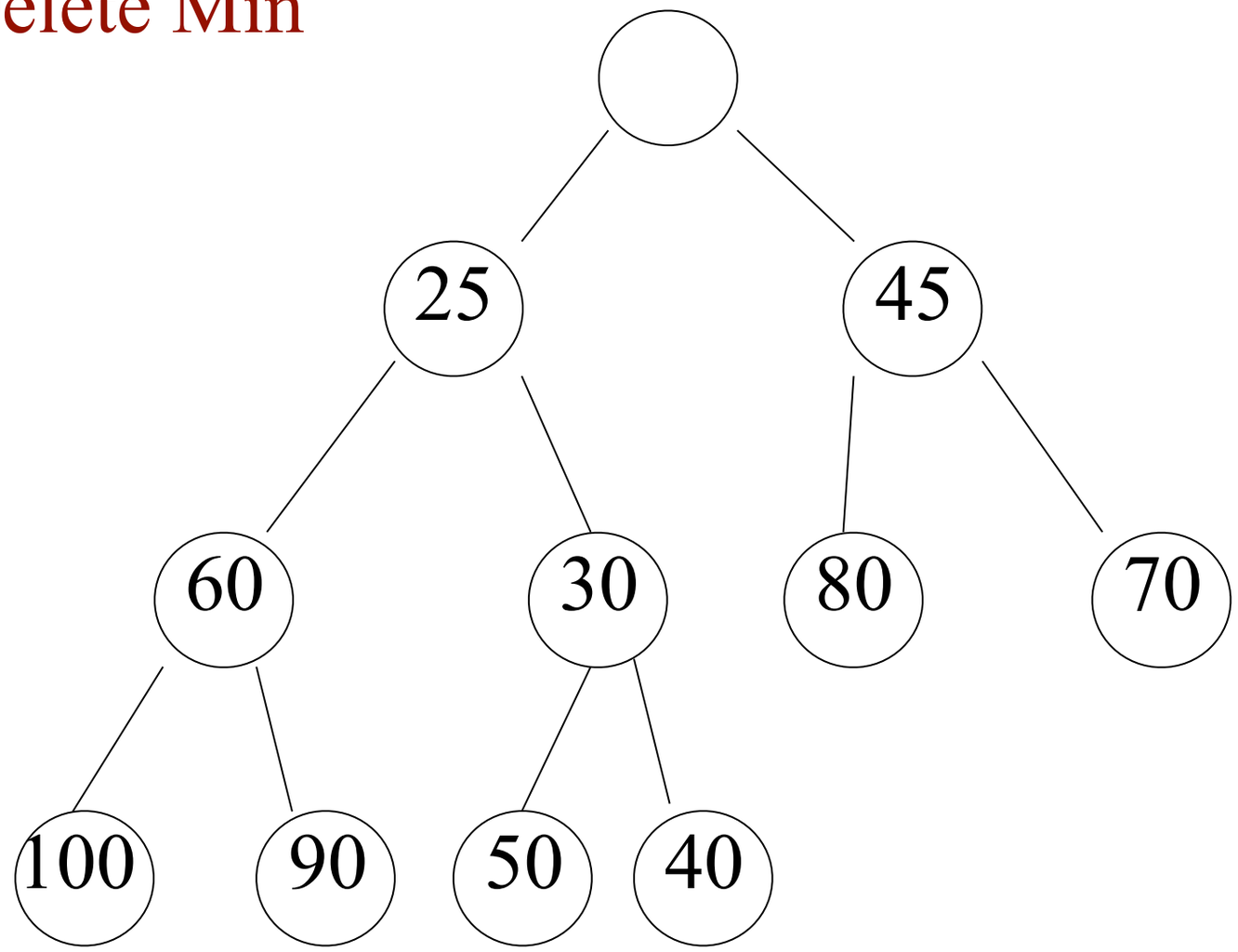
Worst Case time:
O(log(n))
amortized

Delete Min



sentinel	10	25	45	60	30	80	70	100	90	50	40
0	1	2	3	4	5	6	7	8	9	10	11

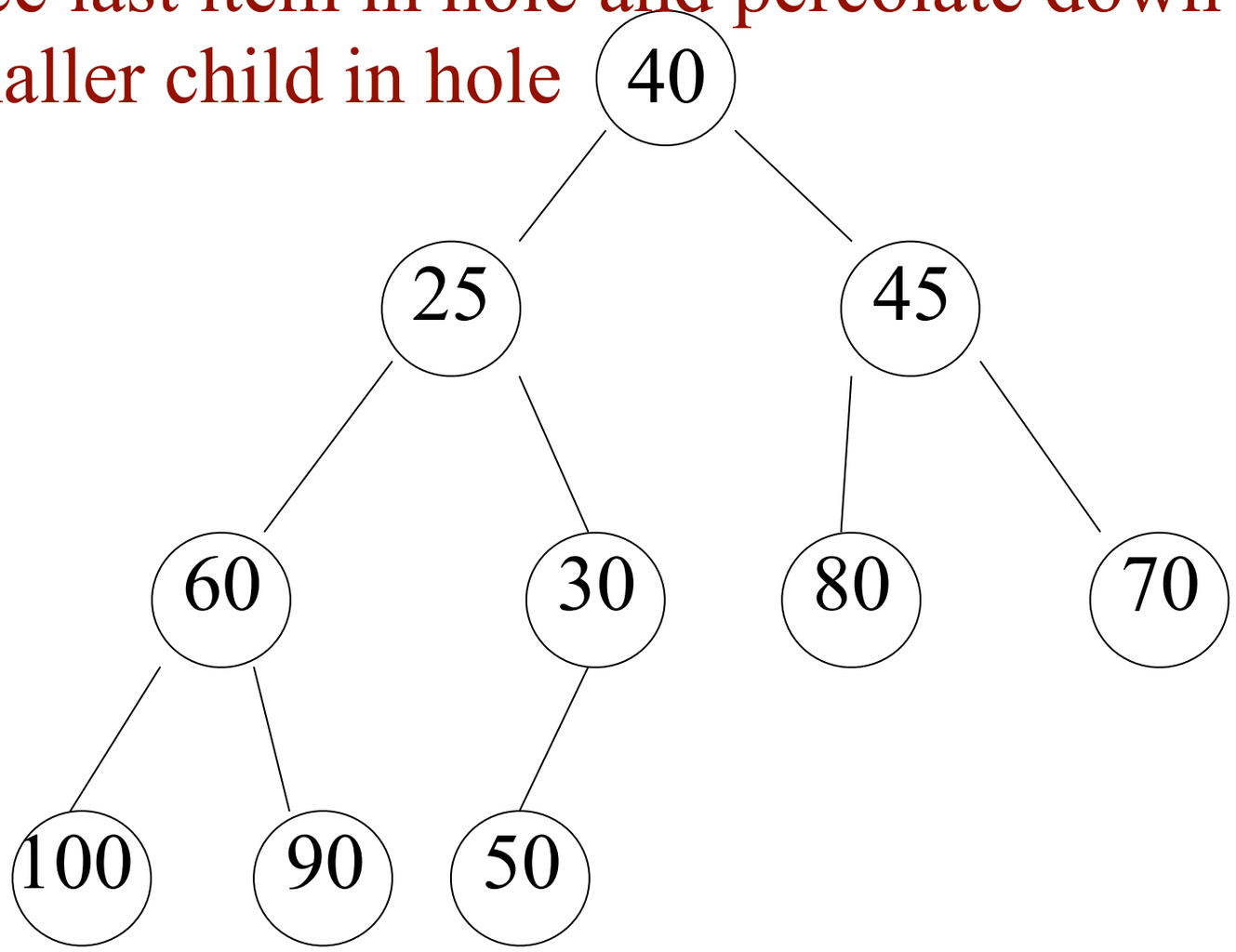
Delete Min



sentinel		25	45	60	30	80	70	100	90	50	40
0	1	2	3	4	5	6	7	8	9	10	11

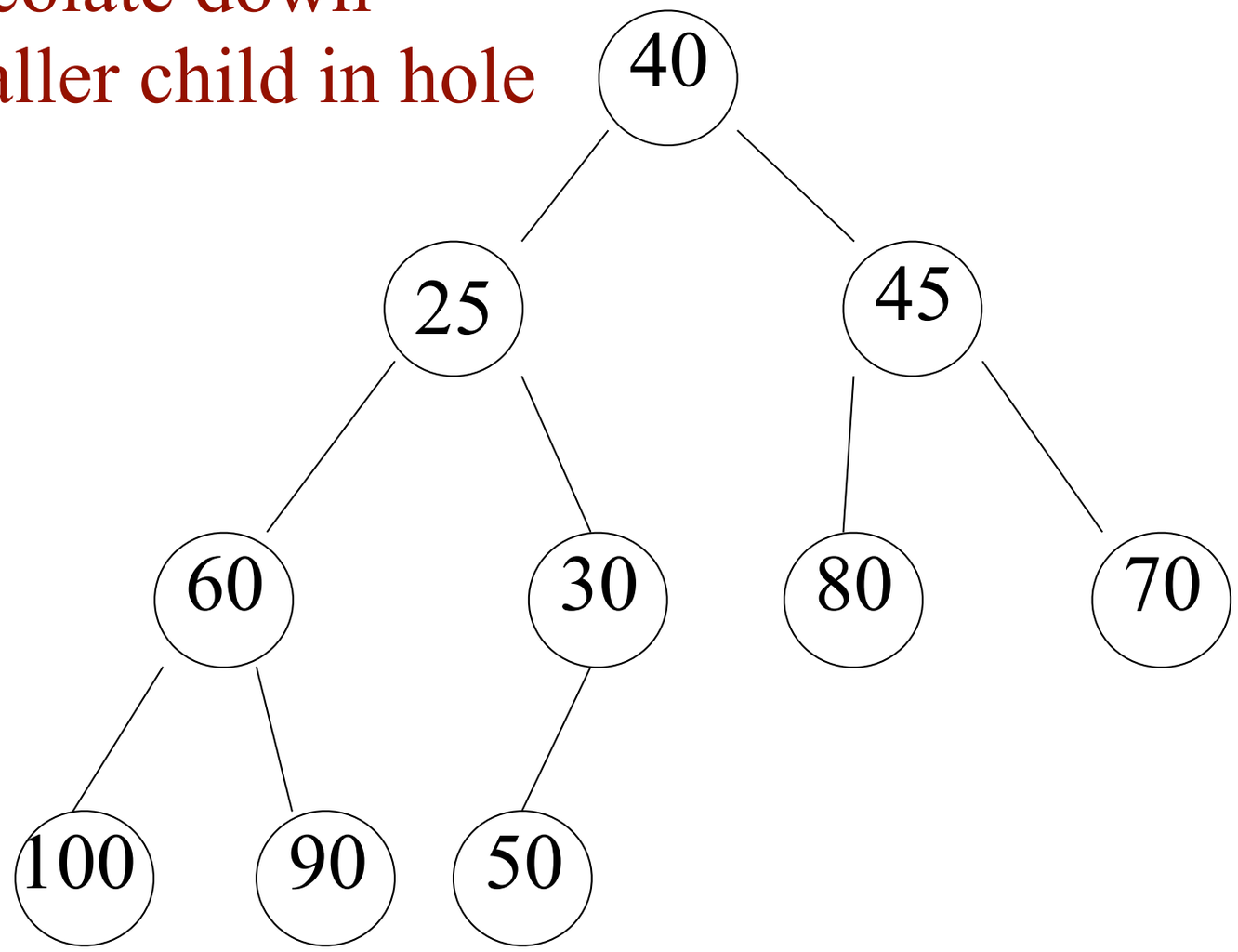
Place last item in hole and percolate down

Smaller child in hole



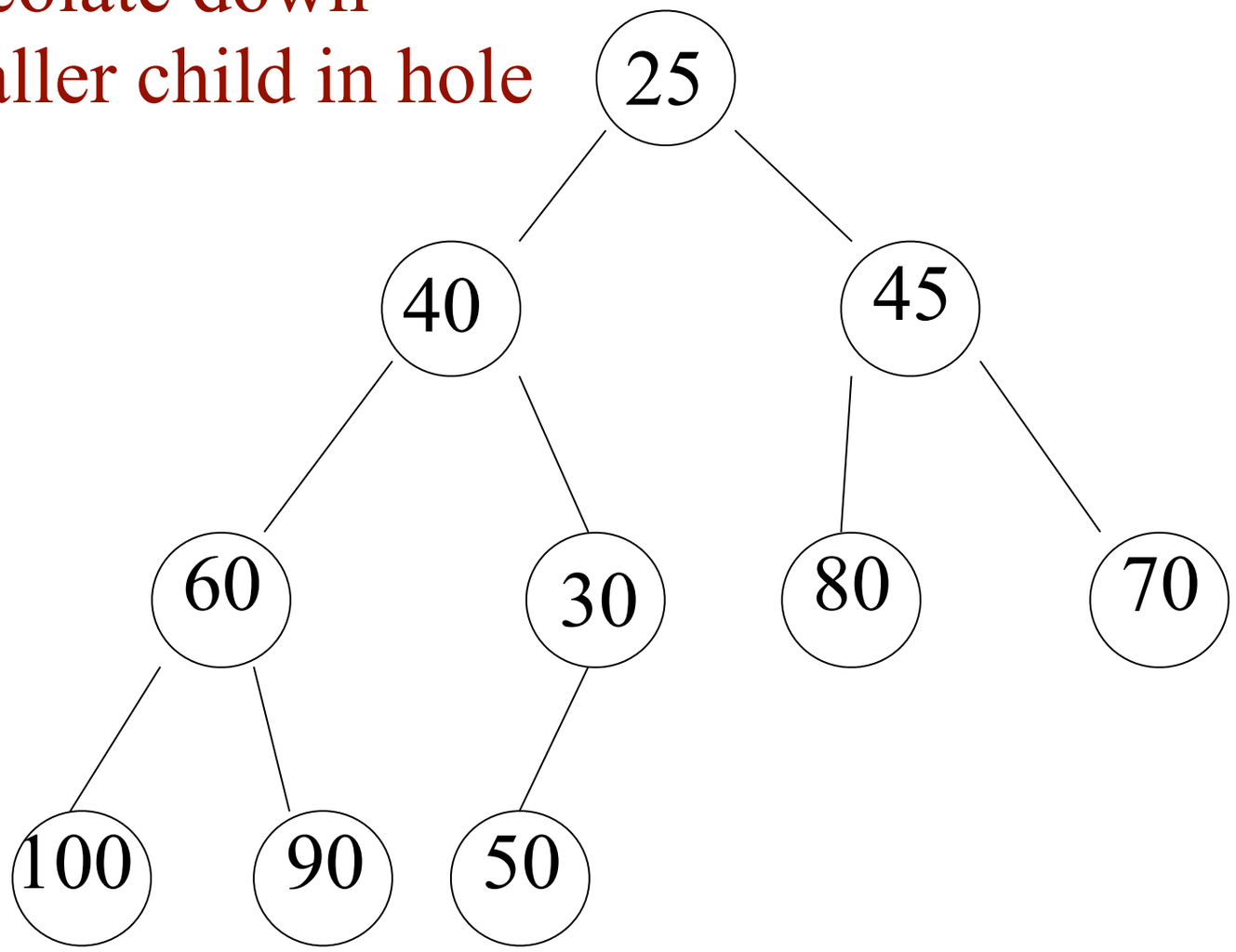
sentinel	40	25	45	60	30	80	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

Percolate down
Smaller child in hole



sentinel	~~40~~ 25	~~25~~ 40	45	60	30	80	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

Percolate down
Smaller child in hole



Time?

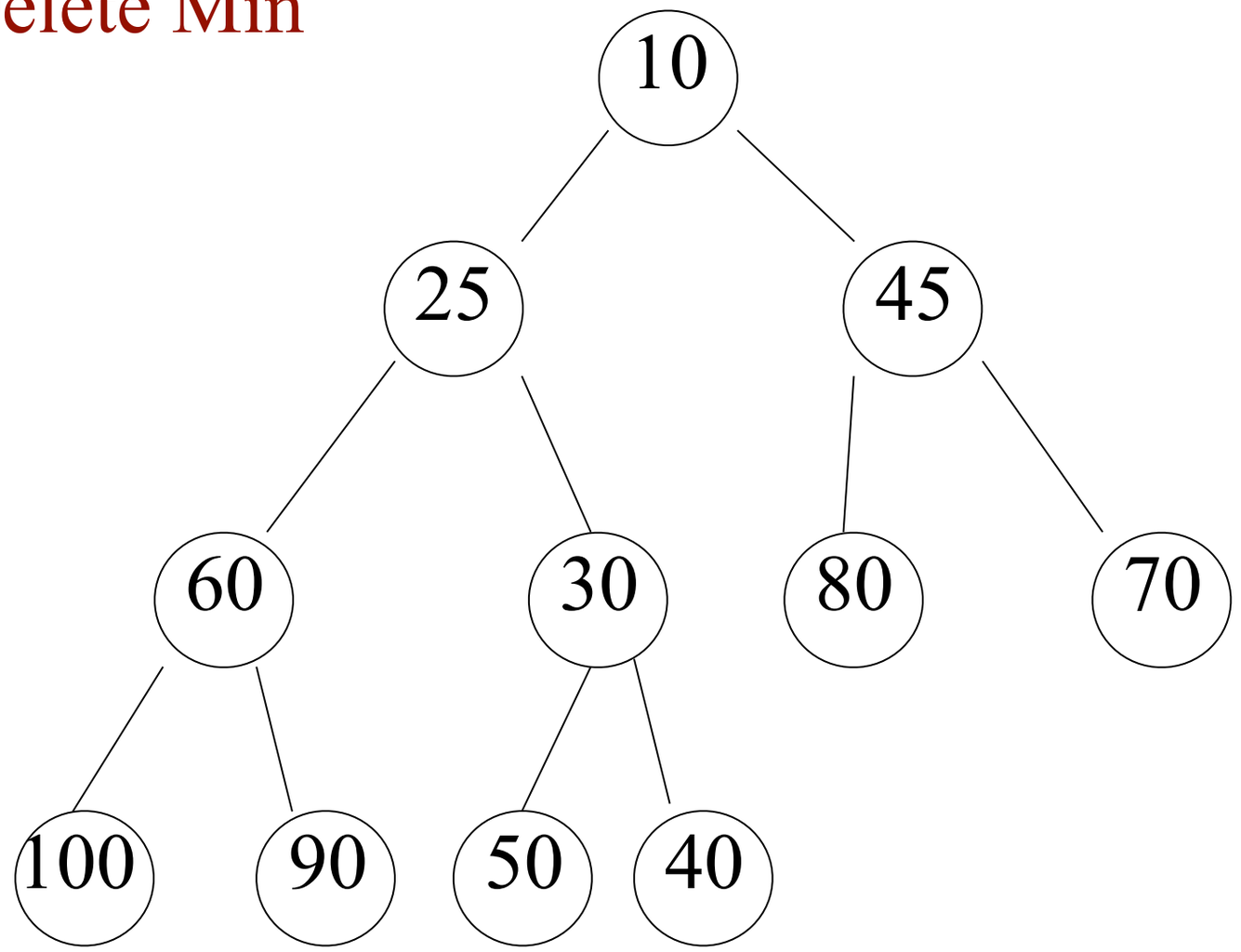
O(log(n))

sentinel	25	30	45	60	40	80	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

Deleting the Minimum Item by "Percolating Down"

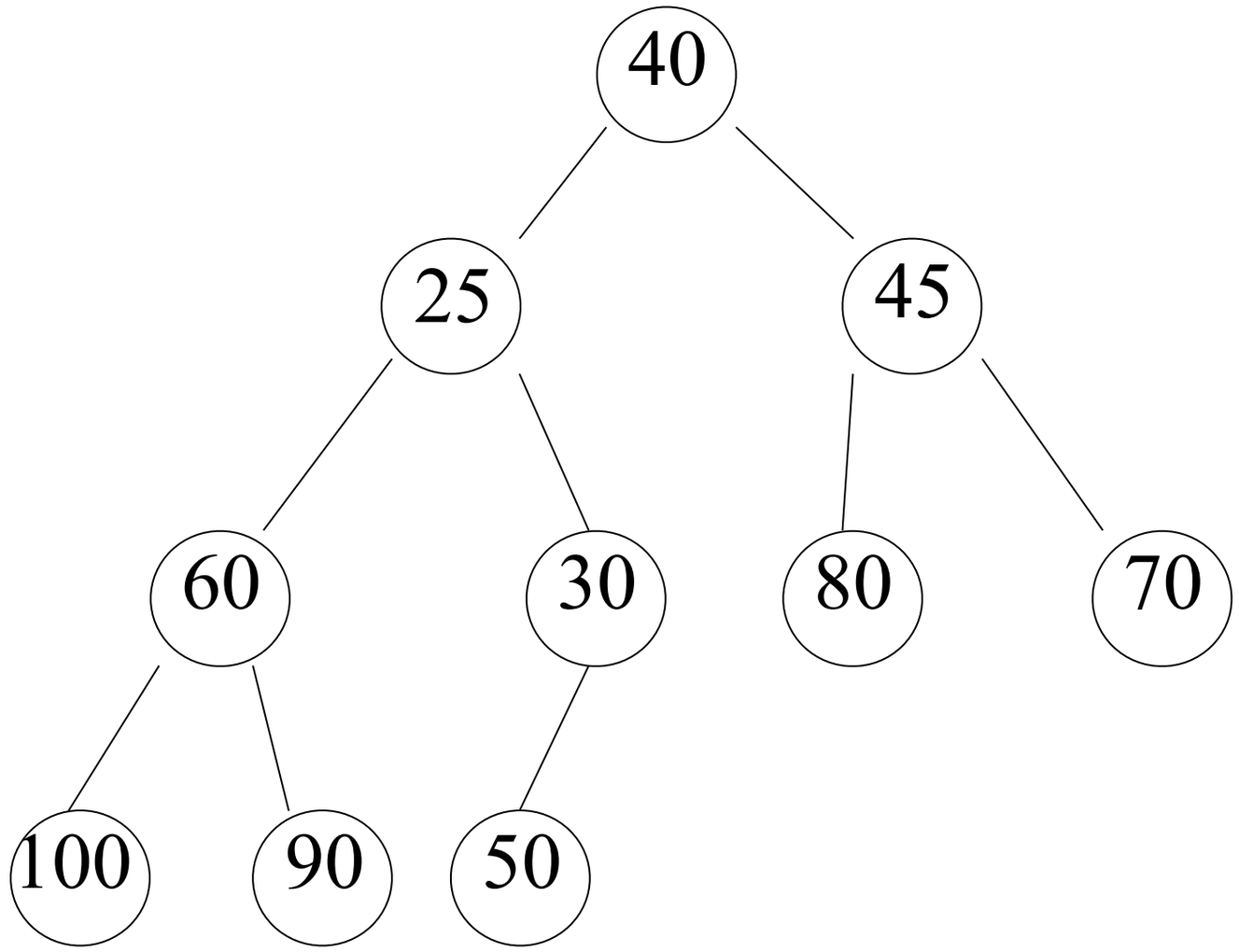
How our code works

Delete Min



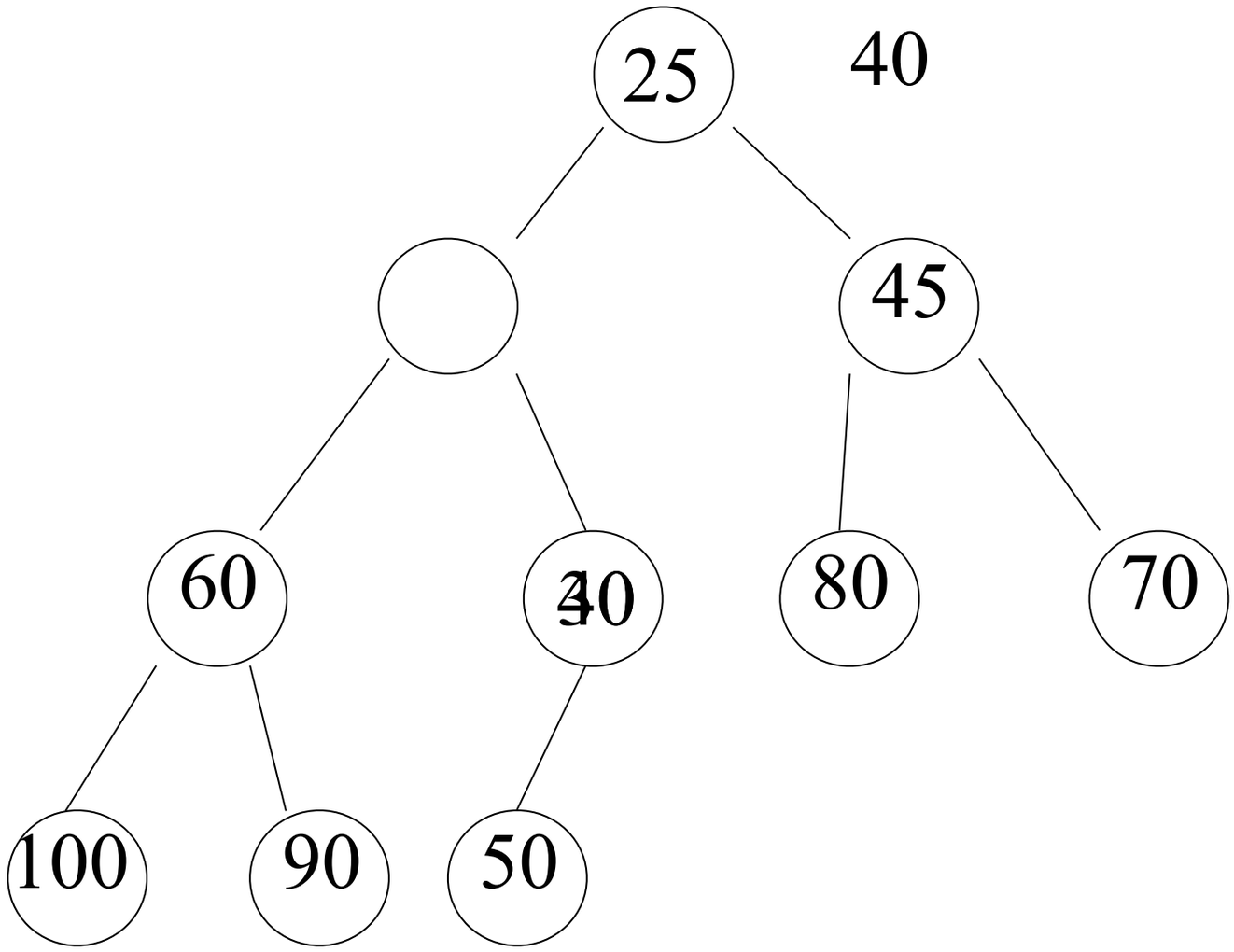
?	40	25	45	60	30	80	70	100	90	50	40
0	1	2	3	4	5	6	7	8	9	10	11

Percolate down by moving smaller child.



?	~~40~~ 25	~~25~~ 40	45	60	30	80	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

Percolate down by moving smaller child.



Time?

O(log(n))

?	25	30	45	60	40	80	70	100	90	50
0	1	2	3	4	5	6	7	8	9	10

```
template <class Comparable>
class BinaryHeap
{
  public:
    BinaryHeap( );
    BinaryHeap( const vector<Comparable> & v );

    bool isEmpty( ) const;
    const Comparable & findMin( ) const;
    void insert( const Comparable & x );
    void insert( Comparable && x );
    void deleteMin( );

  private:
    int                theSize;  // Number of elements in heap
    vector<Comparable> array;    // The heap array

    void buildHeap( );
    void percolateDown( int hole );
};
```

```
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    array[ 1 ] = std::move( array[ theSize-- ] );
    percolateDown( 1 );
}
```

tight loop implementation

percolate down

```
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown( int hole )
{
    int child;
    Comparable tmp = std::move( array[ hole ] );

    for( ; hole * 2 <= theSize; hole = child )
    {
        child = hole * 2;
        if( child != theSize && array[ child + 1 ] < array[ child ] )
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = std::move( array[ child ] );
        else
            break;
    }
    array[ hole ] = std::move( tmp );
}
```

Time?

Average time: O(log(n))

Worst Case time: O(log(n

Complexity and Implementation

- findMin: O(1)
- deleteMin: O(log n)
- insert: O(1) ave, O(log n) amortized worst

- Implementation in array:
  - –elements in slots a[1] ... a[n]
  - –parent of a[i] is in a[i/2]
  - –children of a[j] in a[2j], a[2j +1]
  - –sentinel : in slot 0

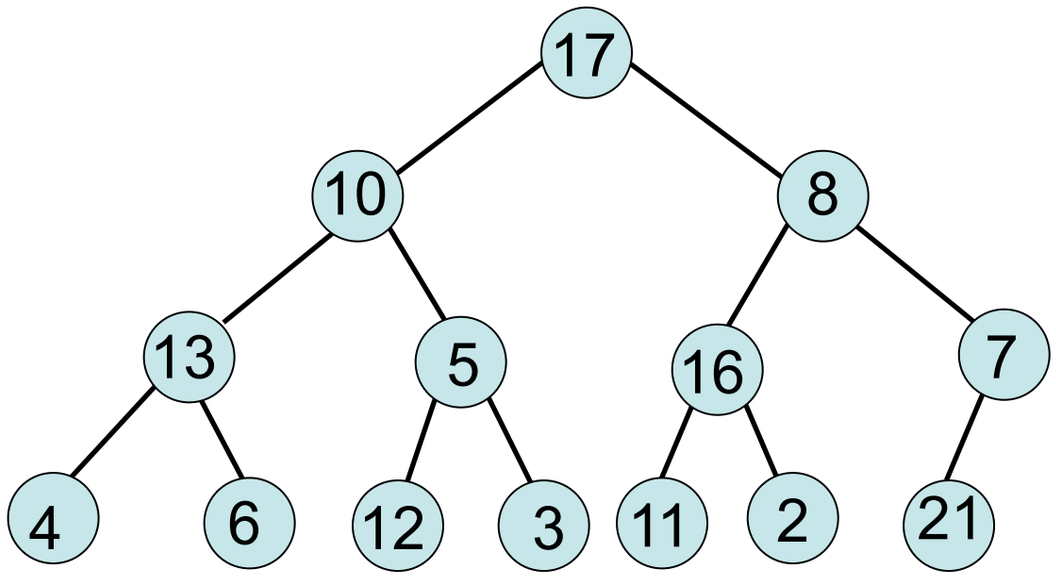
Heapsort

- Sort in descending order
  - –build heap [Can be done in time O(n) – see next slide]
  - –While heap not empty
    - remove min
    - place min in array position
      - that was just “freed up”

(Technicality – begin heap at position 0 of array, not position 1, to use just original array entries)

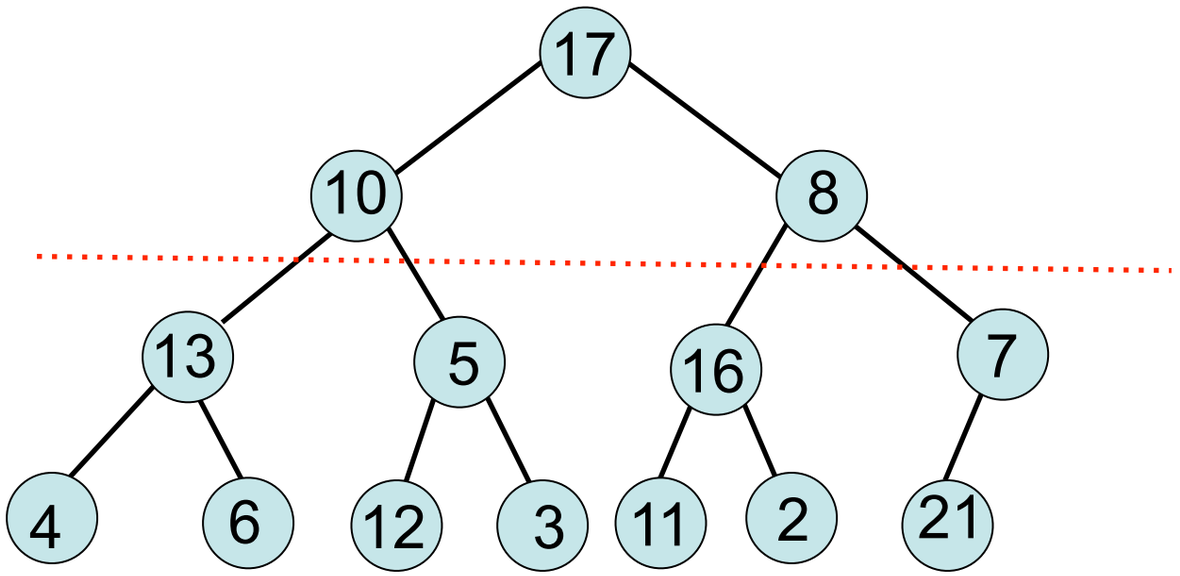
- To sort in ascending order, use maxheap

buildHeap



0	1	2	3	5	6	7	8	9	10	11	12	13	14	15
-inf	17	10	8	13	5	16	7	4	6	12	3	11	2	21

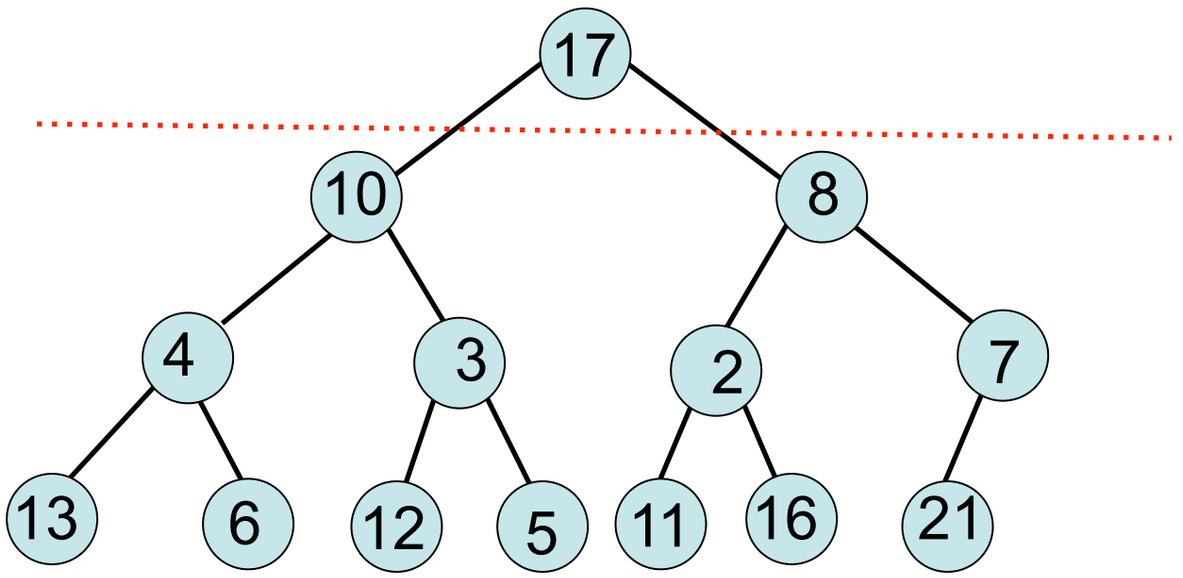
buildHeap



`percolateDown` all trees below the dotted line

0	1	2	3	5	6	7	8	9	10	11	12	13	14	15
-inf	17	10	8	13	5	16	7	4	6	12	3	11	2	21
-inf	17	10	8	13	5	2	7	4	6	12	3	11	16	21
-inf	17	10	8	13	3	2	7	4	6	12	5	11	16	21
-inf	17	10	8	4	3	2	7	13	6	12	5	11	16	21

buildHeap



`percolateDown` all trees below the dotted line

0	1	2	3	5	6	7	8	9	10	11	12	13	14	15
-inf	17	10	8	4	3	2	7	13	6	12	5	11	16	21

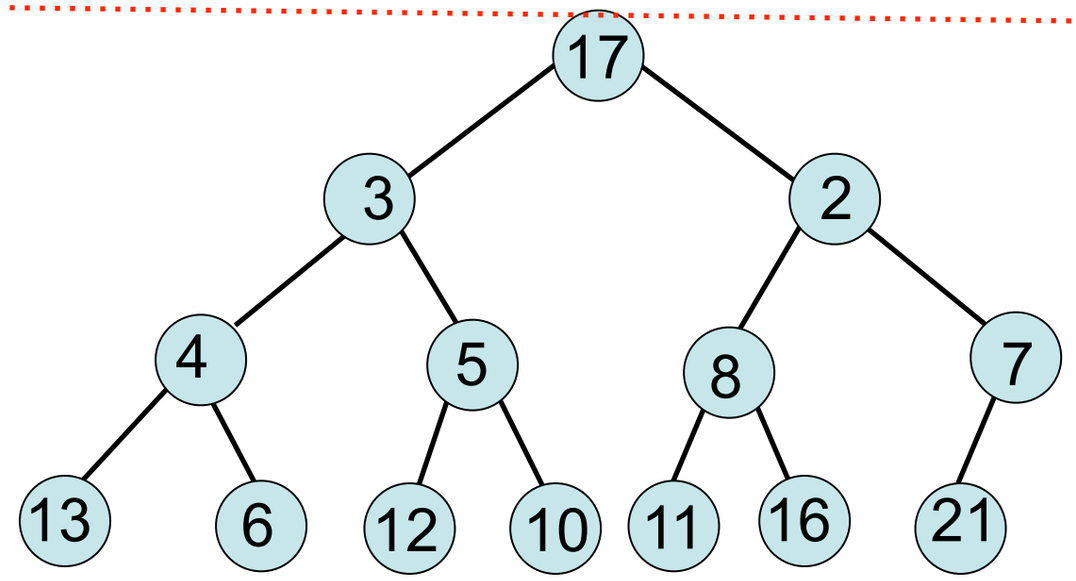
-inf	17	10	2	4	3	8	7	13	6	12	5	11	16	21

-inf	17	3	2	4	10	8	7	13	6	12	5	11	16	21

-inf	17	3	2	4	5	8	7	13	6	12	10	11	16	21

buildHeap

`percolateDown` all trees below the dotted line



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-inf	17	3	2	4	5	8	7	13	6	12	10	11	16	21

-inf	2	3	17	4	5	8	7	13	6	12	10	11	16	21

-inf	2	3	7	4	5	8	17	13	6	12	10	11	16	21

BuildHeap

- Takes unsorted vector and turns it into heap
- Idea
  - For all levels of the tree, beginning at the second to the bottom:
    - Percolate down all elements at that level
- Running time is O(n)
- Most percolate-downs are near leaves

```
template <class Comparable>
class BinaryHeap
{
  public:
    BinaryHeap( );
    BinaryHeap( const vector<Comparable> & v );

    bool isEmpty( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void deleteMin( );

  private:
    int                theSize;  // Number of elements in heap
    vector<Comparable> array;    // The heap array

    void buildHeap( );
    void percolateDown( int hole );
};
```

```
// Construct the binary heap.
// v is a vector containing the initial items.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( const vector<Comparable> & v )
  : array( v.size( ) + 1 ), theSize( v.size( ) )
{
    for( int i = 0; i < v.size( ); i++ )
        array[ i + 1 ] = v[ i ];
    buildHeap( );
}
```

buildHeap

```
template <class Comparable>
void BinaryHeap<Comparable>::buildHeap( )
{
  for( int i = theSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Running Time?

Each call to `percolateDown` takes O(log(n)) time

There are at most n calls to `percolateDown`

Total time is at most O(n log(n))

Running Time?

Tighter analysis shows that building the heap takes O(n) time

STL priority_queue
priority_queue<type,container,functor>

Worst Case - using the vector container

- push() O(log(n)) amortized
- pop() O(log(n))
- top() O(1)
- empty() O(1)
- size() O(1)
- priority_queue<type> p(itrB,itrE)

  O(n) where n is the number of items in the range [itrB, itrE)