Competitive Programming

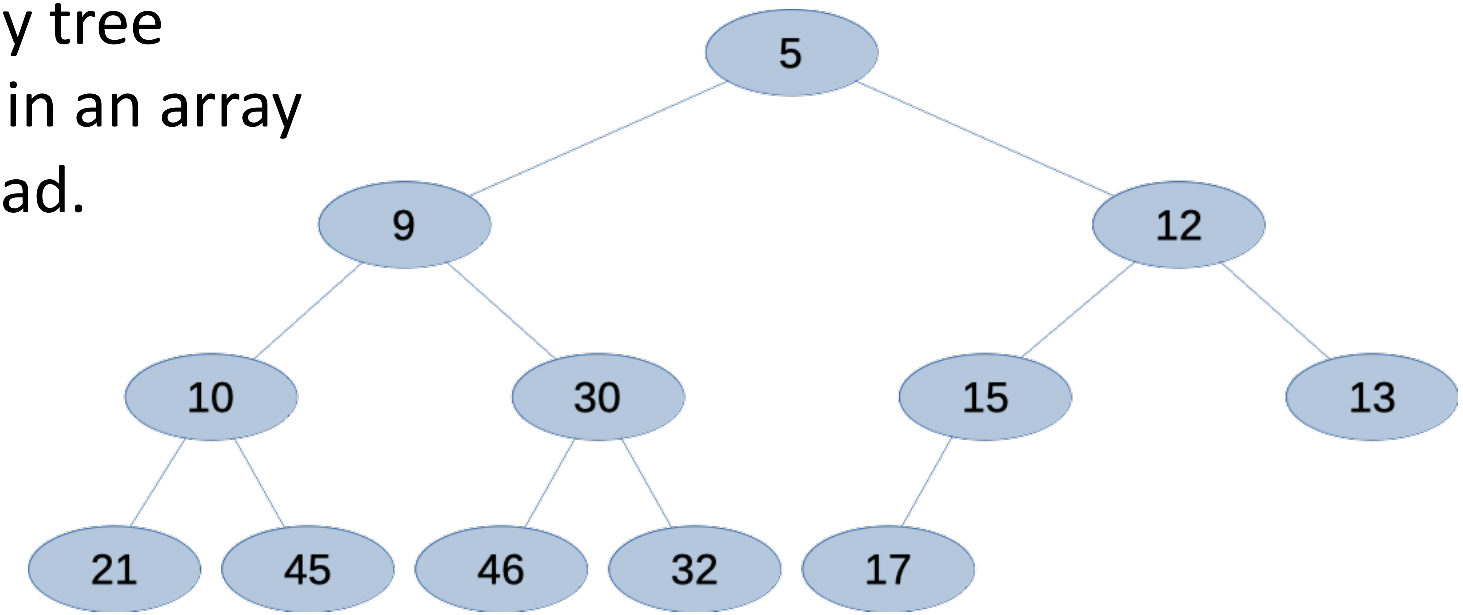
Heap Data Structure

Data Structures in Competitive Programming

- There are a (surprisingly small) number of data structures essential to competitive programming.
- You should:
  - Be familiar with them
  - Know the asymptotic performance of all their standard operations.
  - Be able to implement them if needed
  - Avoid implementing them if at all possible
  - … instead, use an existing, standard implementation

The Heap

- The heap is a simple, efficient way of
  - Maintaining a set of values
  - Efficiently adding values
  - Efficiently finding and extracting the minimum (or maximum)
- A common implementation of a priority queue.
- It’s implemented as a binary tree
- An implicit tree embedded in an array
- … so it has very low overhead.
- It’s easy to implement
- … but you should avoid implementing it.



Re-using a Heap (in C++)

```
#include <cstdio>
#include <vector>
#include <queue>

using namespace std;

int main()
{
  priority_queue< int, vector< int >, greater<int> > Q;

  Q.push( 5 );
  Q.push( 25 );
  Q.push( 19 );
  Q.push( 2 );
  Q.push( 8 );
  Q.push( 27 );
```

Re-using a Heap (in C++)

```
 while ( Q.size() ) {
    printf( "%d\n", Q.top() );
    Q.pop();
  }

  return 0;
}
```

Re-using a Heap (in Java)

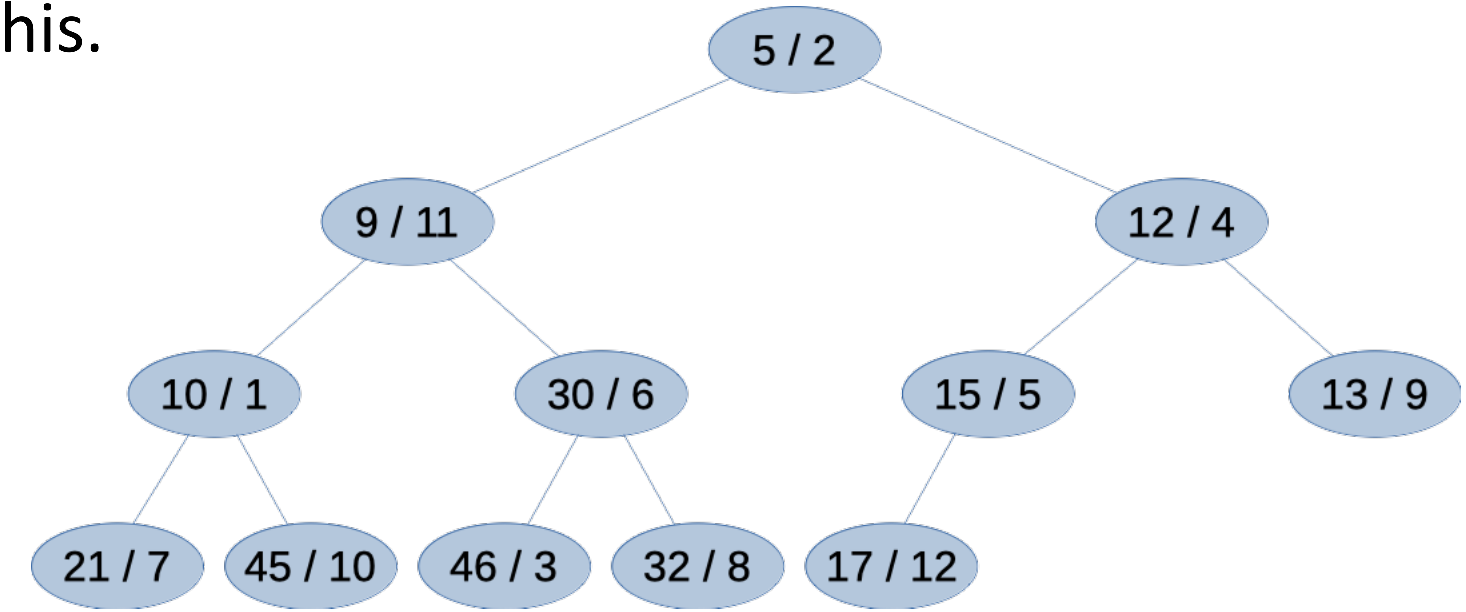
```
import java.util.*;

class Heap1 {
  public static void main( String[] args ) {
    PriorityQueue<Integer> Q = new PriorityQueue<>();

    Q.add( 5 );
    Q.add( 25 );
    Q.add( 19 );
    Q.add( 2 );
    Q.add( 8 );
    Q.add( 27 );

    while ( Q.size() > 0 ) {
      System.out.println( Q.peek() );
      Q.remove();
    }
  } // One more }
```

Re-using a Heap (in Python)

```
import heapq

Q = []

heapq.heappush( Q, 5 )
heapq.heappush( Q, 25 )
heapq.heappush( Q, 19 )
heapq.heappush( Q, 2 )
heapq.heappush( Q, 8 )
heapq.heappush( Q, 27 )

while Q:
    print( Q[ 0 ] )
    heapq.heappop( Q )
```

Augmenting the Heap

- Often, you need your data structure to maintain some auxiliary values
  - One value to govern the heap organization (i.e., a value we minimize over)
  - One or more other values that go along for the ride.
- Your preferred heap implementation probably provides a way of doing this.



Heap with extra value (in C++)

```
#include <cstdio>
#include <vector>
#include <queue>

using namespace std;

int main()
{
  priority_queue< pair< int, int >, vector< pair< int, int > >,
                  greater< pair< int, int > > > Q;

  Q.push( make_pair( 5, 1 ) );
  Q.push( make_pair( 25, 2 ) );
  Q.push( make_pair( 19, 3 ) );
  Q.push( make_pair( 2, 4 ) );
  Q.push( make_pair( 8, 5 ) );
  Q.push( make_pair( 27, 6 ) );
```

Heap with extra value (in C++)

```
  while ( Q.size() ) {
    printf( "%d %d\n", Q.top().first, Q.top().second );
    Q.pop();
  }

  return 0;
}
```

Re-using a Heap (in Java)

```
import java.util.*;

class Heap2 {
  static class Pair implements Comparable< Pair > {
    int a, b;

    public Pair( int va, int vb ) {
      a = va; b = vb;
    }

    public int compareTo( Pair p ) {
      return Integer.compare( a, p.a );
    }
  };
```

Re-using a Heap (in Java)

```
  public static void main( String[] args ) {
    PriorityQueue<Pair> Q = new PriorityQueue<>();

    Q.add( new Pair( 5, 1 ) );
    Q.add( new Pair( 25, 2 ) );
    Q.add( new Pair( 19, 3 ) );
    Q.add( new Pair( 2, 4 ) );
    Q.add( new Pair( 8, 5 ) );
    Q.add( new Pair( 27, 6 ) );

    while ( Q.size() > 0 ) {
      System.out.println( Q.peek().a + " " + Q.peek().b );
      Q.remove();
    }
  }
}
```

Re-using a Heap (in Python)

```
import heapq

Q = []

heapq.heappush( Q, ( 5, 1 ) )
heapq.heappush( Q, ( 25, 2 ) )
heapq.heappush( Q, ( 19, 3 ) )
heapq.heappush( Q, ( 2, 4 ) )
heapq.heappush( Q, ( 8, 5 ) )
heapq.heappush( Q, ( 27, 6 ) )

while Q:
    print( "%d %d" % ( Q[ 0 ][ 0 ], Q[ 0 ][ 1 ] ) )
    heapq.heappop( Q )
```