Priority Queue

What is that?
Implementation with linked list with O(n) behaviour
The Heap (O(log(n))
An implementation using an array
Your mission …

Priority queue

- Store a collection of prioritized elements
  - Elements are comparable
- Allow insertion of an element
- Can only remove the element with highest priority
  - Comes first in order

```
Q.insert(e)
e = Q.removeMin()
Q.size()
Q.isEmpty()
Q.min()
```

We present 3 implementations

Priority queue

Example applications of a priority queue

- Dispatching processes in a computer
- Hospital waiting lists
- Standby passengers for a flight
- Queuing at call centres
- Internal data structure for another algorithm (graph algorithm)
- …

How can we compare objects?

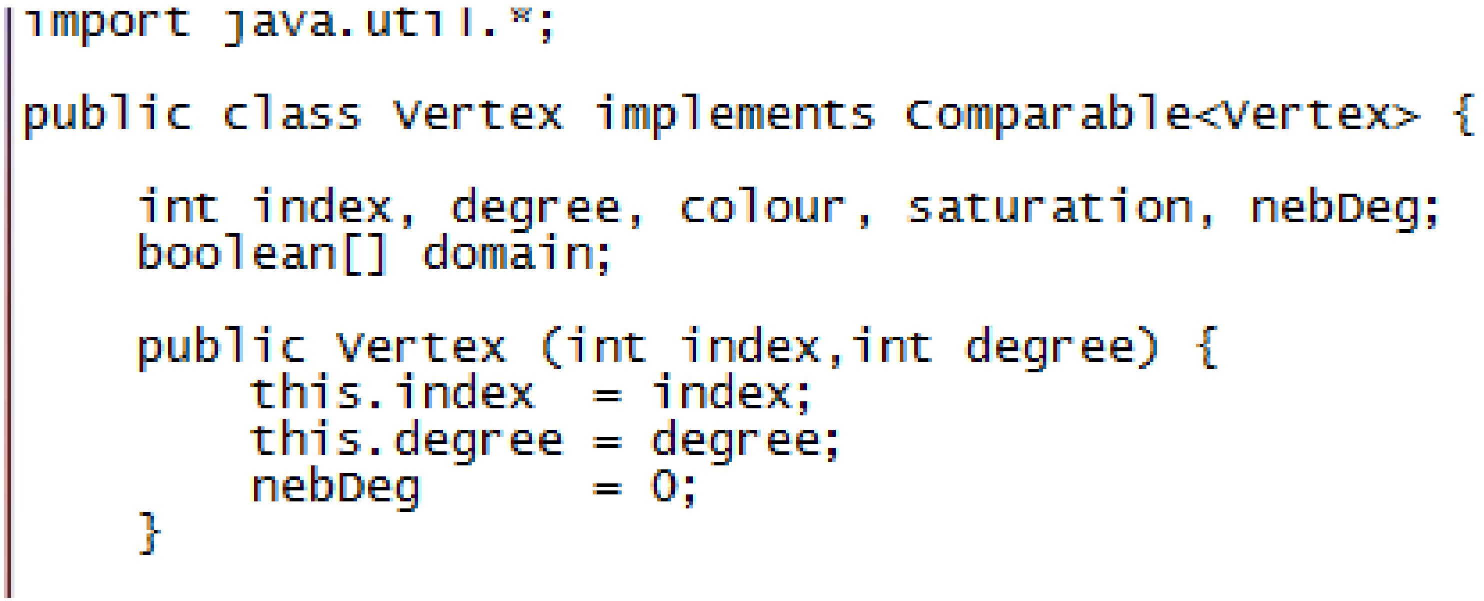
- Store a collection of prioritized elements
    - ***Elements are comparable***
- Allow insertion of an element
- Can only remove the element with highest priority
    - Comes first in order

Two examples on ***comparing things*** … …

How can we compare objects?

An example

This is a Vertex

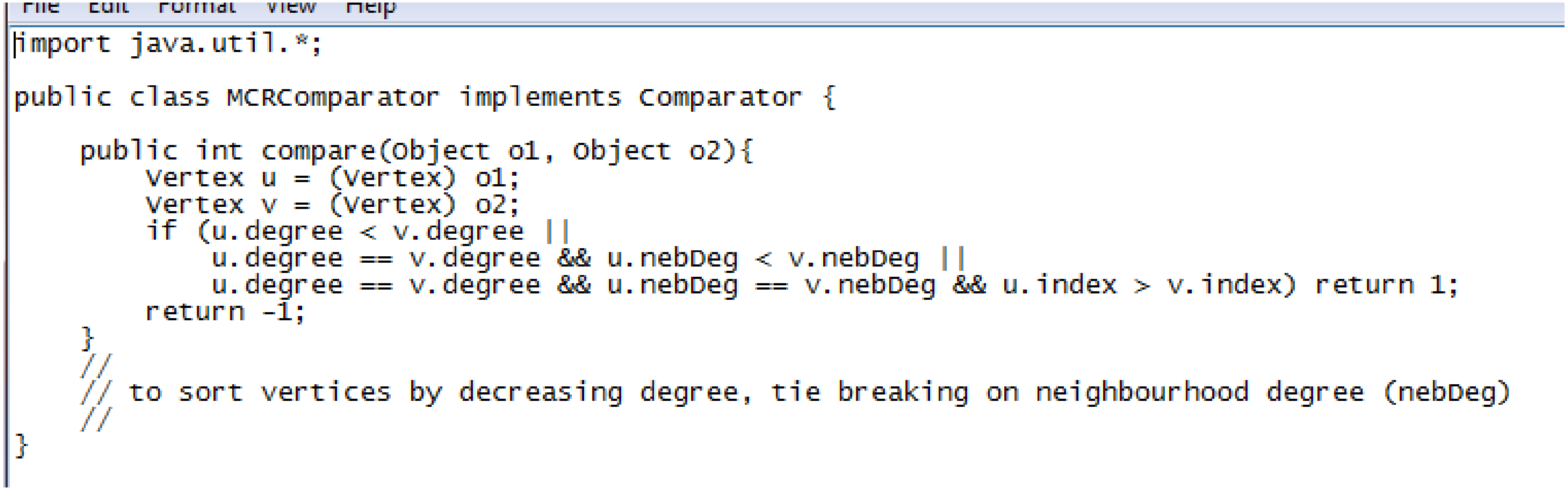


```
import java.util.*;

public class Vertex implements Comparable<Vertex> {

    int index, degree, colour, saturation, nebDeg;
    boolean[] domain;

    public Vertex (int index,int degree) {
        this.index  = index;
        this.degree = degree;
        nebDeg      = 0;
    }
```

How can we compare objects?

An example

This is a Comparator for vertices

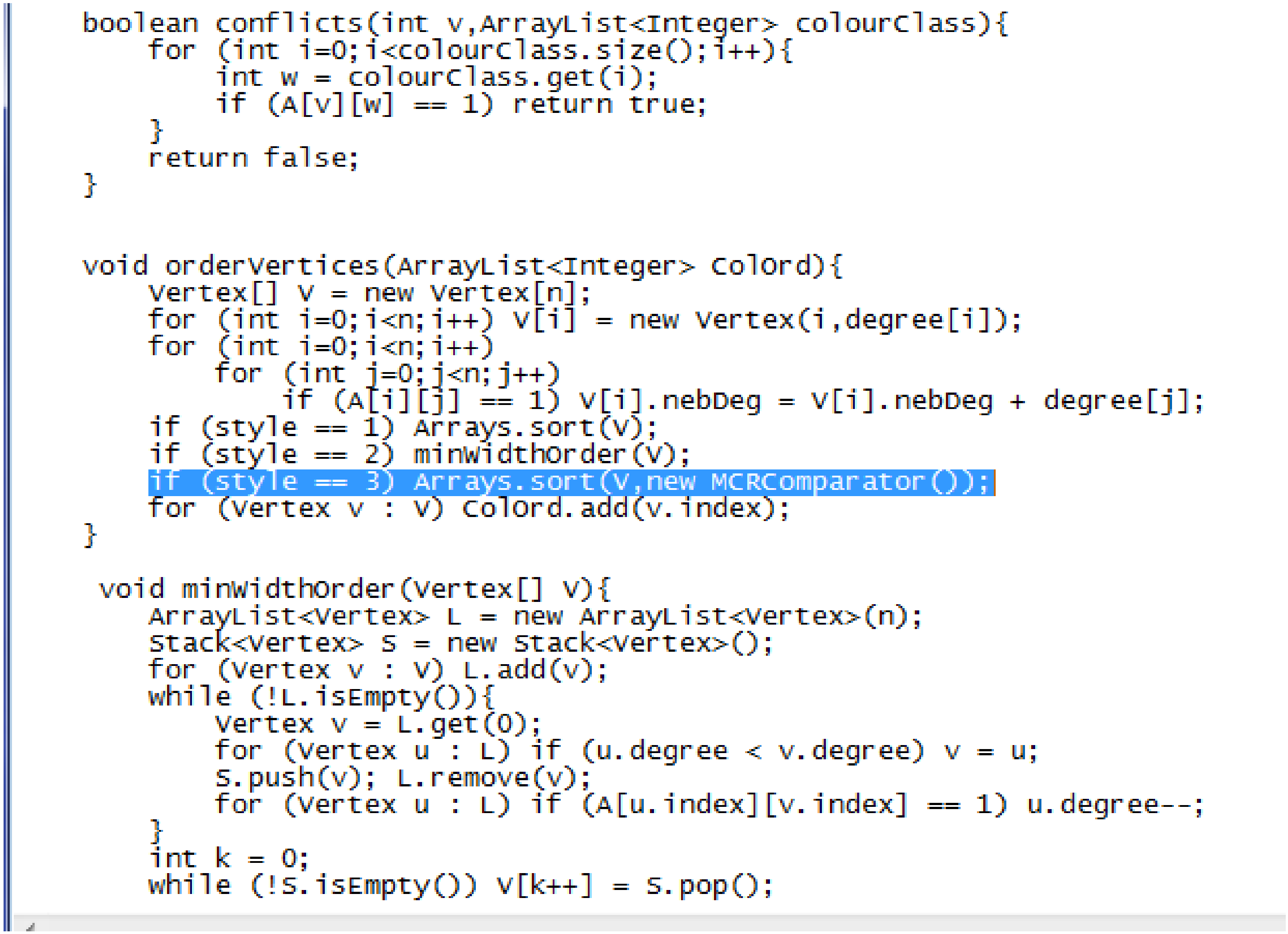


```
import java.util.*;

public class MCRComparator implements Comparator {

    public int compare(Object o1, Object o2){
        Vertex u = (Vertex) o1;
        Vertex v = (Vertex) o2;
        if (u.degree < v.degree ||
            u.degree == v.degree && u.nebDeg < v.nebDeg ||
            u.degree == v.degree && u.nebDeg == v.nebDeg && u.index > v.index) return 1;
        return -1;
    }
    //
    // to sort vertices by decreasing degree, tie breaking on neighbourhood degree (nebDeg)
    //
}
```

How can we compare objects?

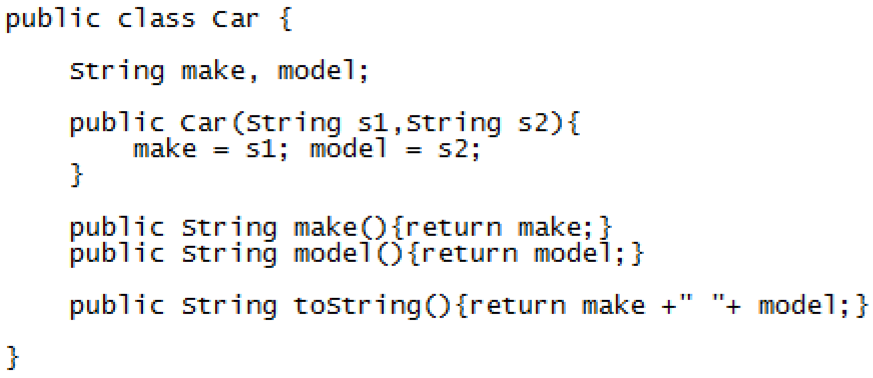
An example

Using the comparator to sort an array of Vertex



```
boolean conflicts(int v,ArrayList<Integer> colourClass){
    for (int i=0;i<colourClass.size();i++){
        int w = colourClass.get(i);
        if (A[v][w] == 1) return true;
    }
    return false;
}


void orderVertices(ArrayList<Integer> colOrd){
    Vertex[] V = new Vertex[n];
    for (int i=0;i<n;i++) V[i] = new Vertex(i,degree[i]);
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            if (A[i][j] == 1) V[i].nebDeg = V[i].nebDeg + degree[j];
    if (style == 1) Arrays.sort(V);
    if (style == 2) minWidthOrder(V);
    if (style == 3) Arrays.sort(V,new MCRComparator());
    for (Vertex v : V) colOrd.add(v.index);
}

 void minWidthOrder(Vertex[] V){
    ArrayList<Vertex> L = new ArrayList<Vertex>(n);
    Stack<Vertex> S = new Stack<Vertex>();
    for (Vertex v : V) L.add(v);
    while (!L.isEmpty()){
        Vertex v = L.get(0);
        for (Vertex u : L) if (u.degree < v.degree) v = u;
        S.push(v); L.remove(v);
        for (Vertex u : L) if (A[u.index][v.index] == 1) u.degree--;
    }
    int k = 0;
    while (!S.isEmpty()) V[k++] = S.pop();
```

Another example: a Car

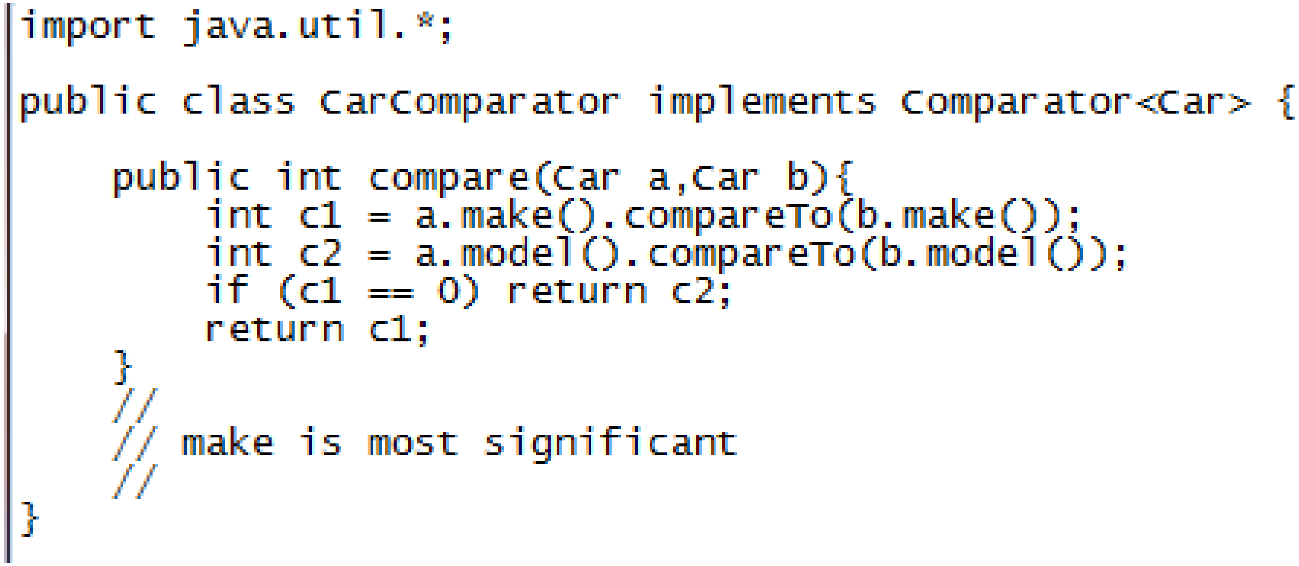


```
public class Car {

    String make, model;

    public Car(String s1,String s2){
        make = s1; model = s2;
    }

    public String make(){return make;}
    public String model(){return model;}

    public String toString(){return make +" "+ model;}

}
```

How can we compare objects?

An example

This is a CarComparator

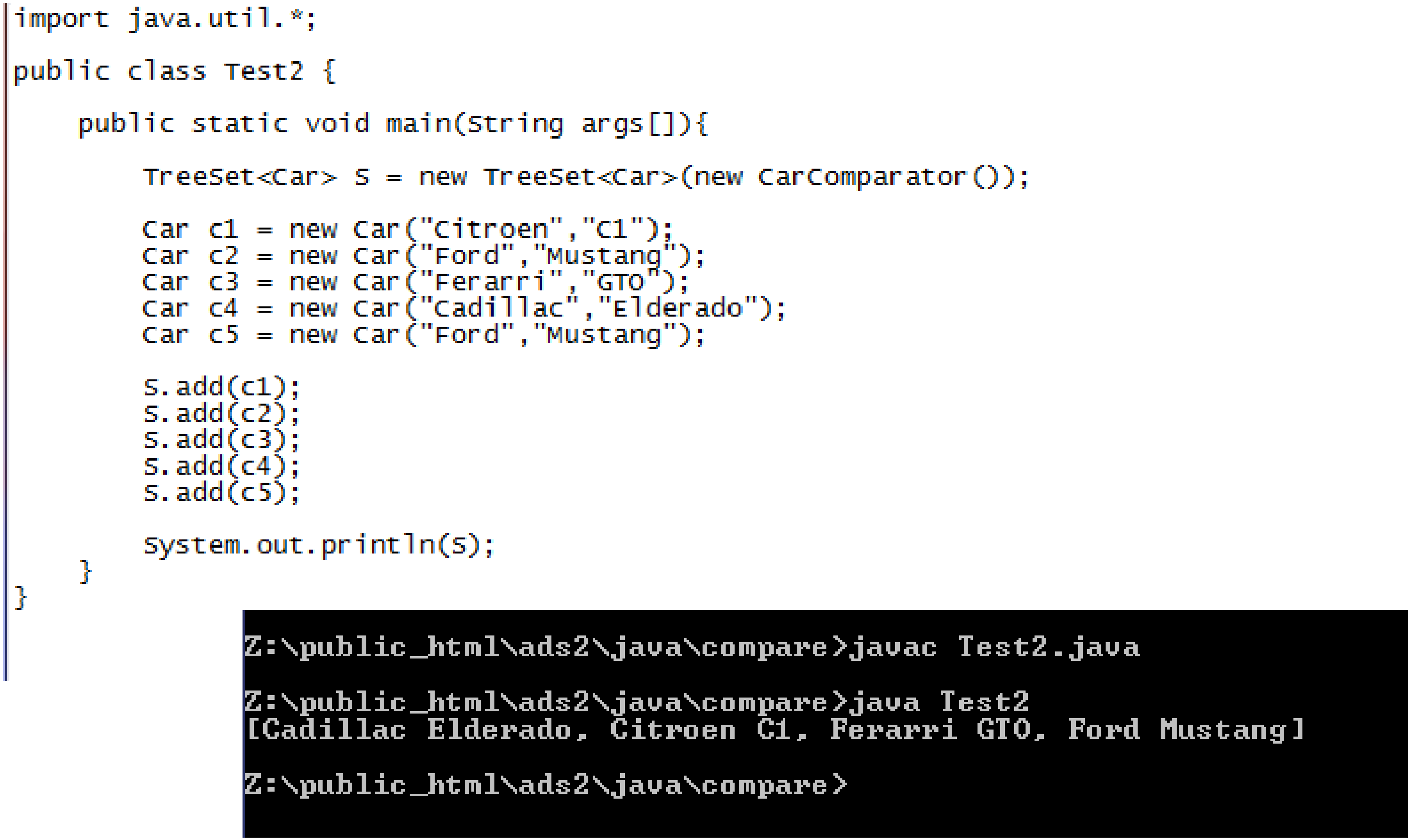


```
import java.util.*;

public class CarComparator implements Comparator<Car> {

    public int compare(Car a,Car b){
        int c1 = a.make().compareTo(b.make());
        int c2 = a.model().compareTo(b.model());
        if (c1 == 0) return c2;
        return c1;
    }
    //
    // make is most significant
    //
}
```

How can we compare objects?

An example

Using the CarComparator



```
import java.util.*;

public class Test2 {

    public static void main(String args[]){

        TreeSet<Car> S = new TreeSet<Car>(new CarComparator());

        Car c1 = new Car("Citroen","C1");
        Car c2 = new Car("Ford","Mustang");
        Car c3 = new Car("Ferarri","GTO");
        Car c4 = new Car("Cadillac","Elderado");
        Car c5 = new Car("Ford","Mustang");

        S.add(c1);
        S.add(c2);
        S.add(c3);
        S.add(c4);
        S.add(c5);

        System.out.println(S);
    }
}
```

```
Z:\public_html\ads2\java\compare>javac Test2.java

Z:\public_html\ads2\java\compare>java Test2
[Cadillac Elderado, Citroen C1, Ferarri GTO, Ford Mustang]

Z:\public_html\ads2\java\compare>
```

Implementing a priority queue with an **unsorted** list

We might use a linked list

- To insert we add to the front of the list
- To find the minimum we must iterate over entire the list
- To remove the minimum we must find the minimum and remove it
- Maintain a counter of number of elements in the list

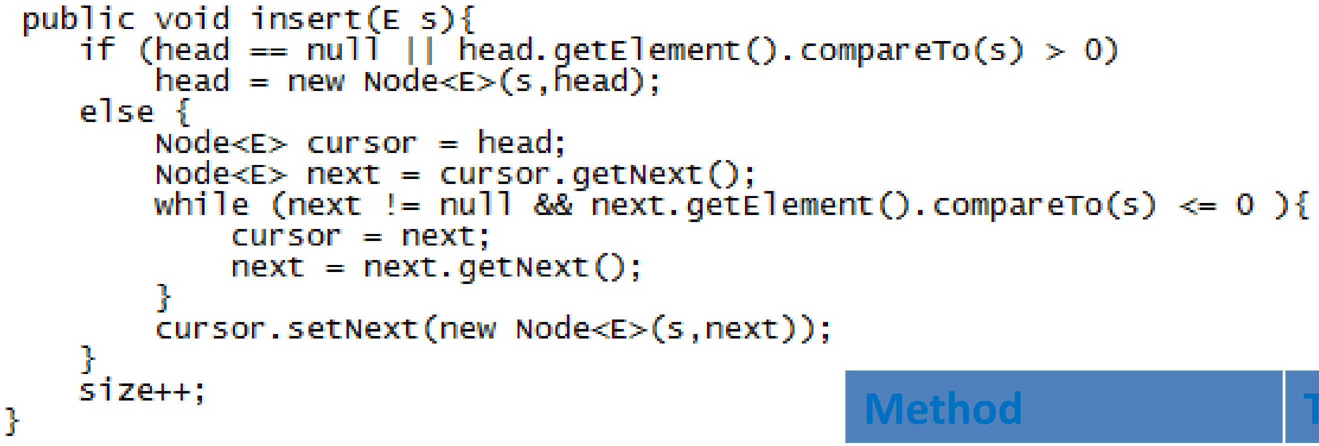
Method	Time
size	O(1)
isEmpty	O(1)
insert	O(1)
removeMin	O(n)
min	O(n)

Implementing a priority queue with an **sorted** list

sorted list

We might use a linked list

- The list is **maintained in non-decreasing order**
- To insert we scan to find position and splice in (see below)
- To find the minimum we deliver the first element in the list
- To remove the minimum we return and remove the first element



```
public void insert(E s){
    if (head == null || head.getElement().compareTo(s) > 0)
        head = new Node<E>(s,head);
    else {
        Node<E> cursor = head;
        Node<E> next = cursor.getNext();
        while (next != null && next.getElement().compareTo(s) <= 0 ){
            cursor = next;
            next = next.getNext();
        }
        cursor.setNext(new Node<E>(s,next));
    }
    size++;
}
```

Method	Time
size	O(1)
isEmpty	O(1)
insert	O(n)
removeMin	O(1)
min	O(1)

An alternative

THE HEAP

heap

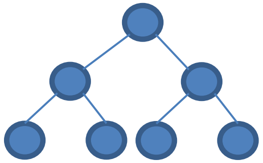
- a heap H is a binary tree

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree

heap

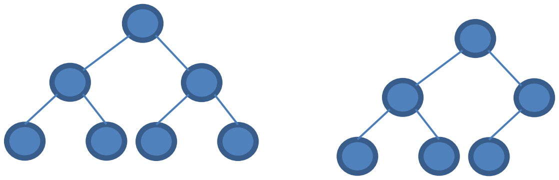
- a heap H is a binary tree
- H is a ***complete*** binary tree



Fill up level d before moving to level d+1
- each level but the last must be full
- in last level fill from left to right

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree

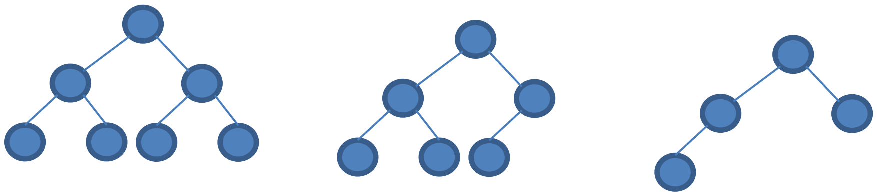


Fill up level d before moving to level d+1

- each level but the last must be full
- in last level fill from left to right

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree

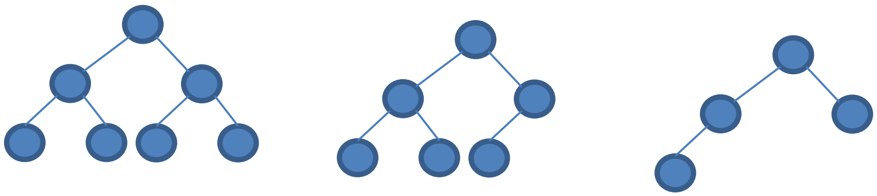


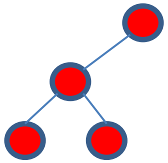
Fill up level d before moving to level d+1

- each level but the last must be full
- in last level fill from left to right

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree





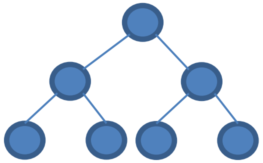
Not a heap!

Fill up level d before moving to level d+1

- each level but the last must be full
- in last level fill from left to right

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree
- ***heap order property*** is maintained



heap

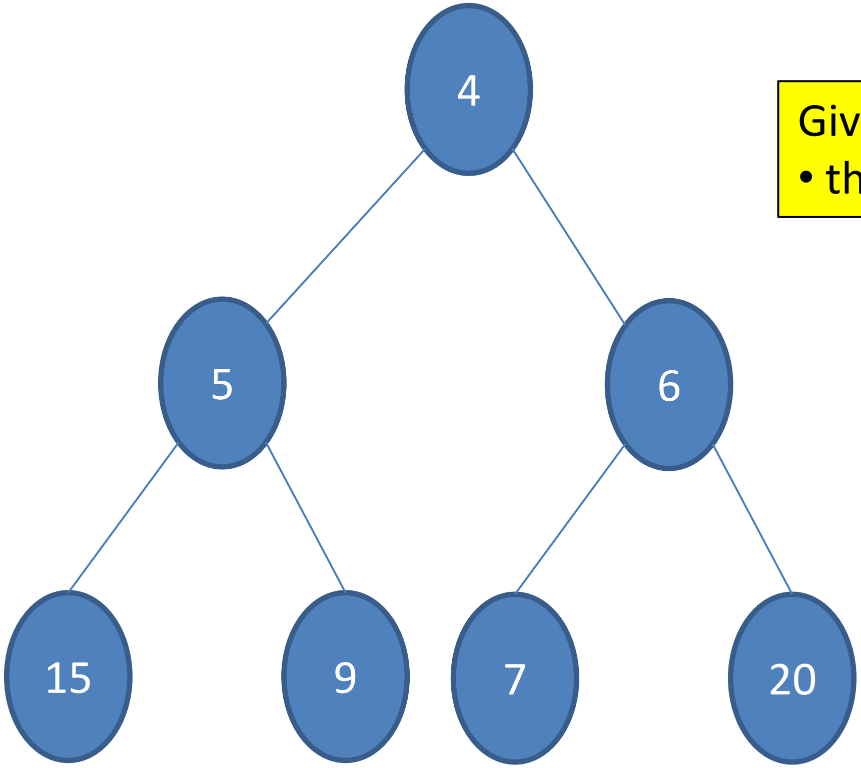
- a heap H is a binary tree
- H is a ***complete*** binary tree
- ***heap order property*** is maintained

Given a node v (not the root)

- the parent of v is less than or equal to v

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree
- ***heap order property*** is maintained

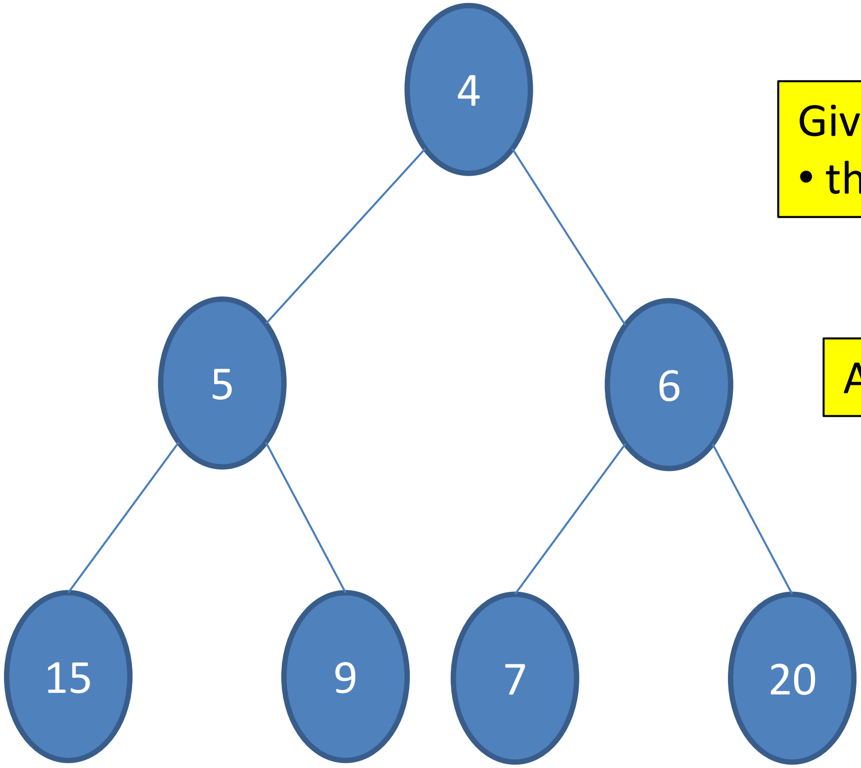


Given a node v (not the root)

- the parent of v is less than or equal to v

heap

- a heap H is a binary tree
- H is a ***complete*** binary tree
- ***heap order property*** is maintained



Given a node v (not the root)

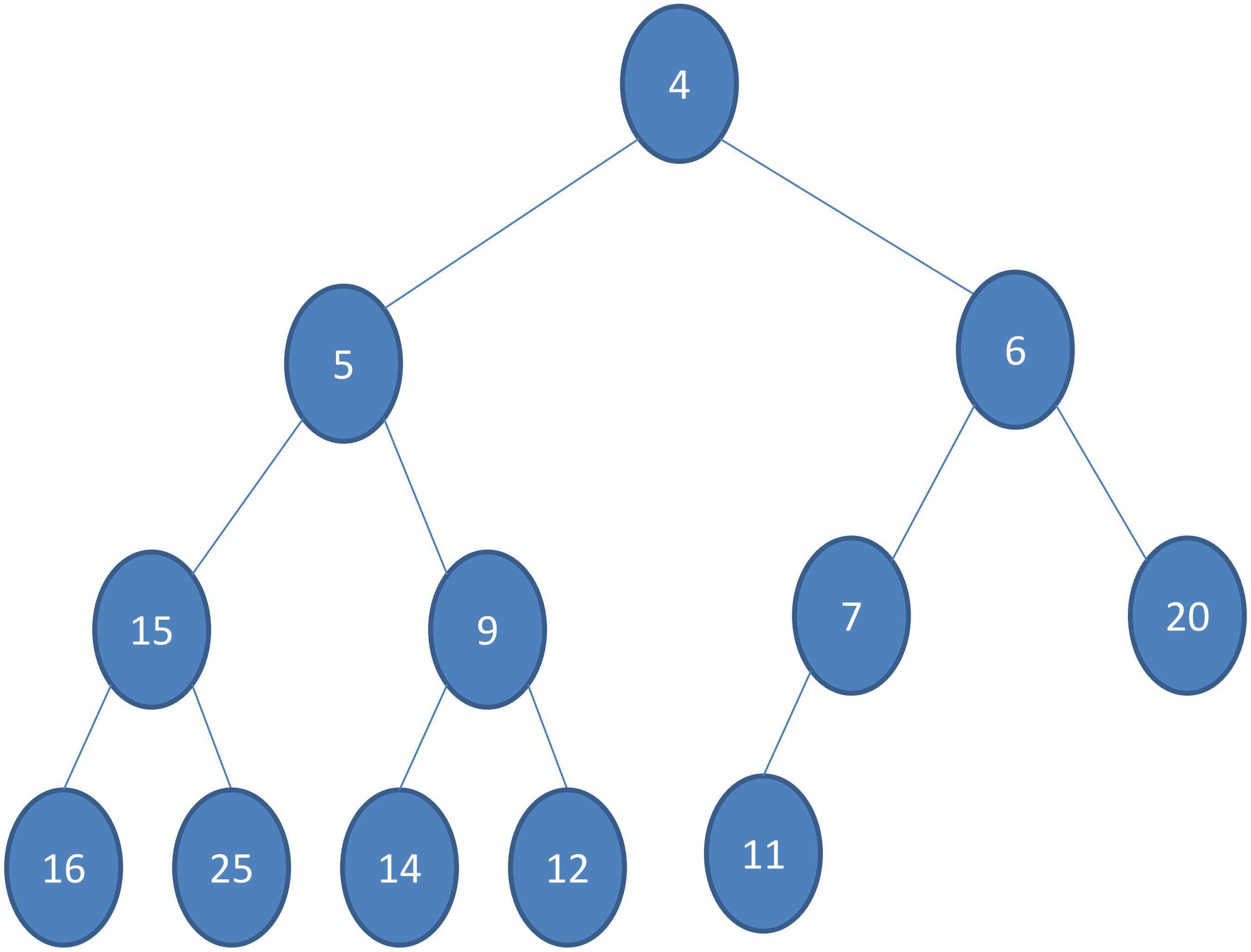
- the parent of v is less than or equal to v

A heap H with n nodes has height O(log(n))

Example: adding to a heap

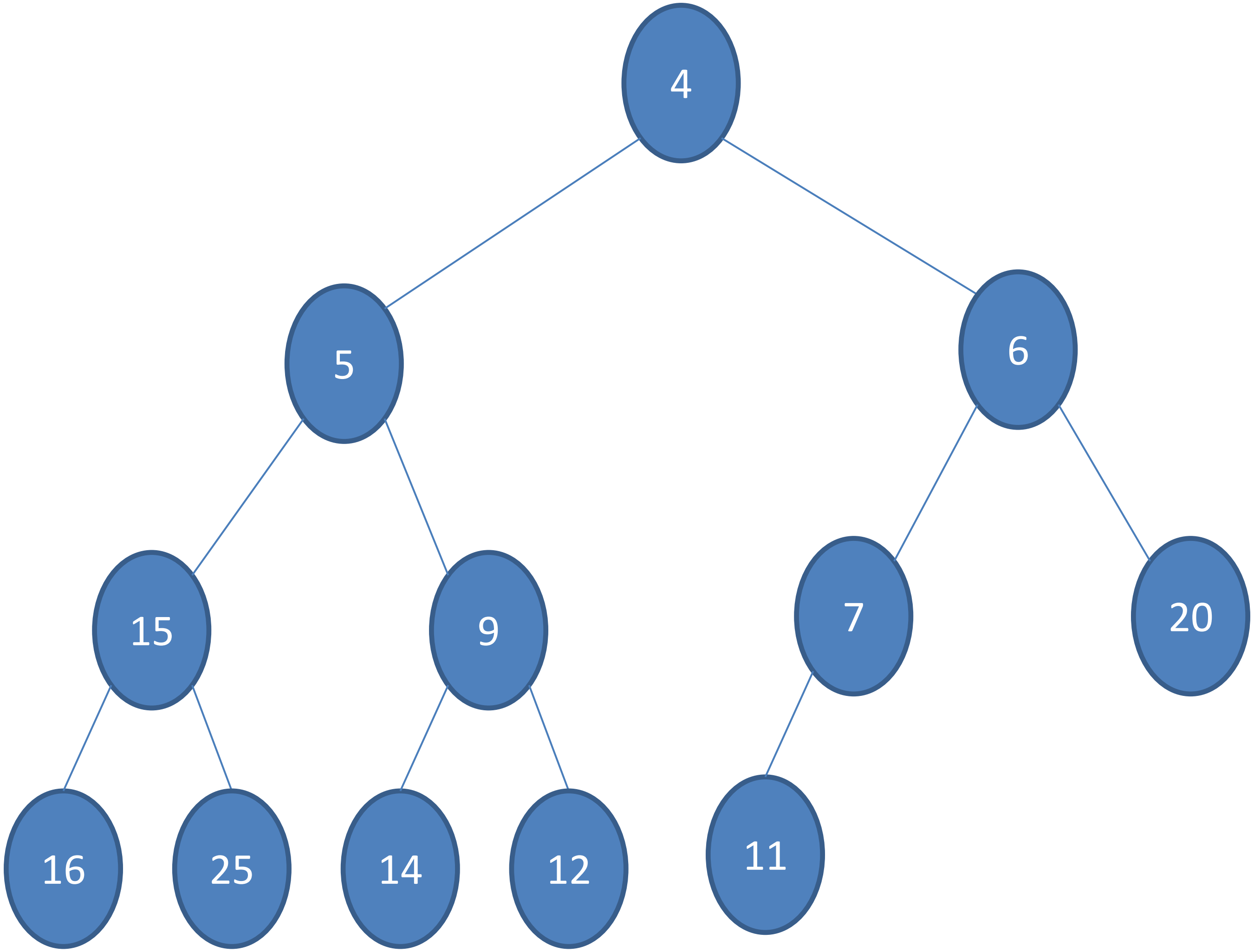
Example: adding to a heap

heap



Example: adding to a heap

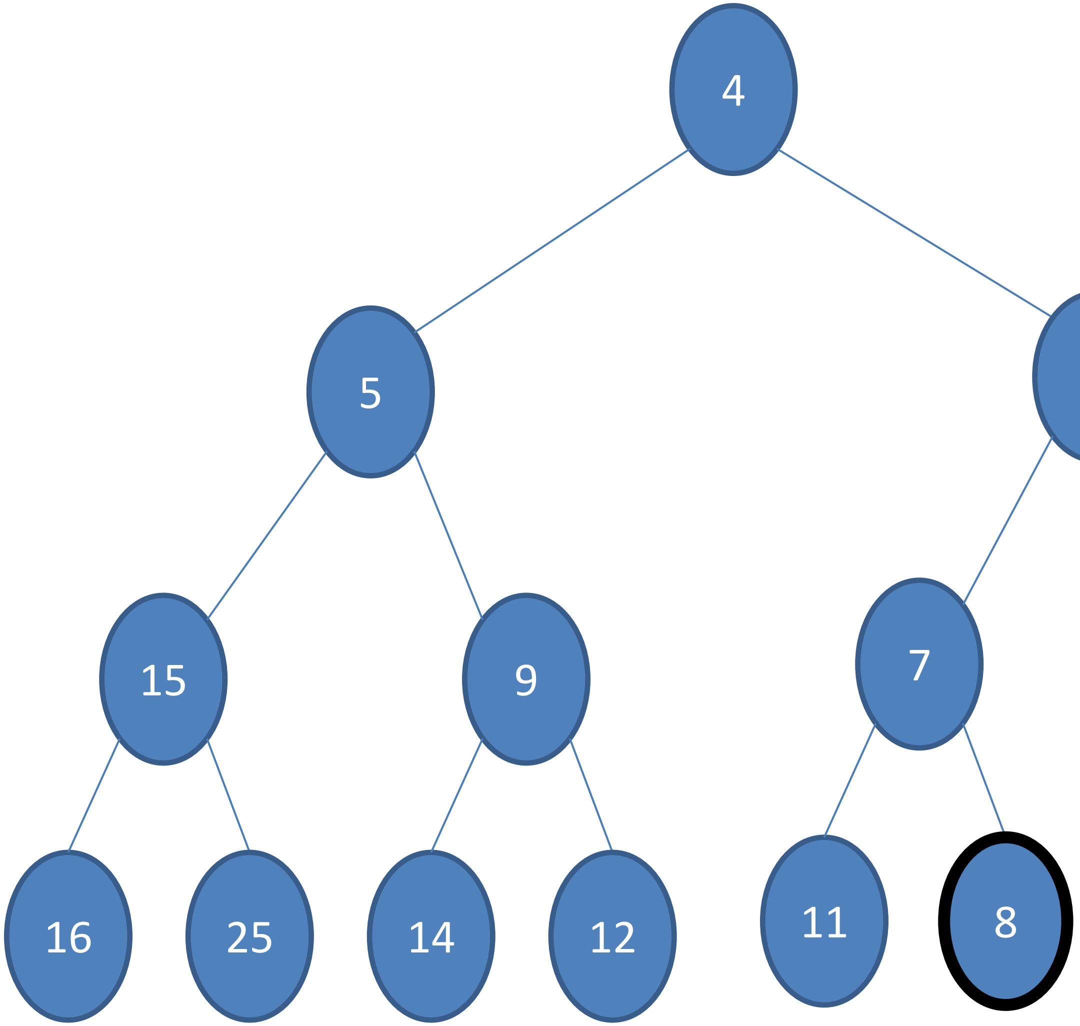
heap



Insert 8

Example: adding to a heap

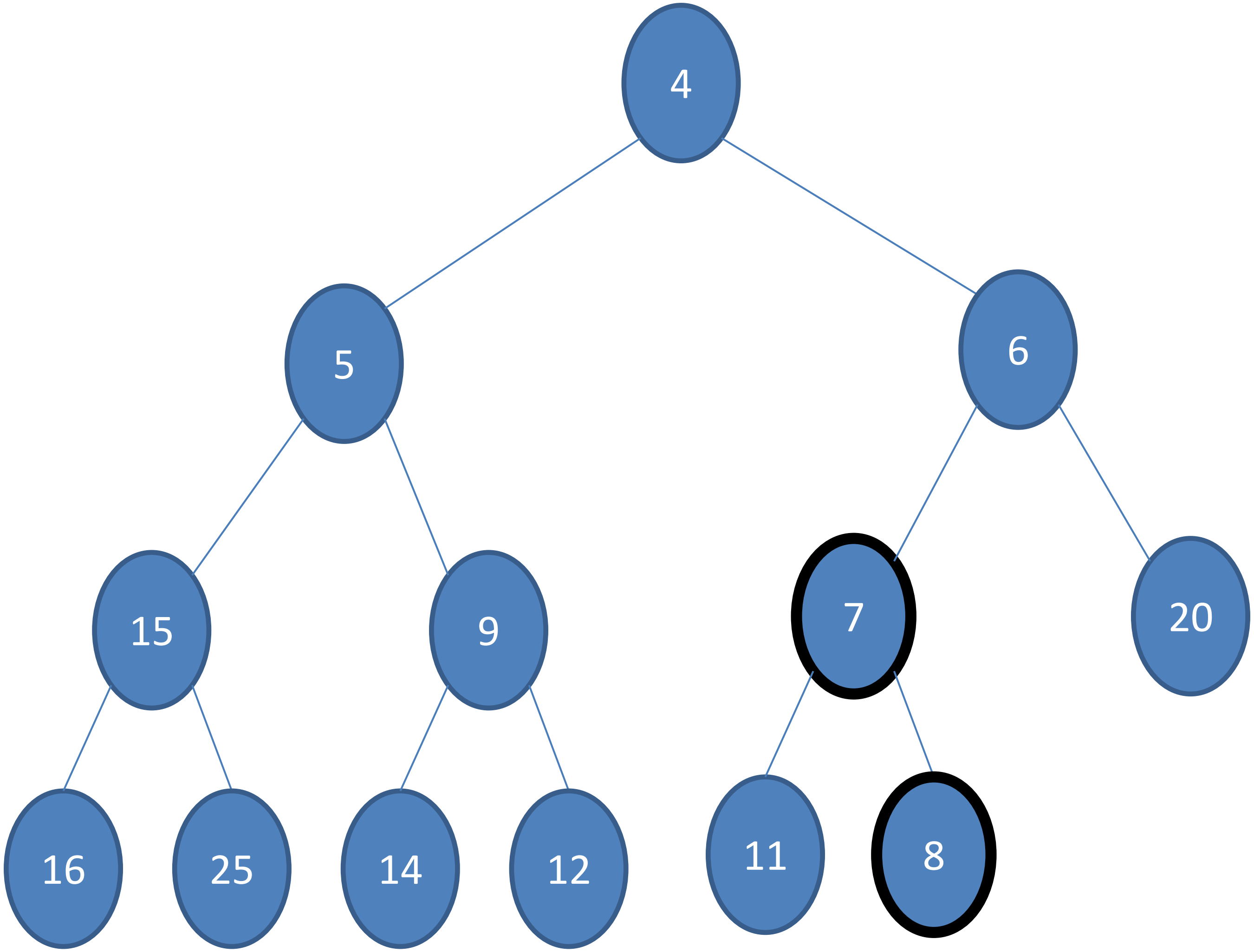
heap



Insert 8

Example: adding to a heap

heap

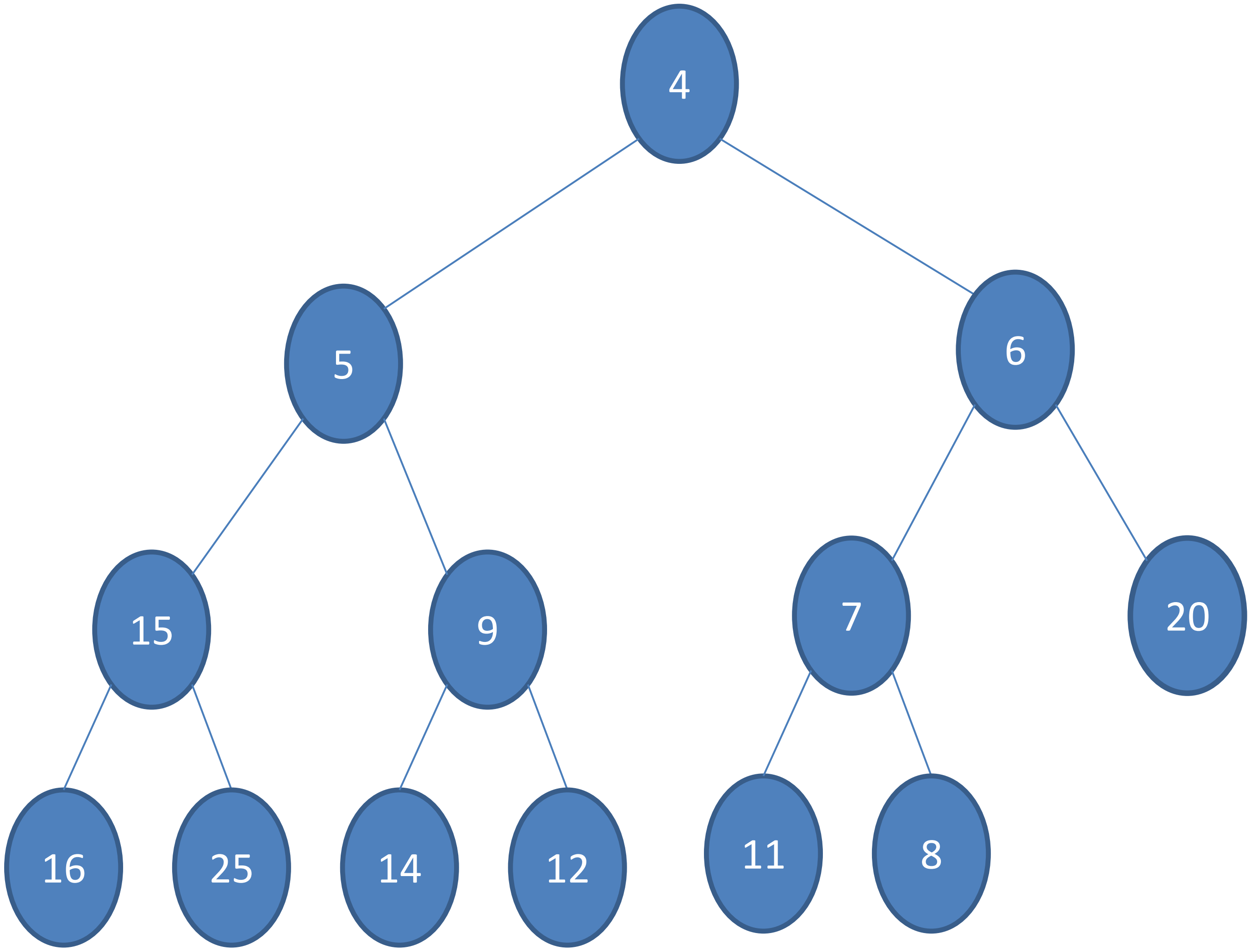


Insert 8

8 is greater than parent (7) ... done

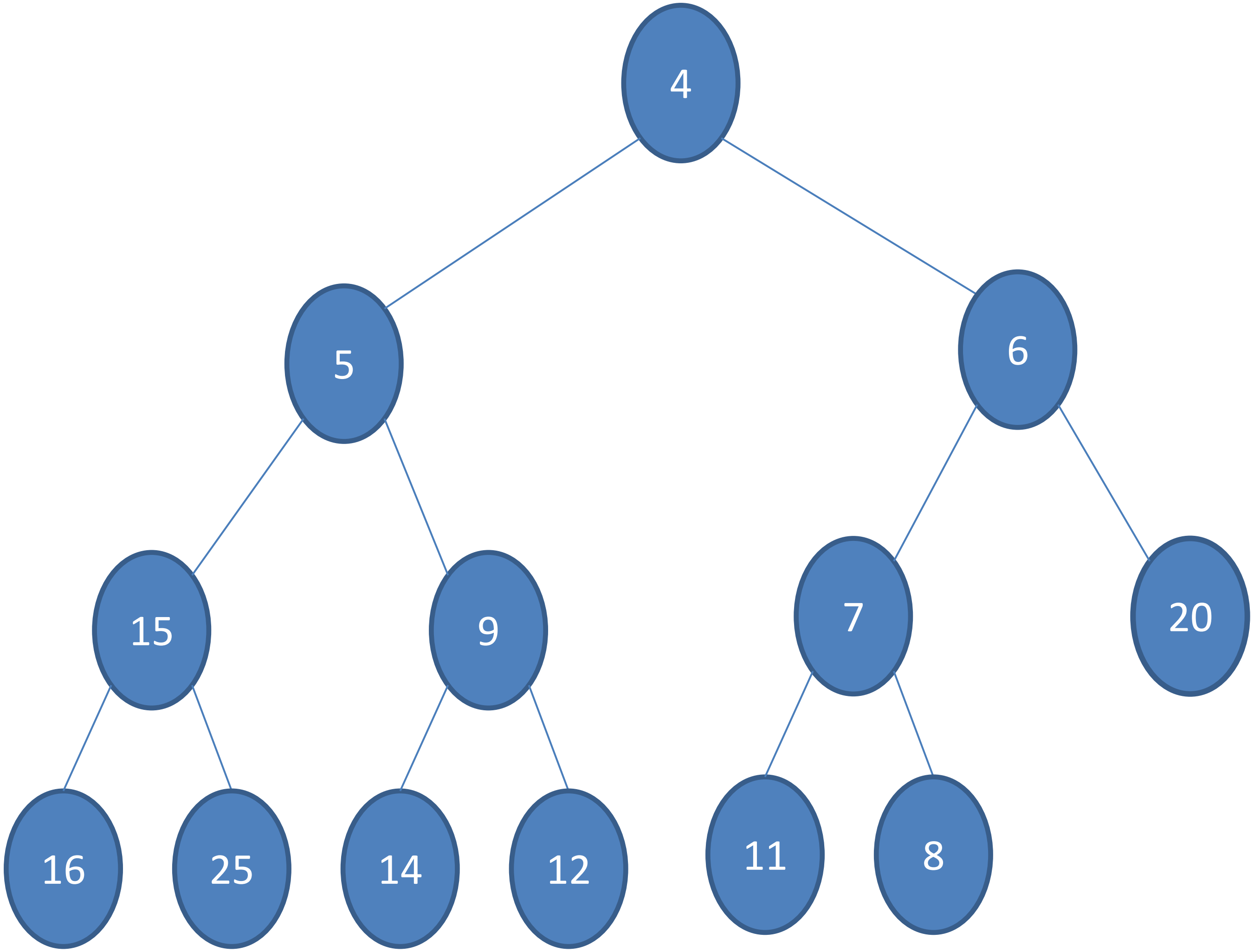
Example: adding to a heap

heap



Example: adding to a heap

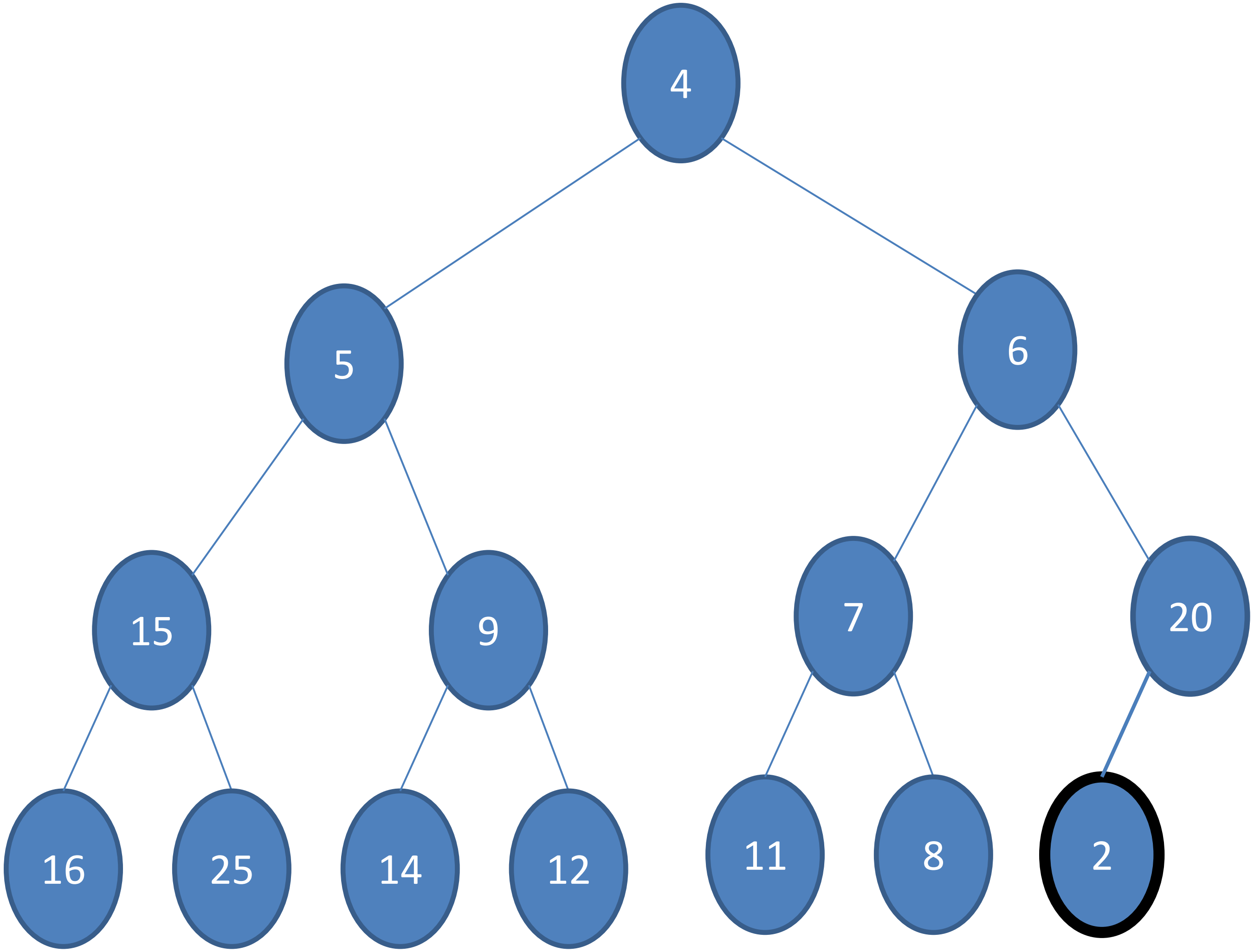
heap



Insert 2

Example: adding to a heap

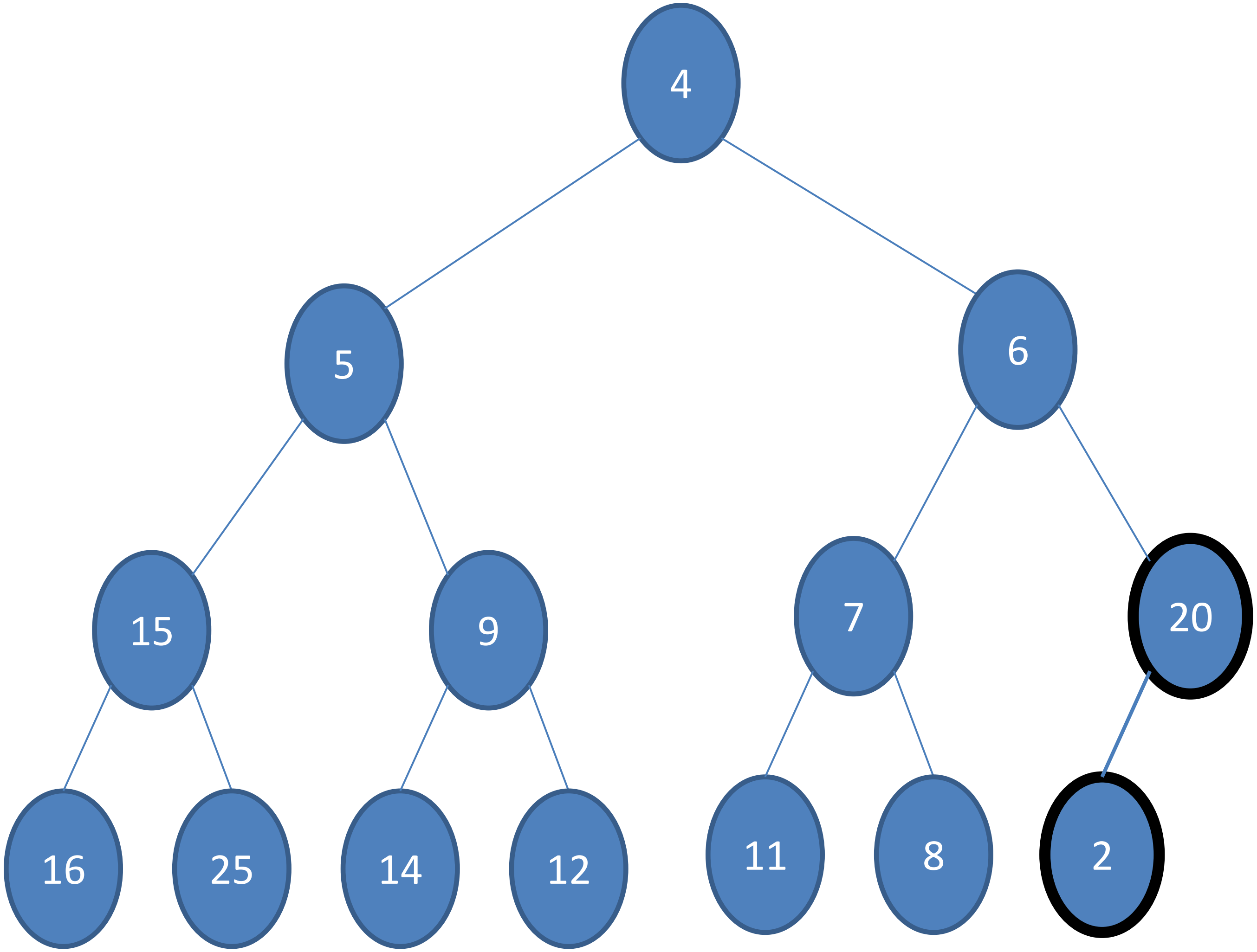
heap



Insert 2

Example: adding to a heap

heap

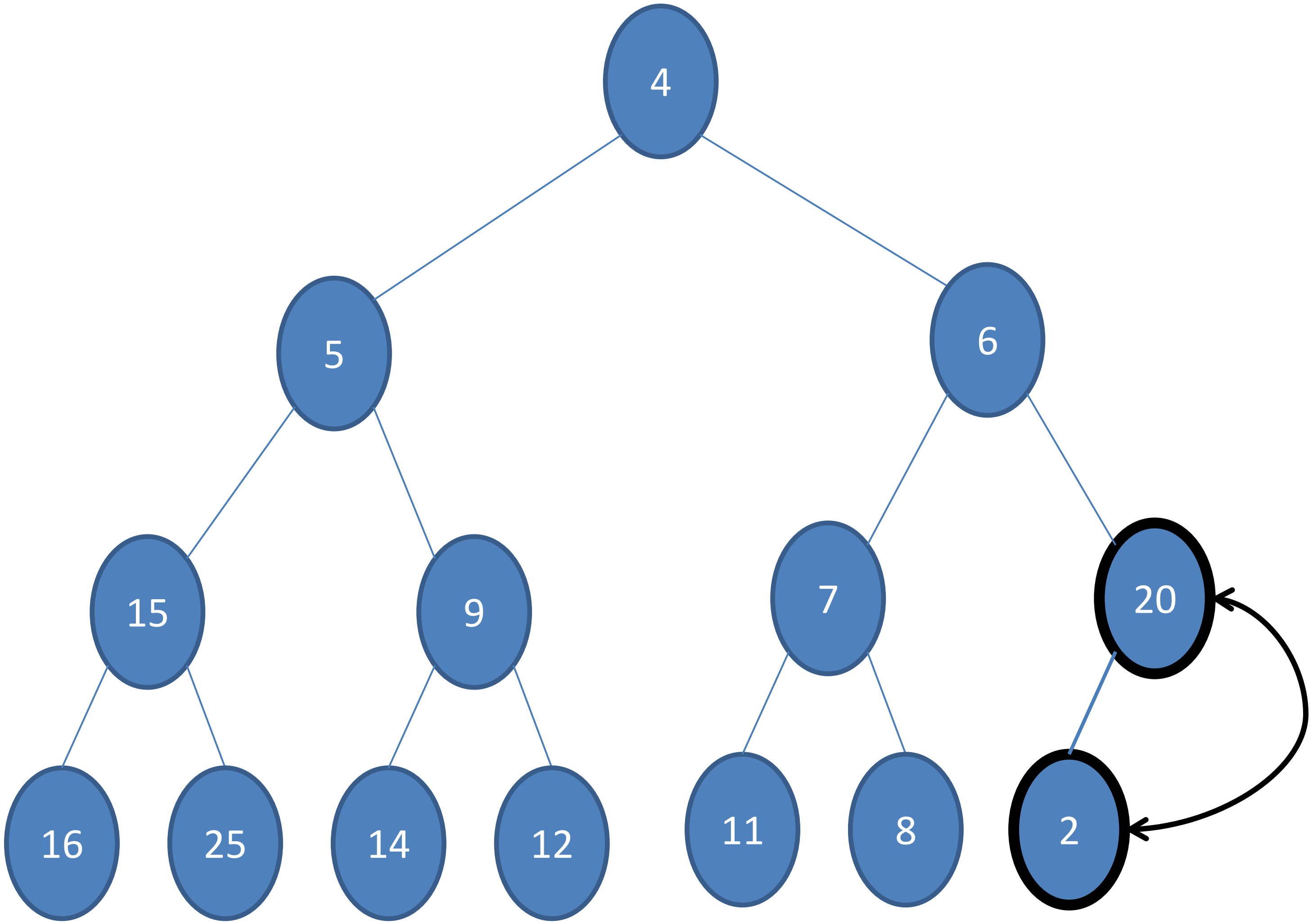


Insert 2

2 is less than parent 20

Example: adding to a heap

heap

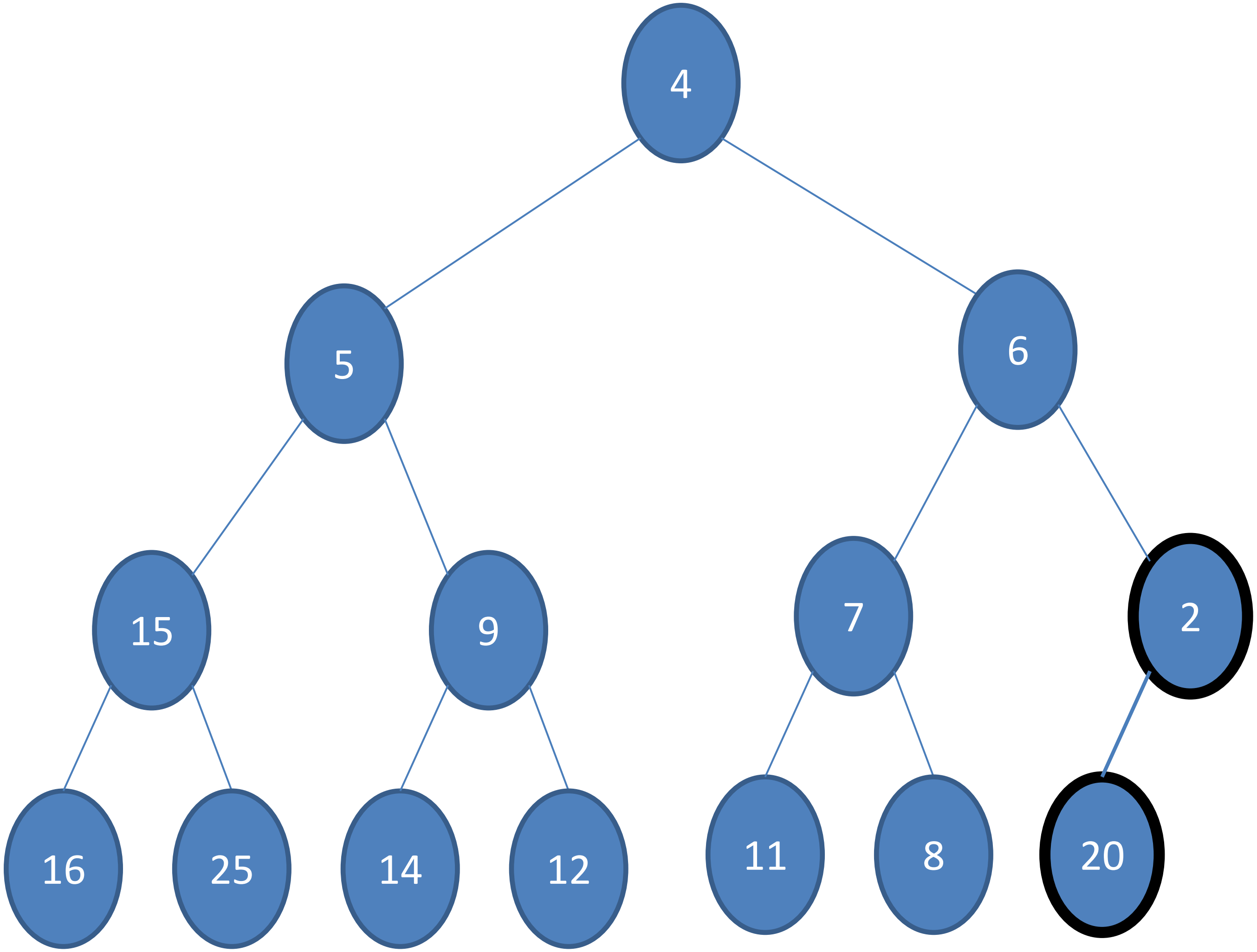


Insert 2

2 is less than parent 20 ... swap!

Example: adding to a heap

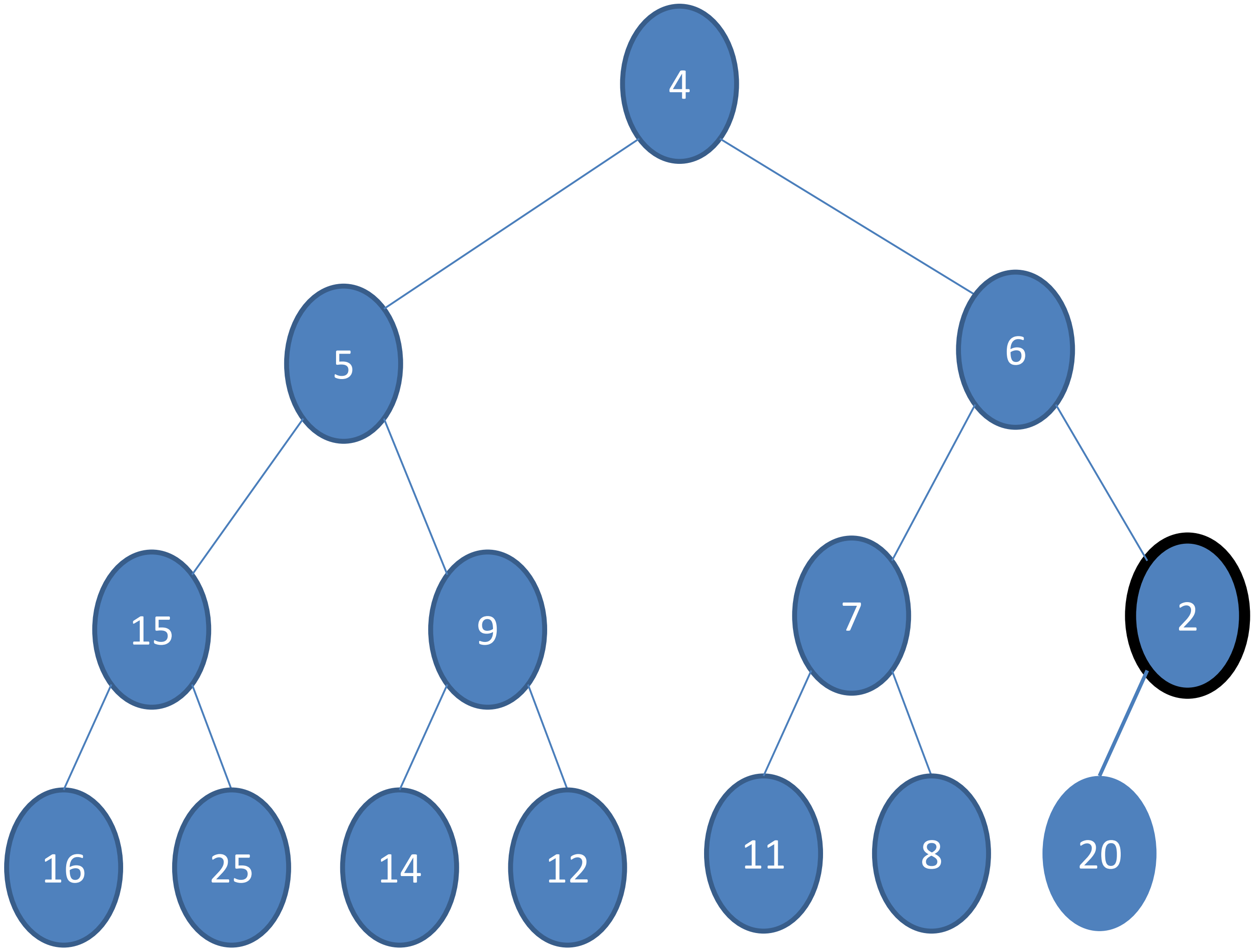
heap



Insert 2

Example: adding to a heap

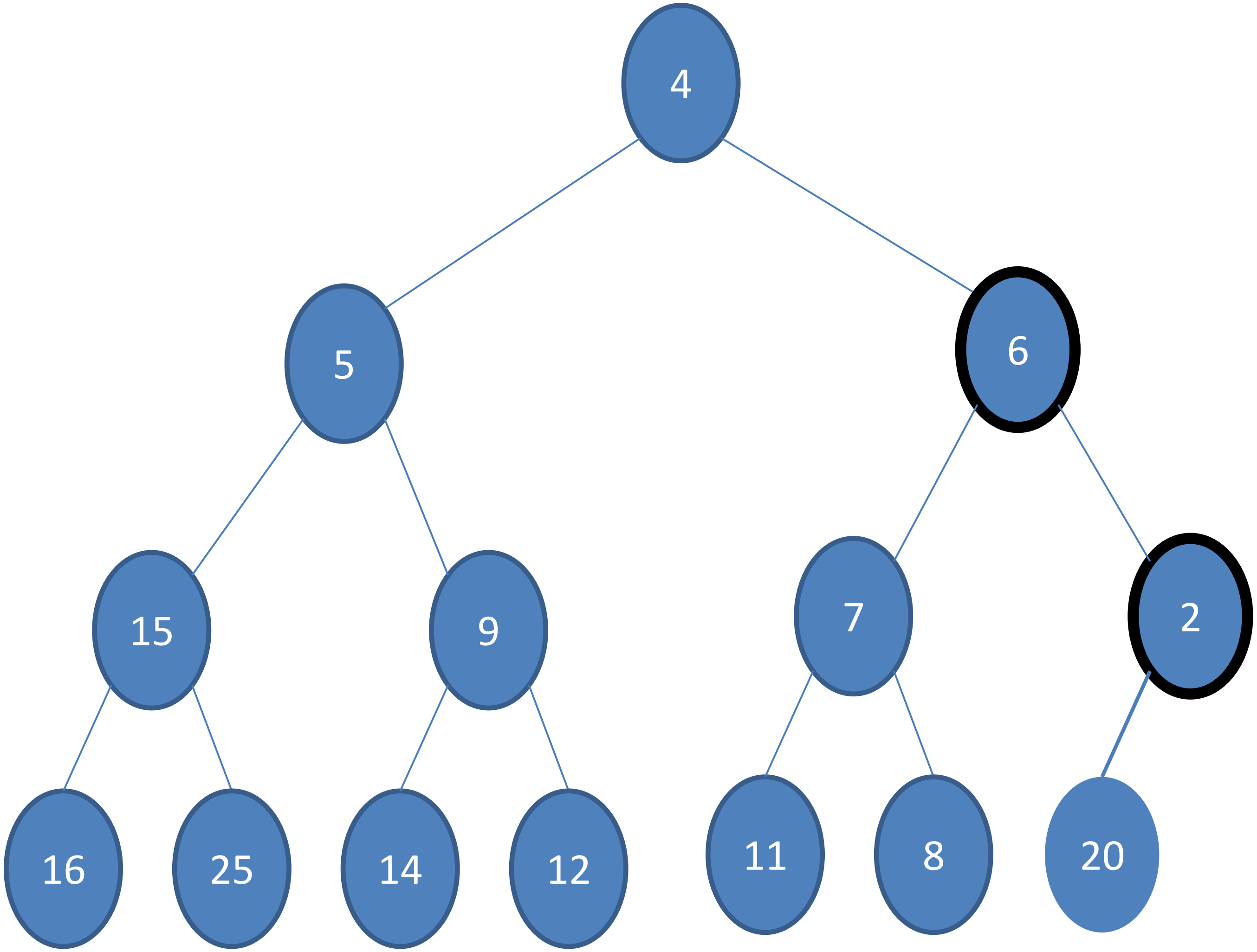
heap



Insert 2

Example: adding to a heap

heap

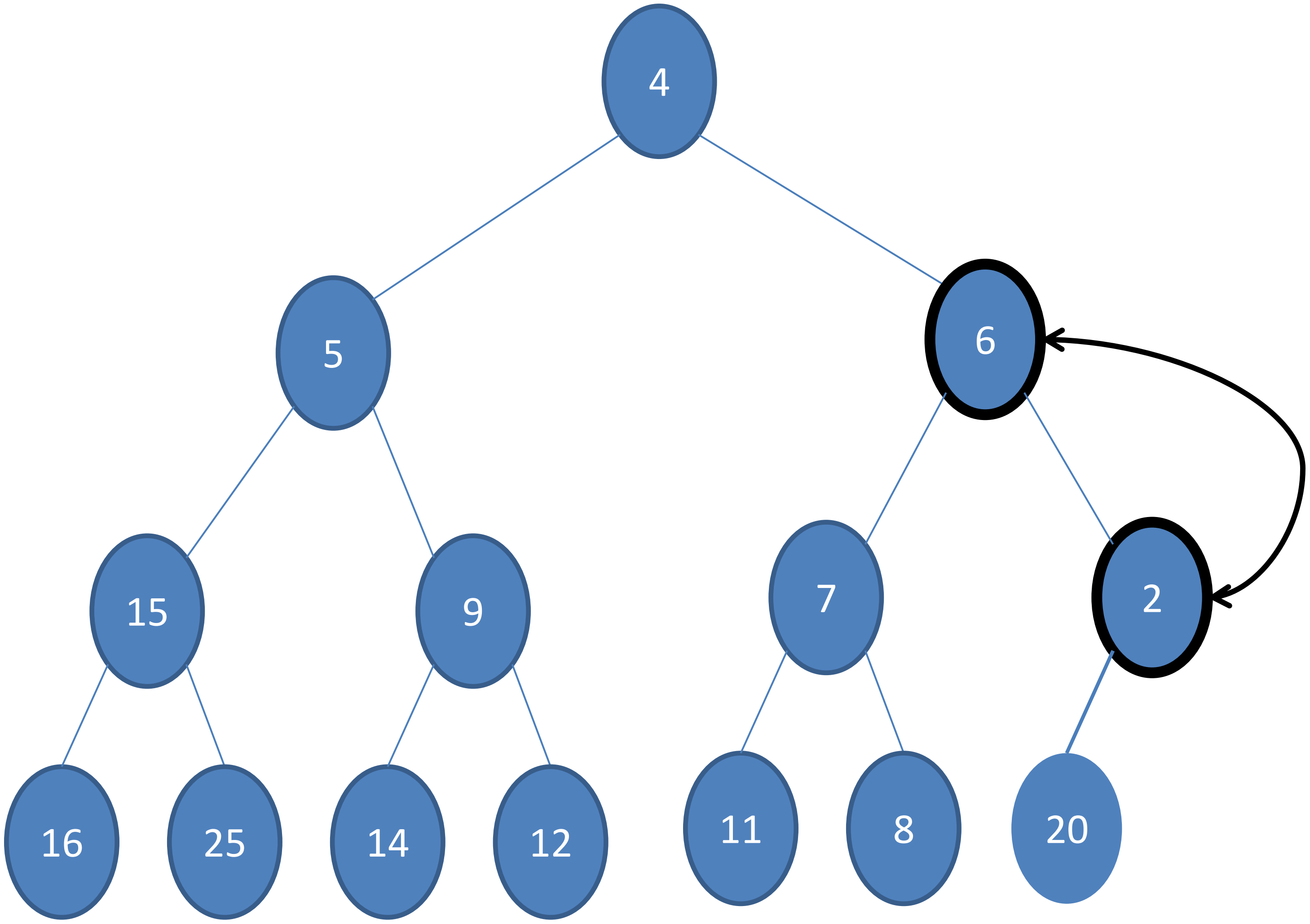


Insert 2

2 is less than parent 6

Example: adding to a heap

heap

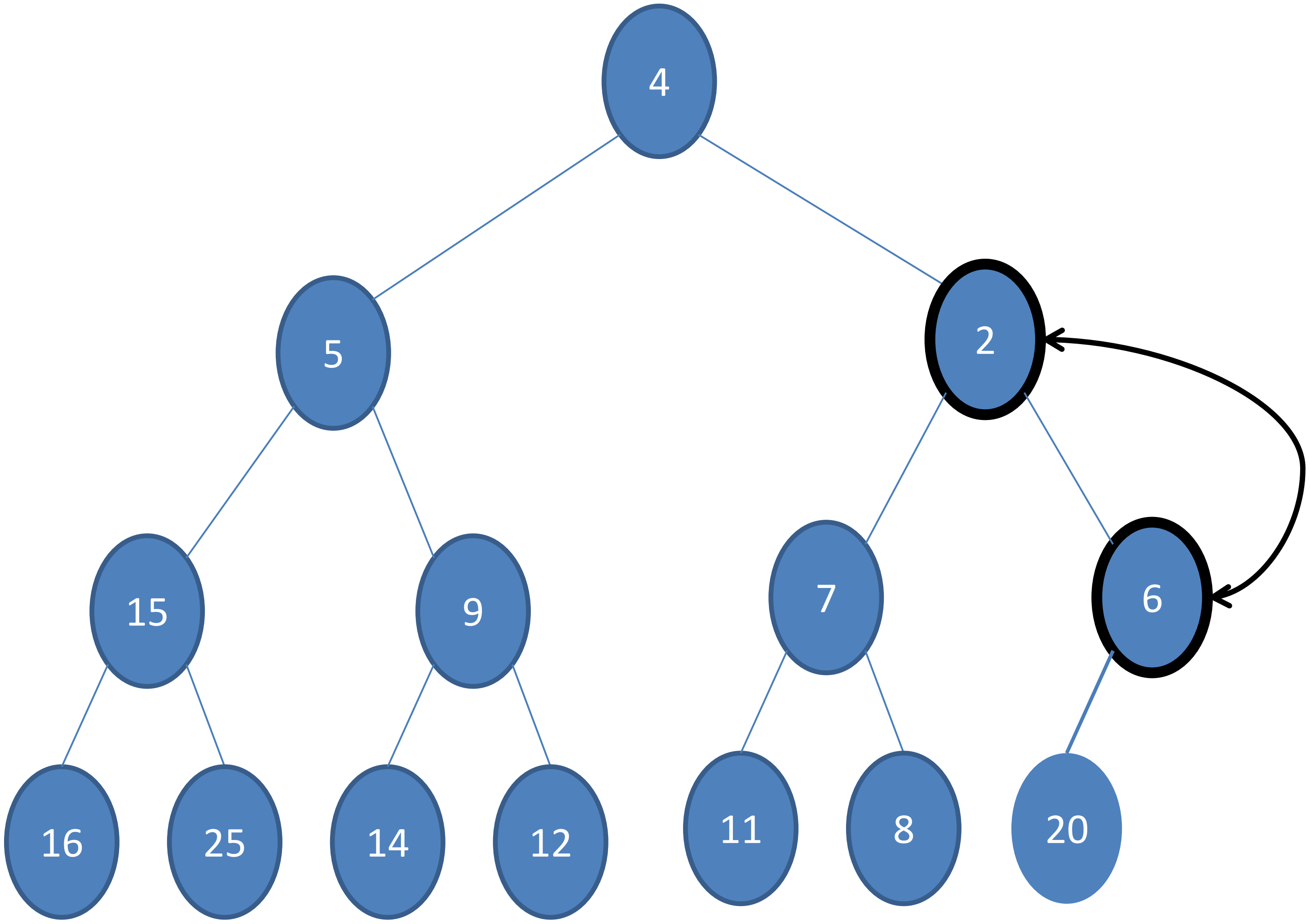


Insert 2

2 is less than parent 6 ... swap!

Example: adding to a heap

heap

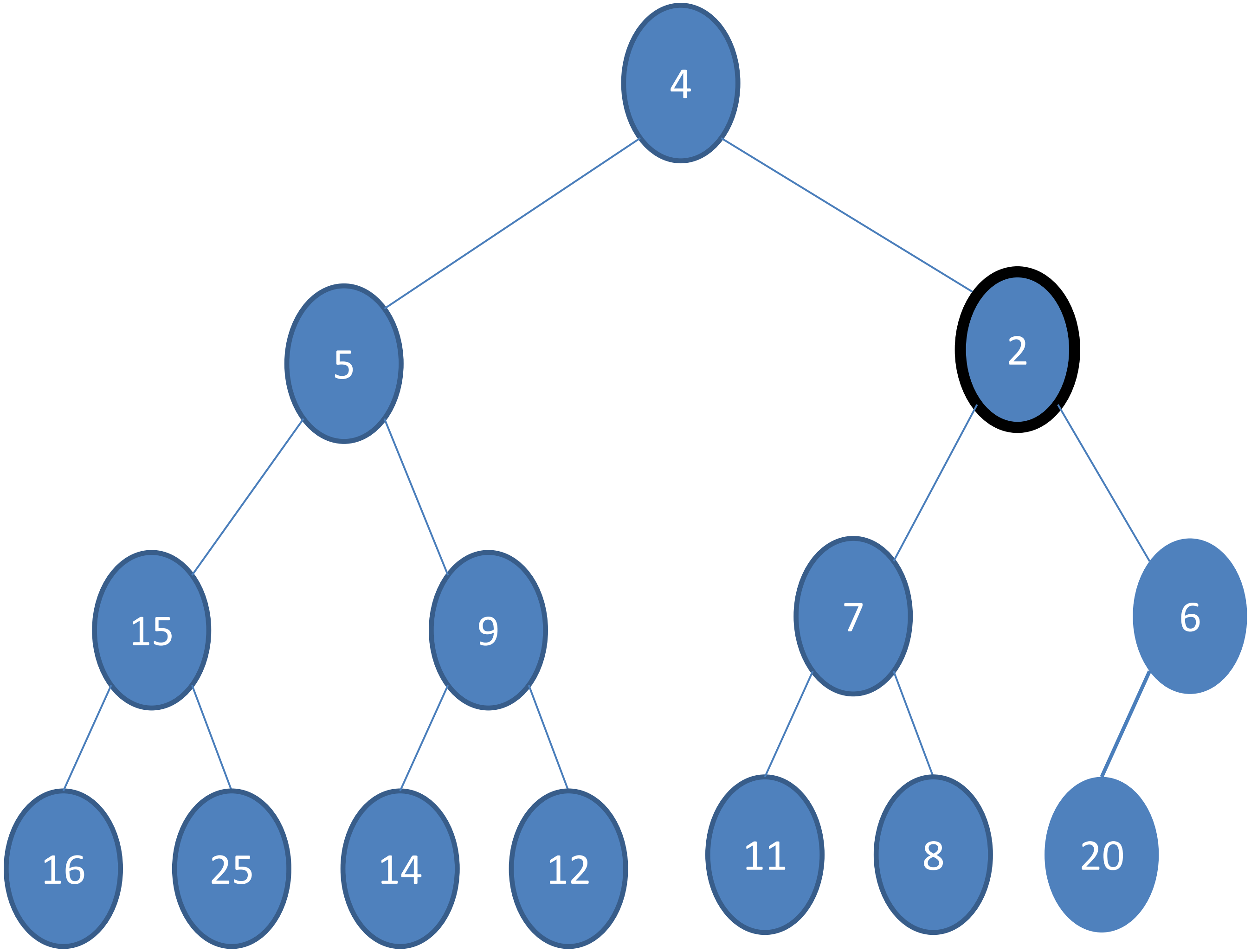


Insert 2

2 is less than parent 6 ... swap!

Example: adding to a heap

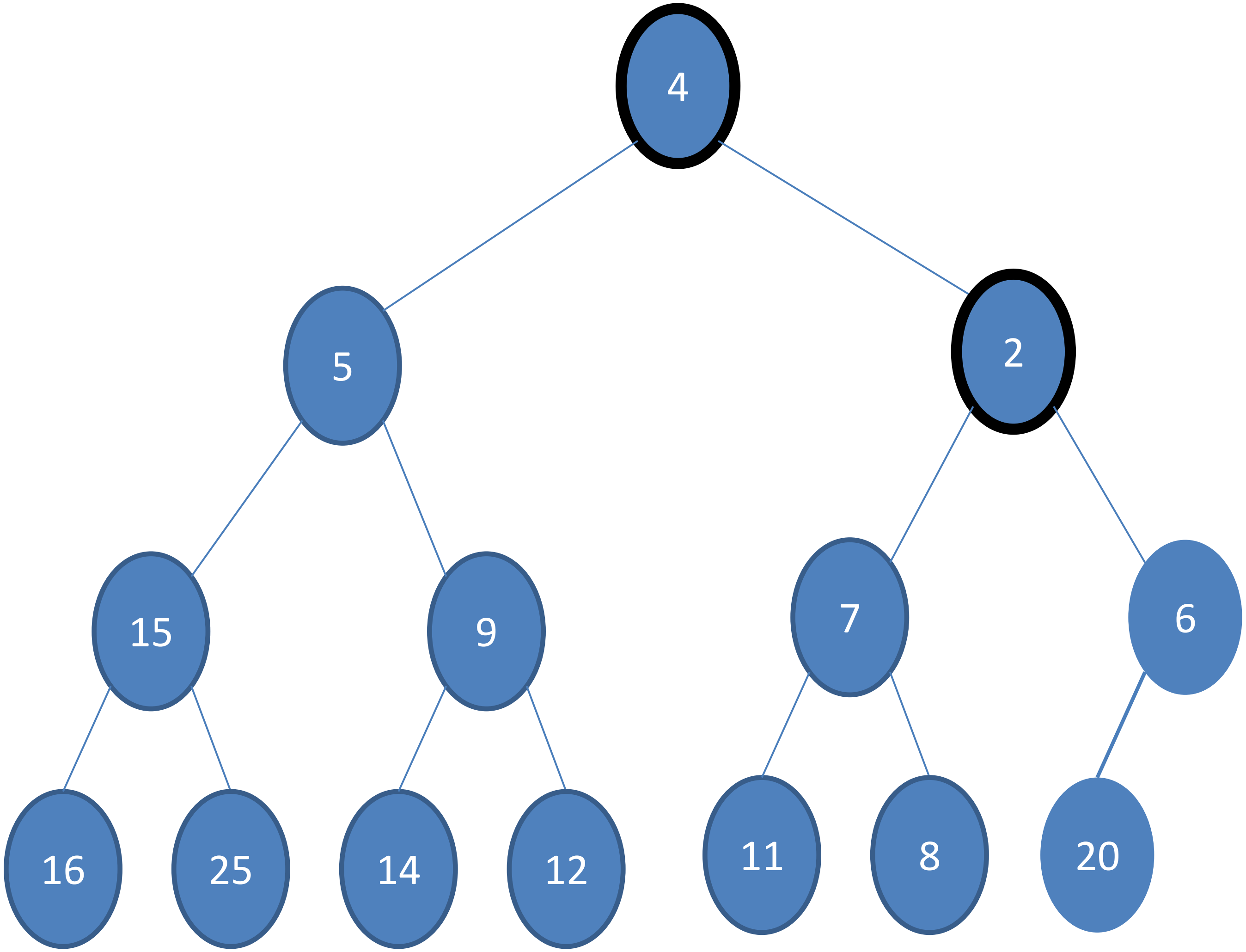
heap



Insert 2

Example: adding to a heap

heap

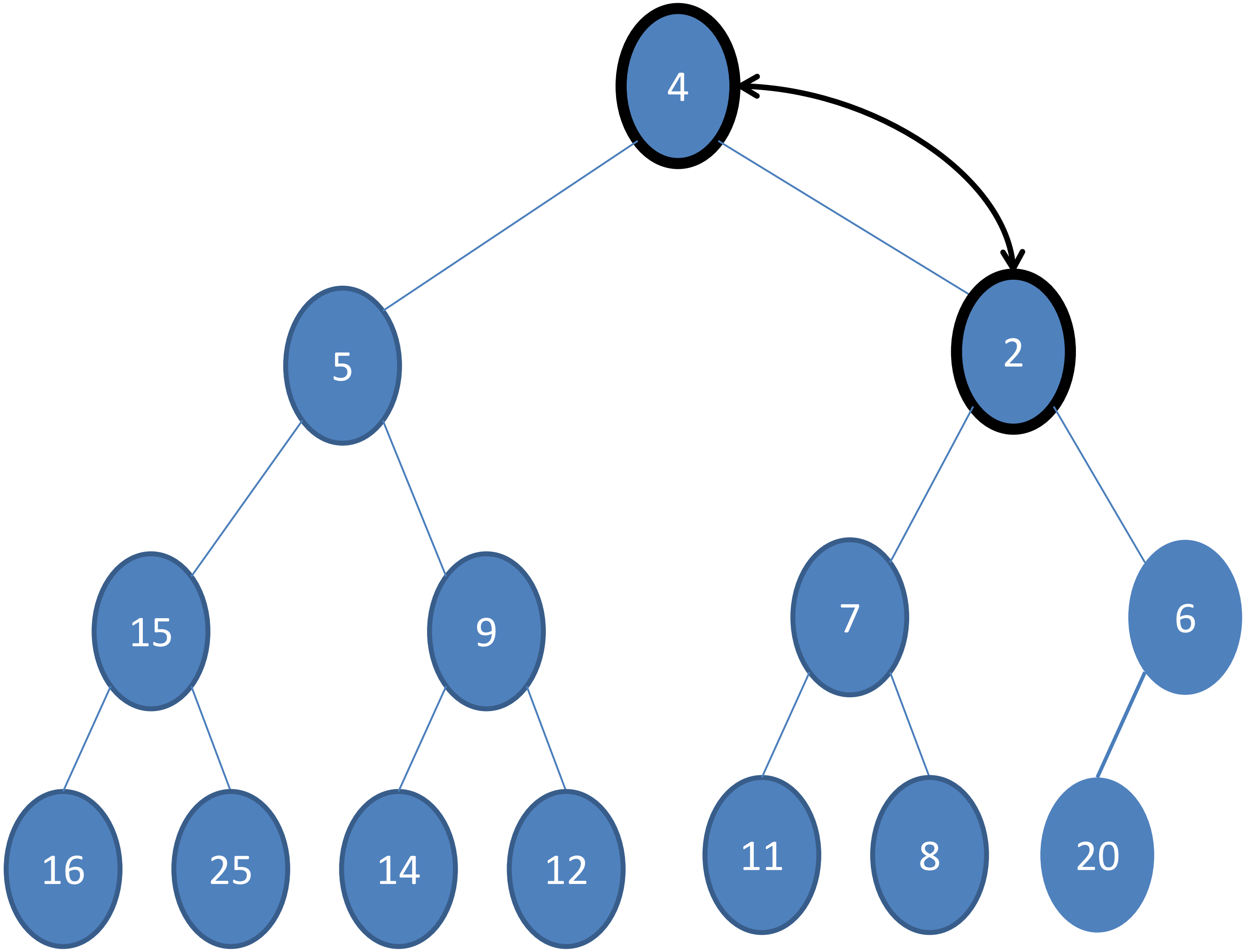


Insert 2

2 is less than parent 4

Example: adding to a heap

heap

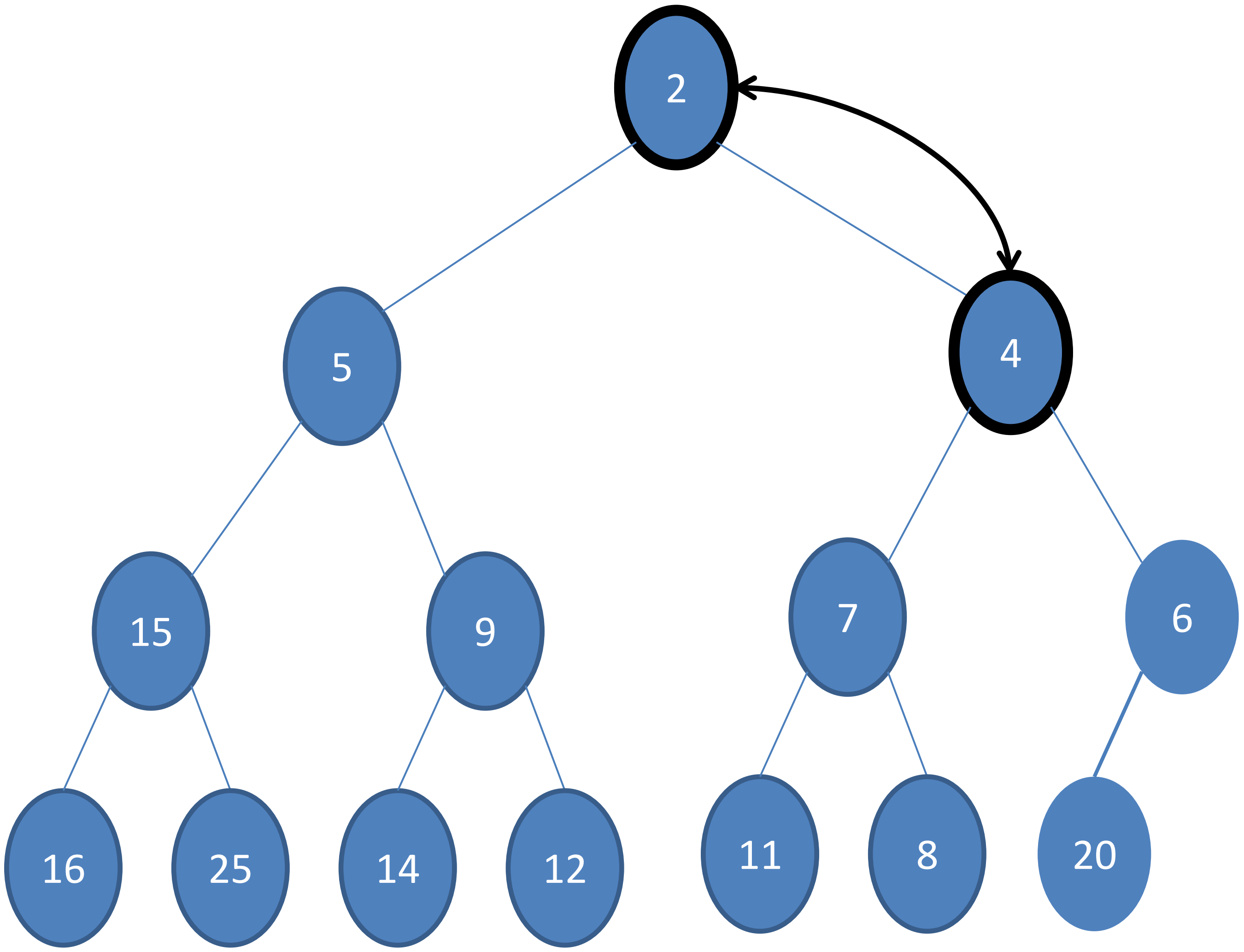


Insert 2

2 is less than parent 4 ... swap!

Example: adding to a heap

heap

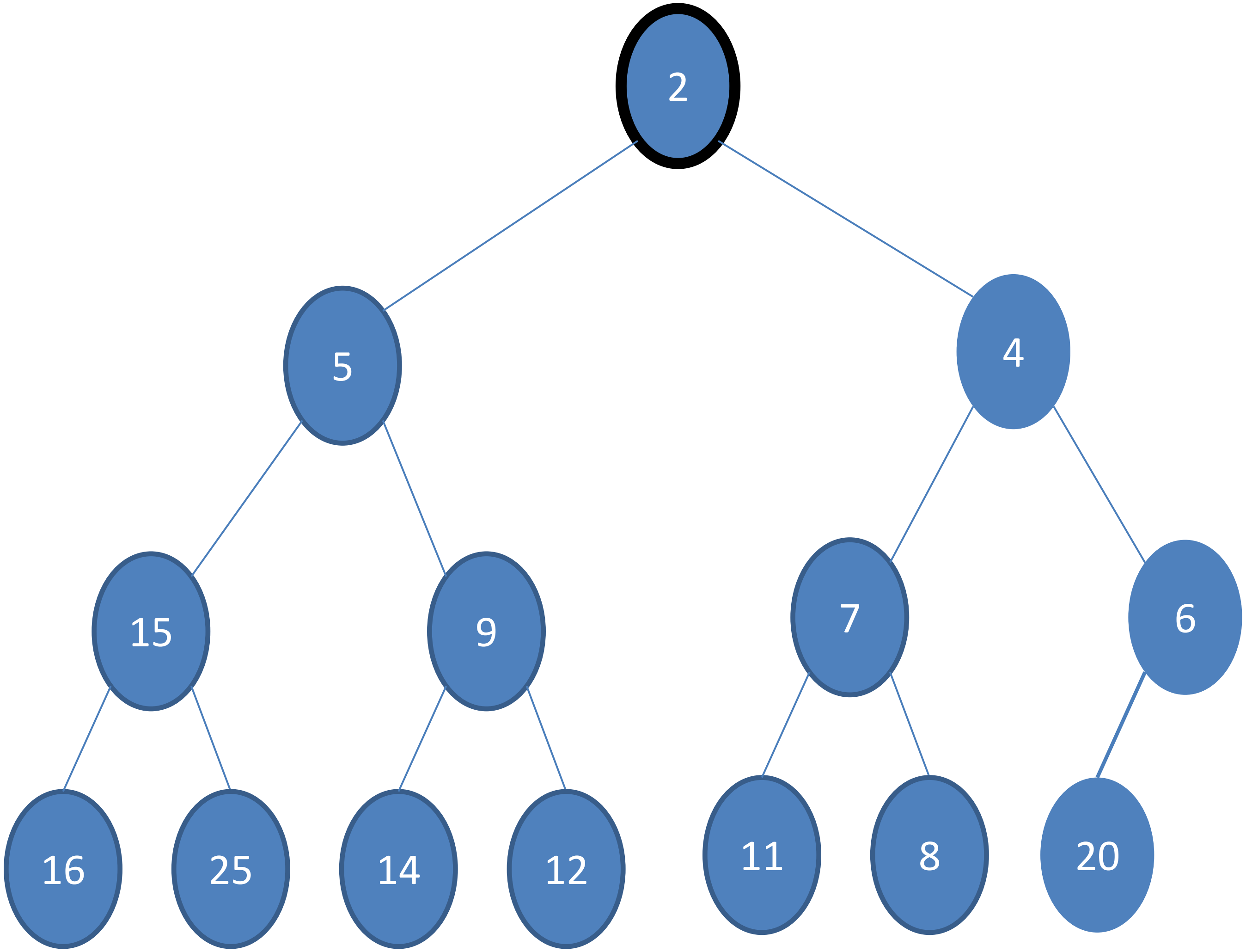


Insert 2

2 is less than parent 4 ... swap!

Example: adding to a heap

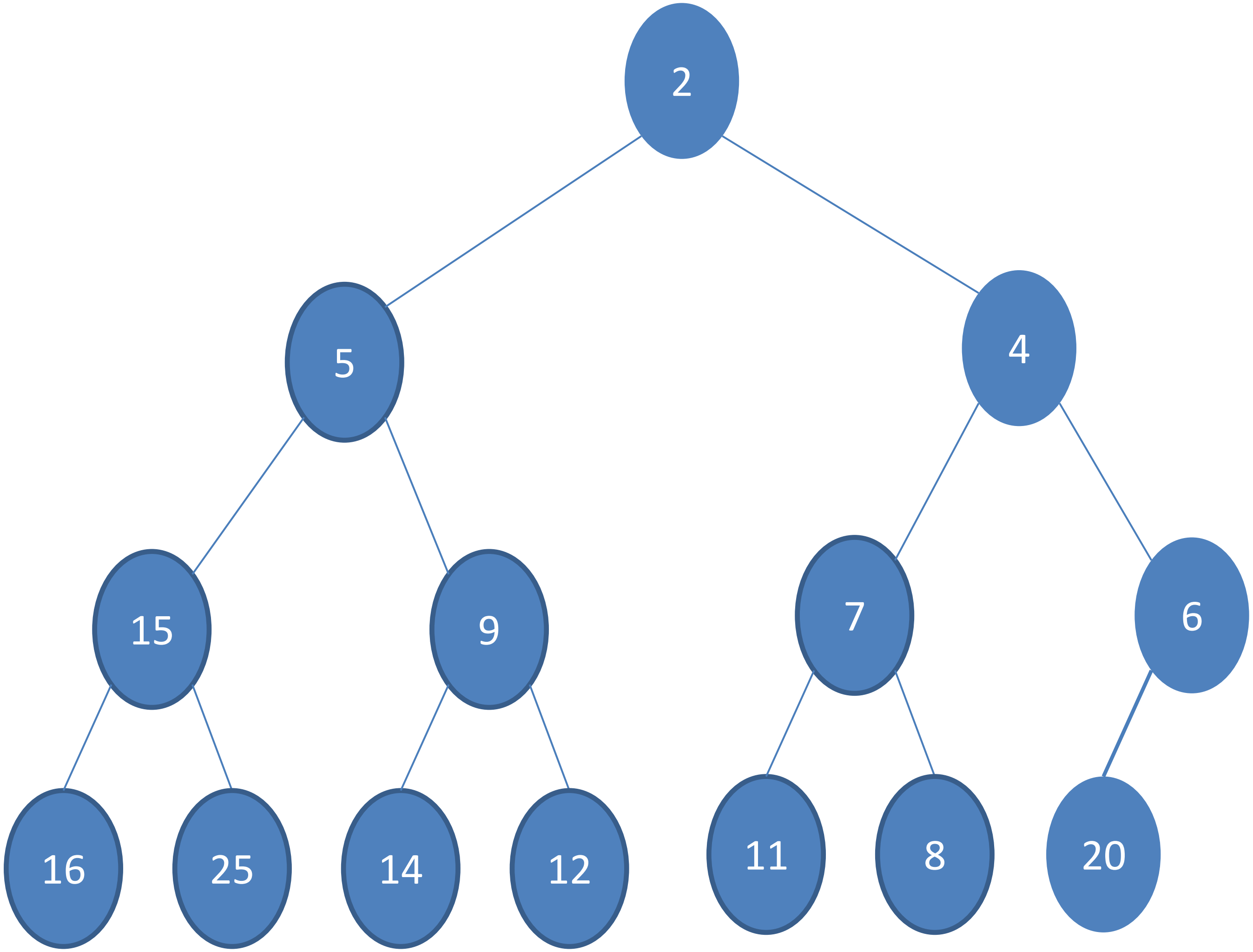
heap



Insert 2

Example: adding to a heap

heap

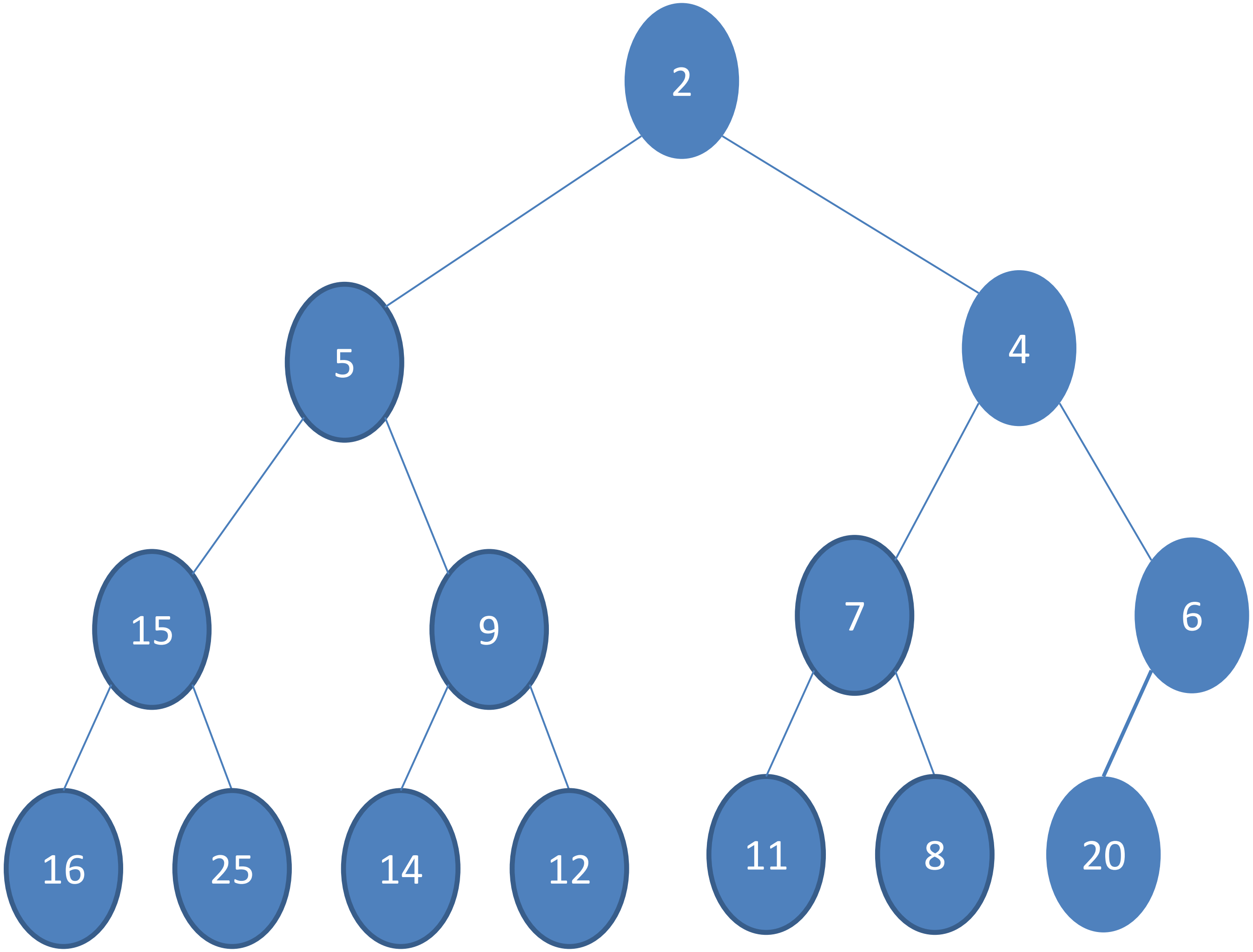


Insert 2

Done!

Example: adding to a heap

heap

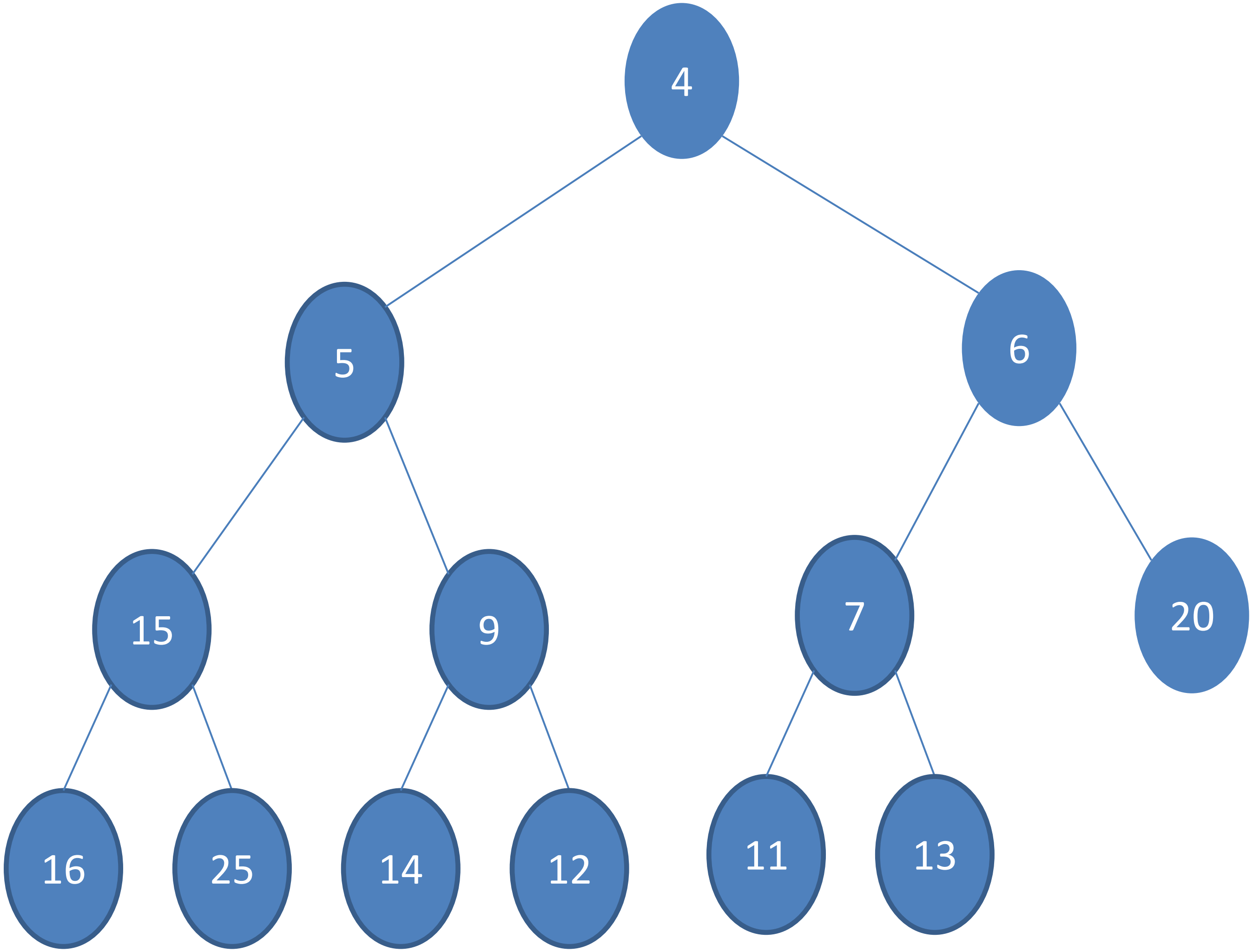


Example: removal from a heap

NOTE: it is a heap in a different state

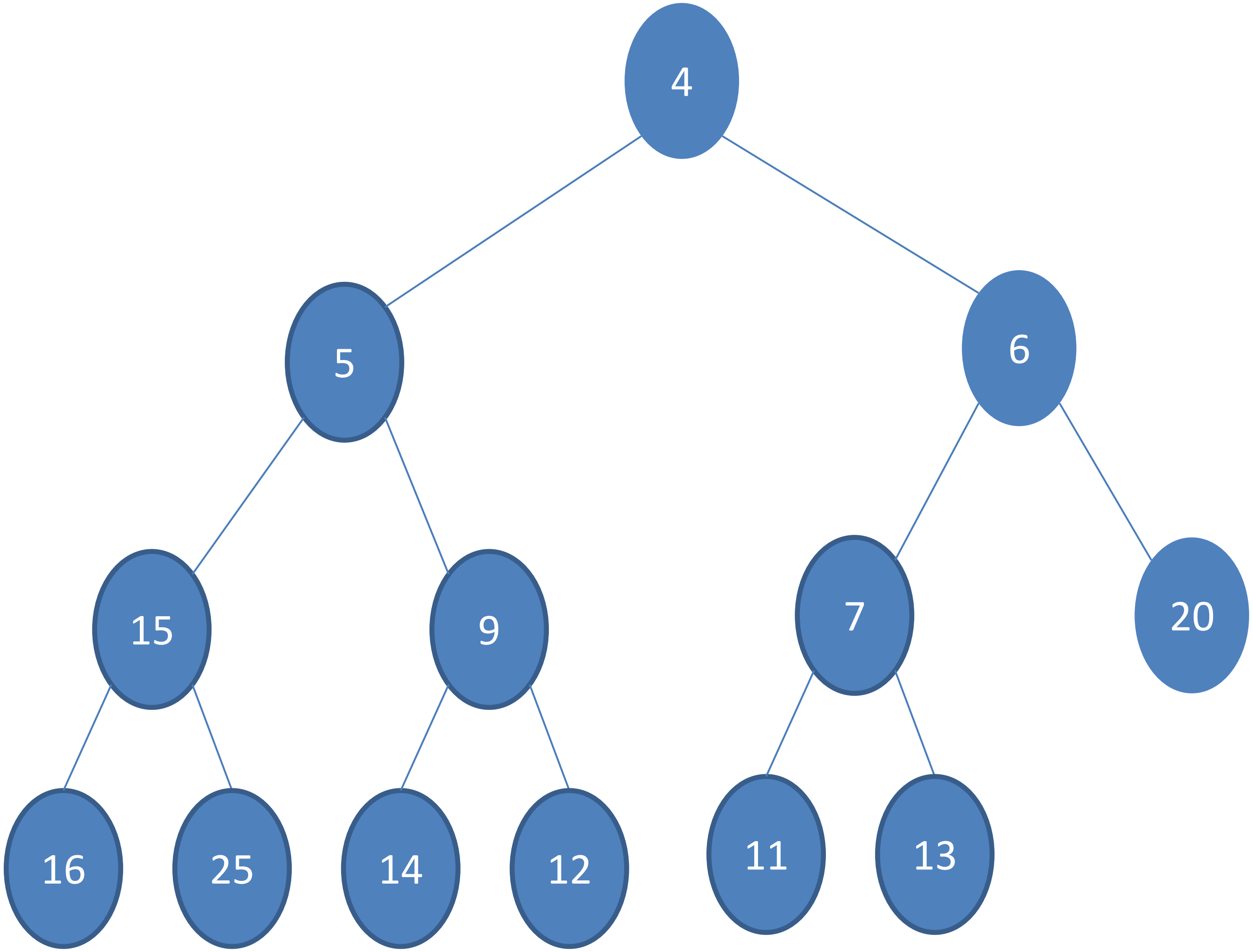
Example: removal from a heap

heap



Example: removal from a heap

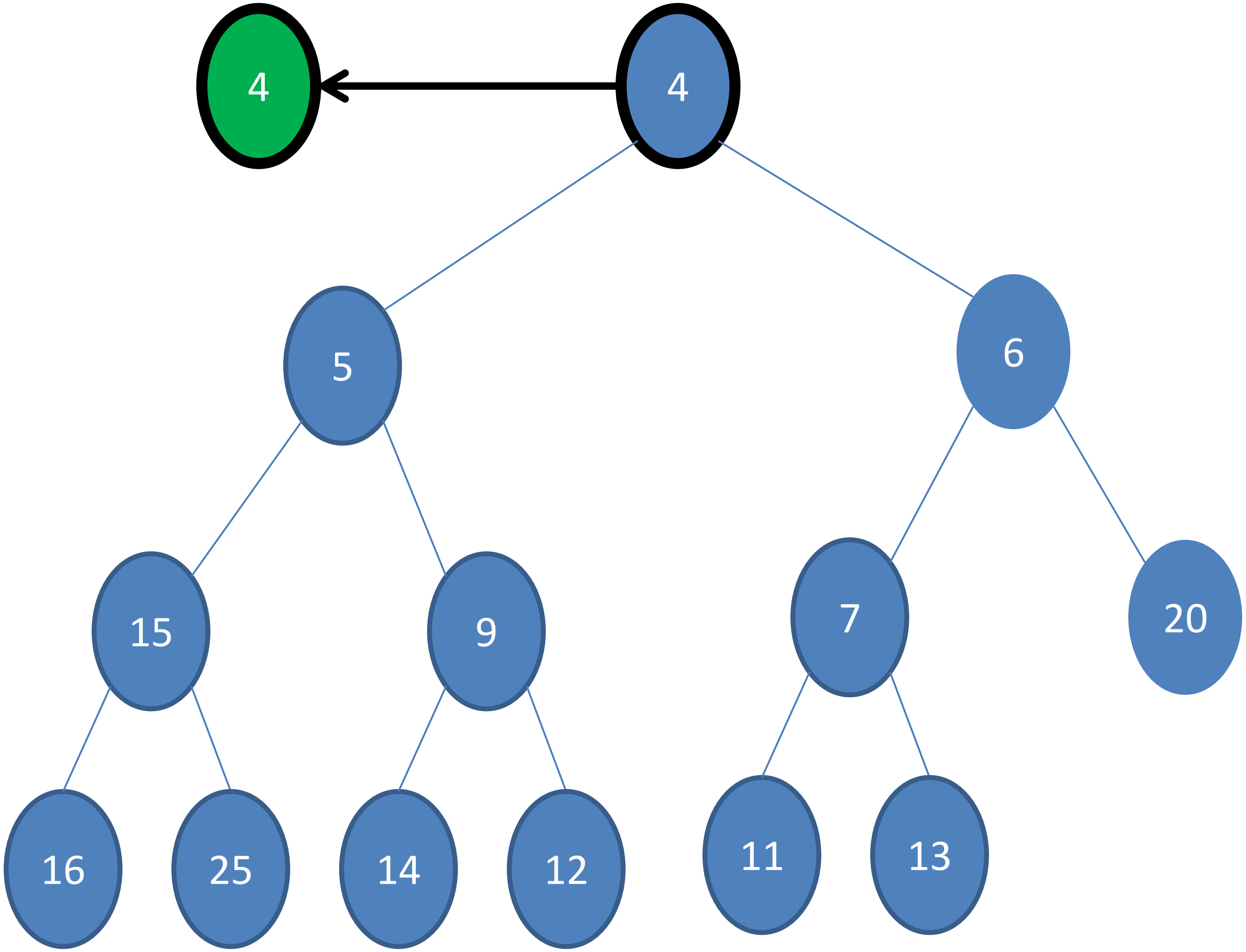
heap



Save off top of heap

Example: removal from a heap

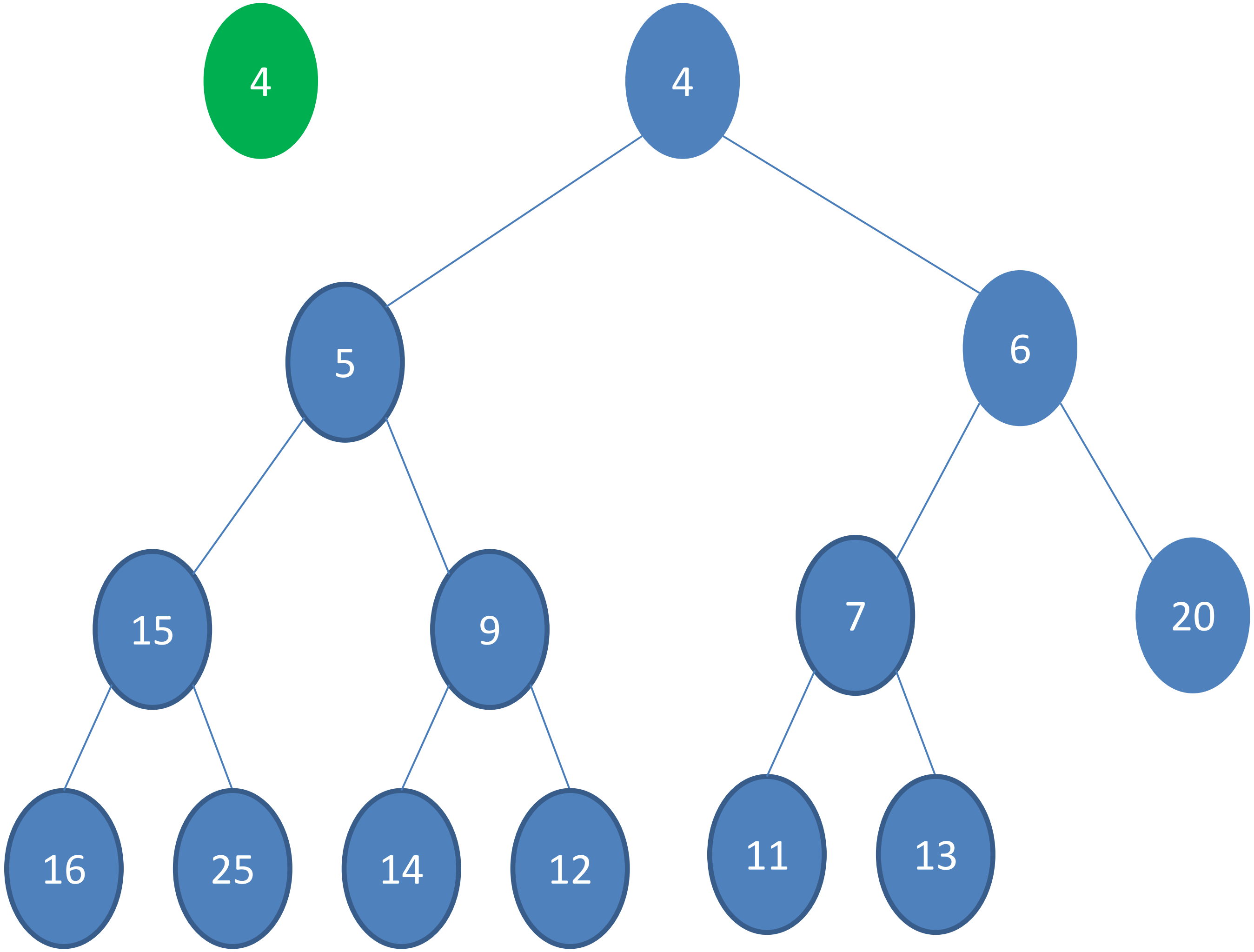
heap



Save off top of heap

Example: removal from a heap

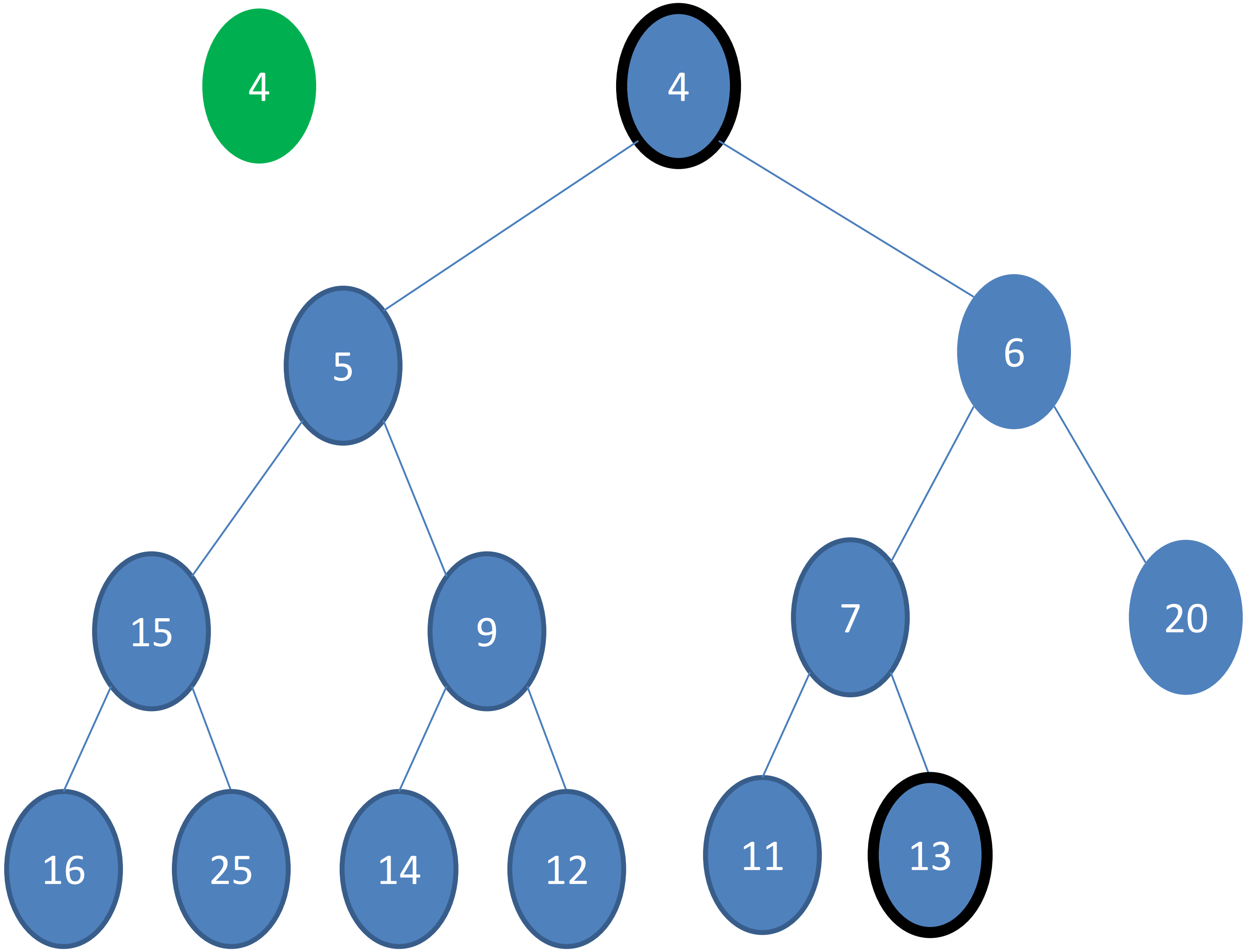
heap



Copy last item in heap to top of heap

Example: removal from a heap

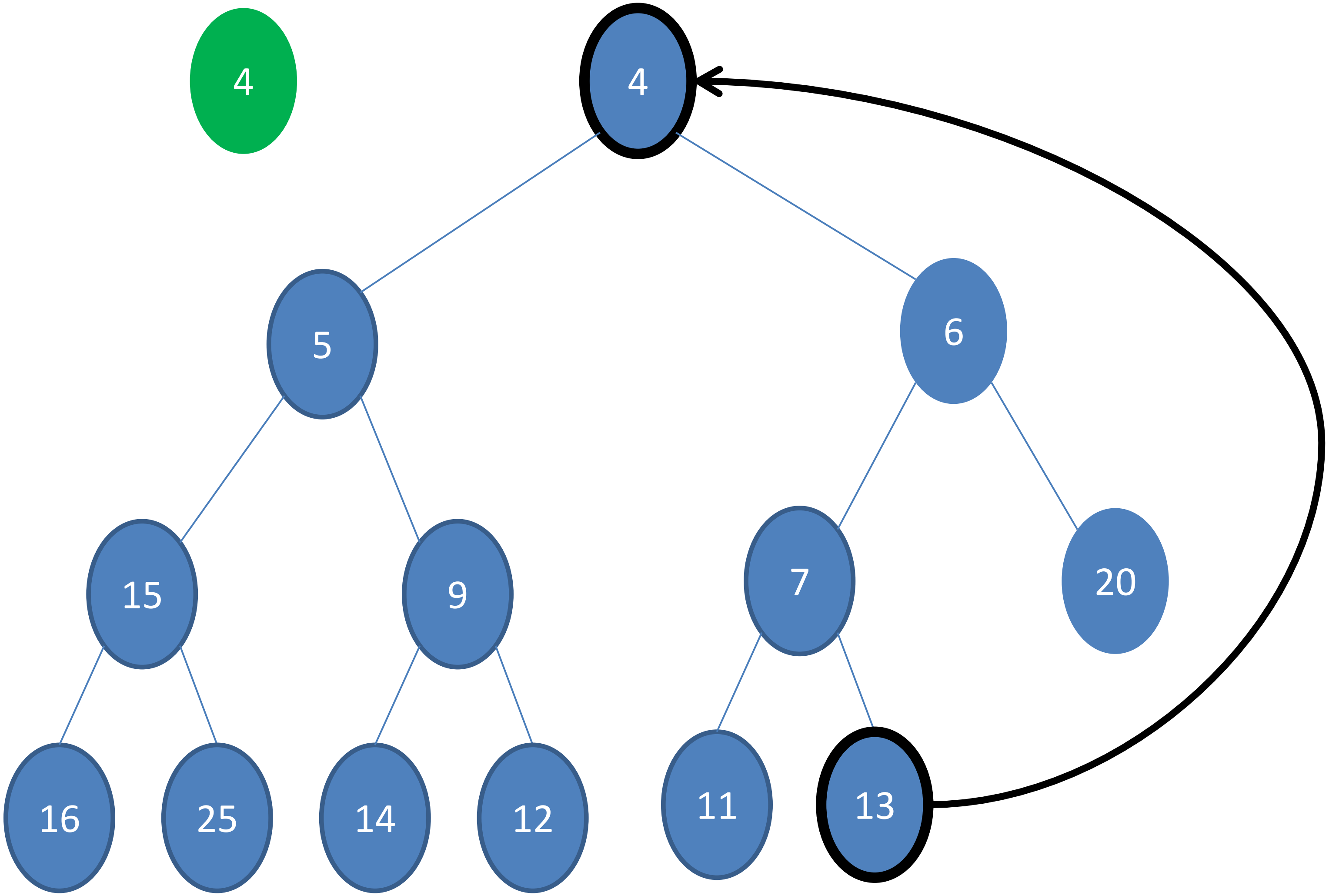
heap



Copy last item in heap to top of heap

Example: removal from a heap

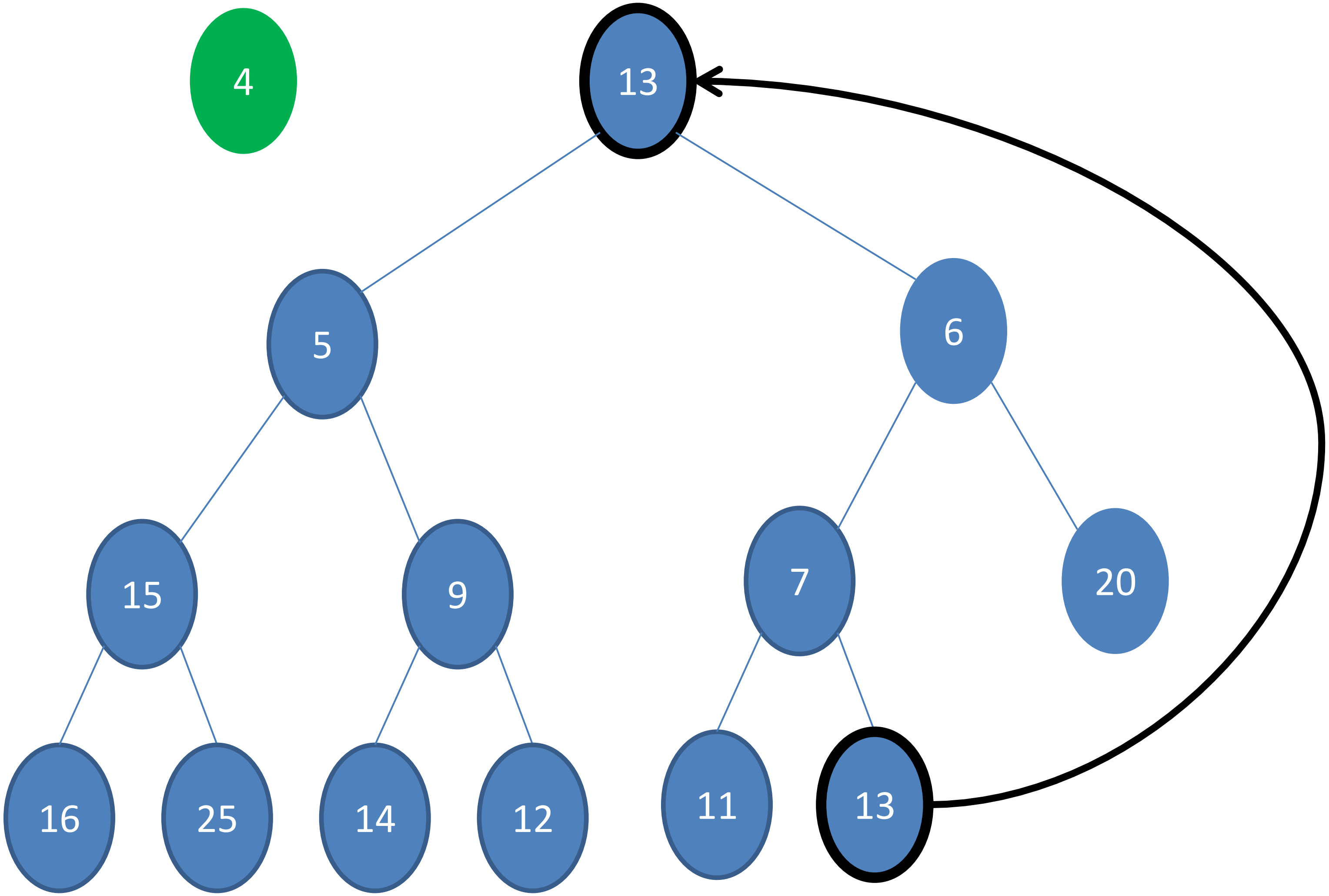
heap



Copy last item in heap to top of heap

Example: removal from a heap

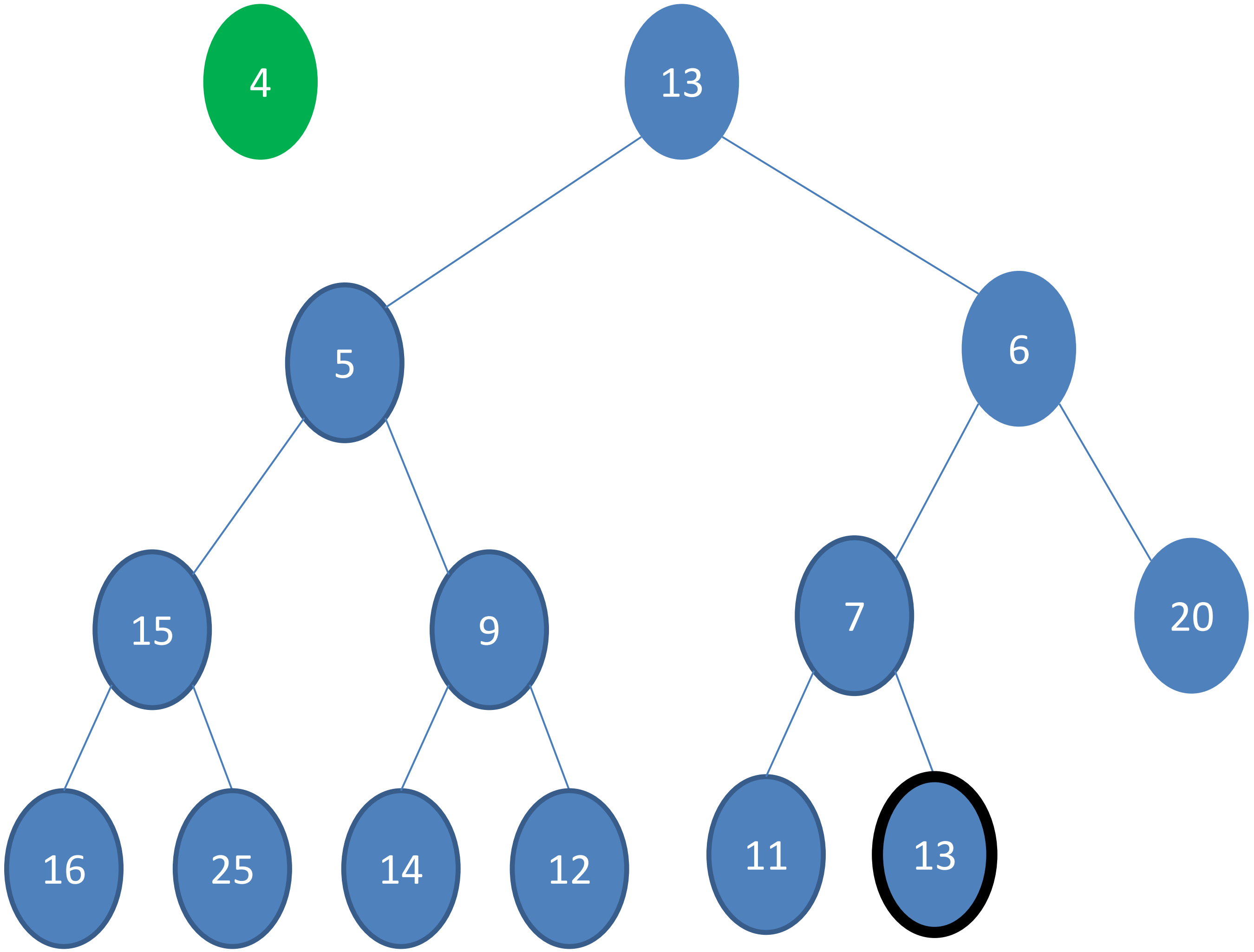
heap



Copy last item in heap to top of heap

Example: removal from a heap

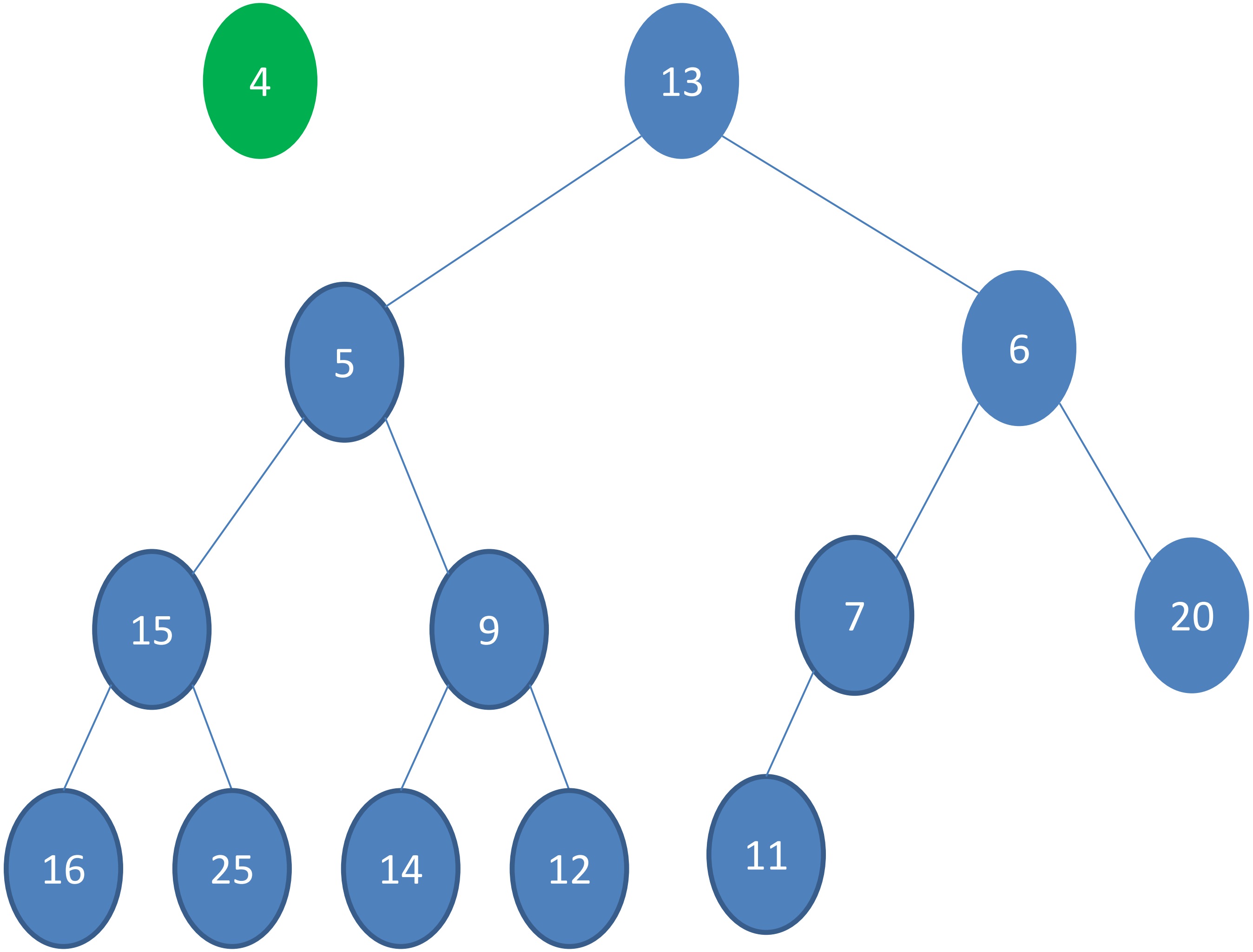
heap



Delete last item in heap

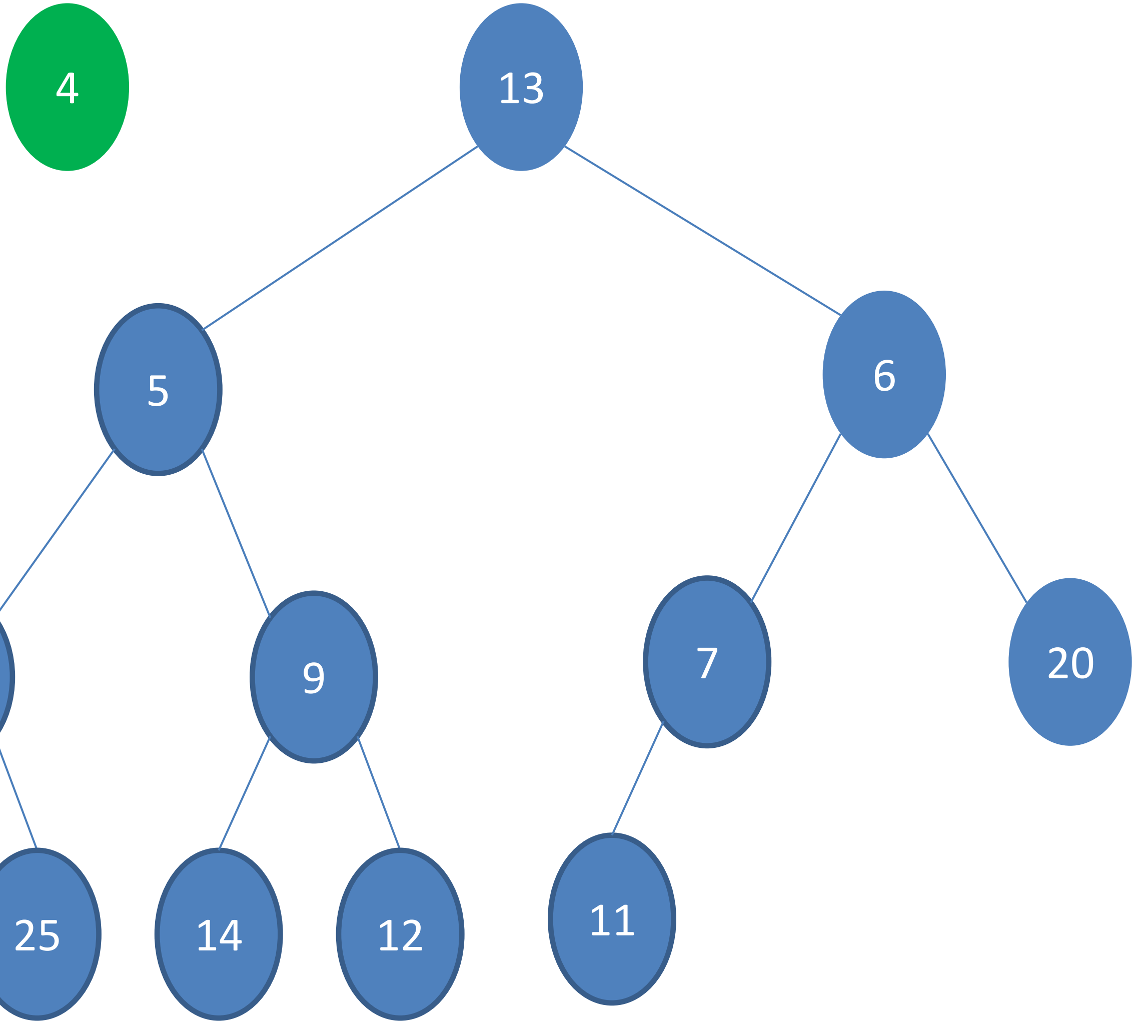
Example: removal from a heap

heap



Example: removal from a heap

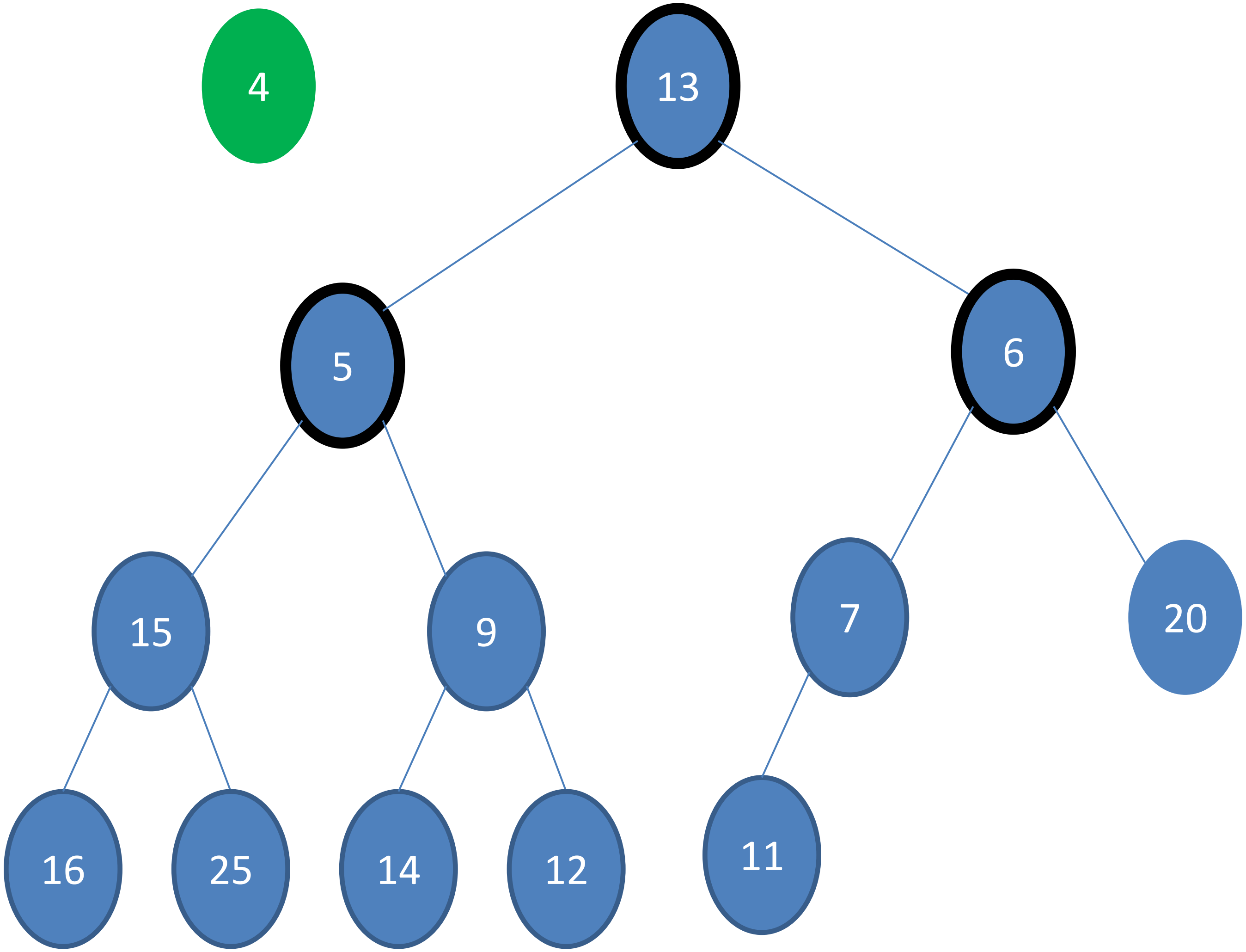
heap



Compare current node with its children

Example: removal from a heap

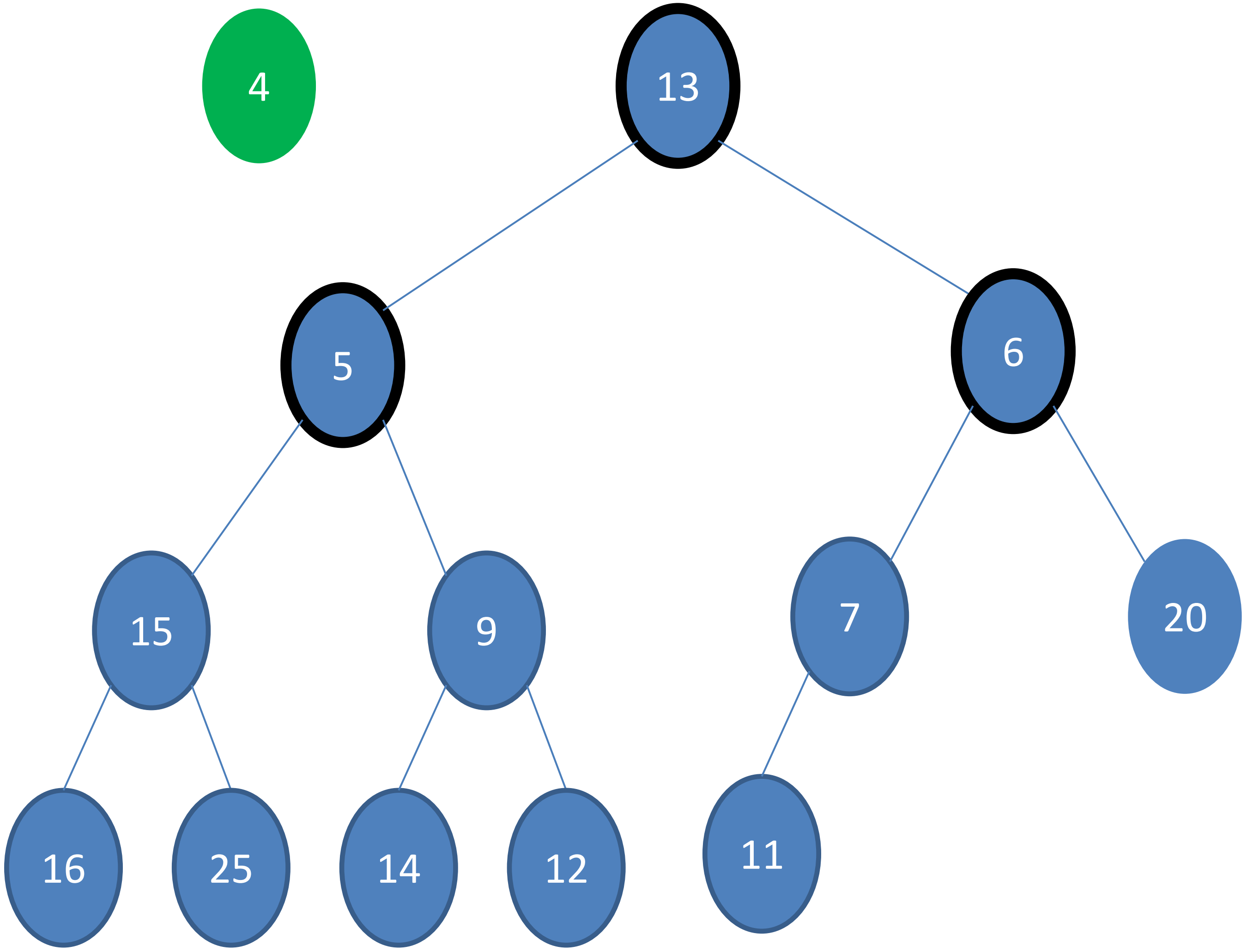
heap



Compare current node with its children

Example: removal from a heap

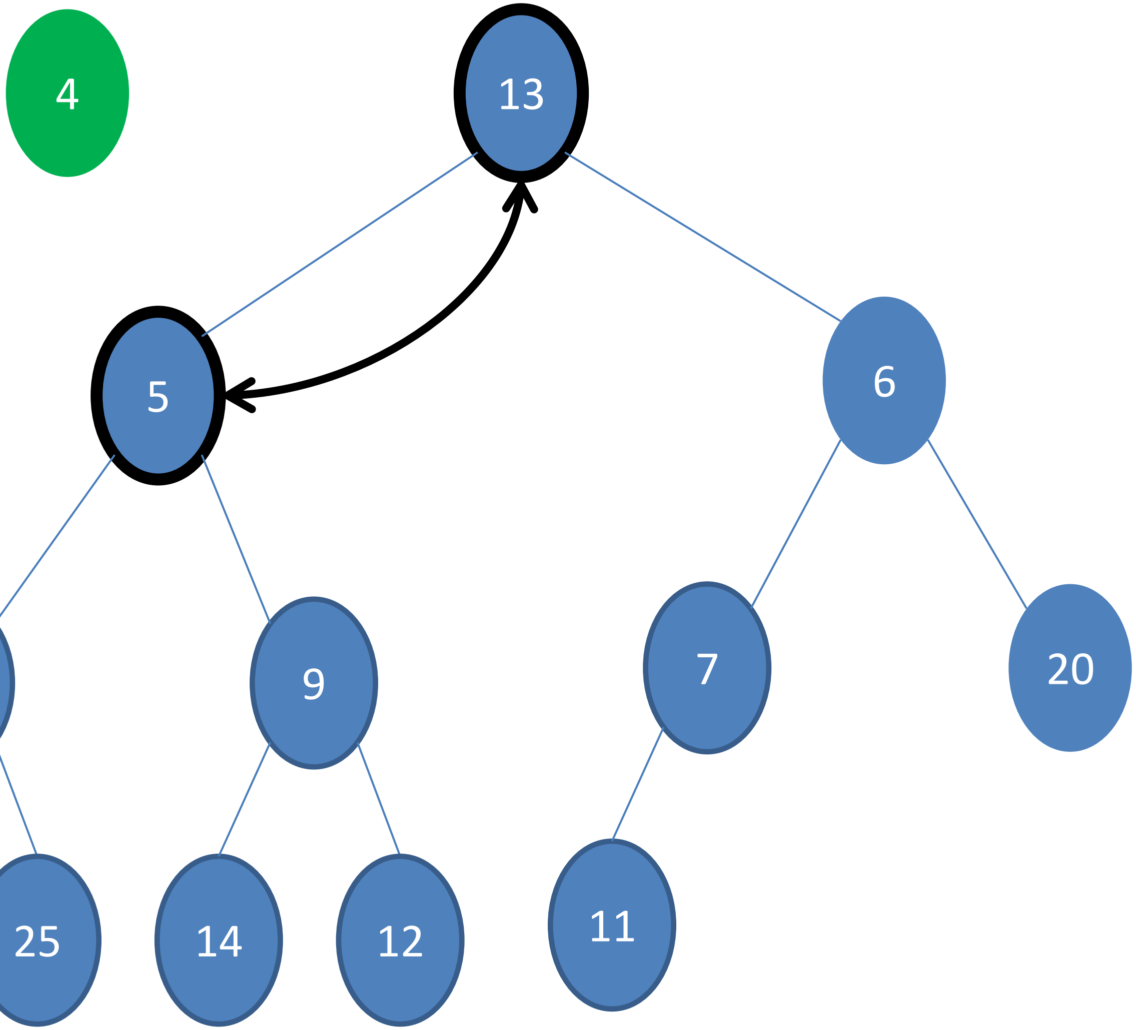
heap



If greater then swap with smallest child

Example: removal from a heap

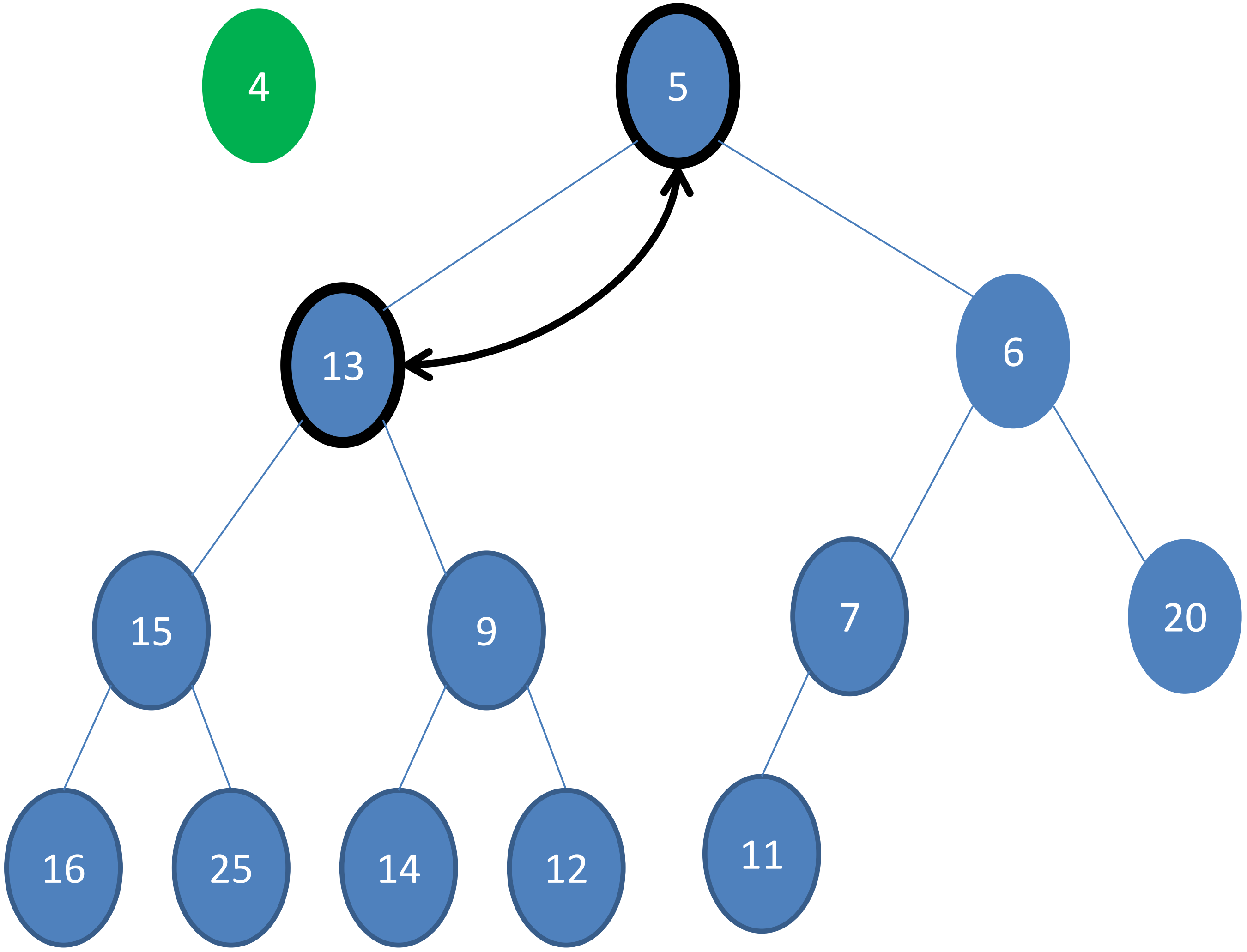
heap



If greater then swap with smallest child

Example: removal from a heap

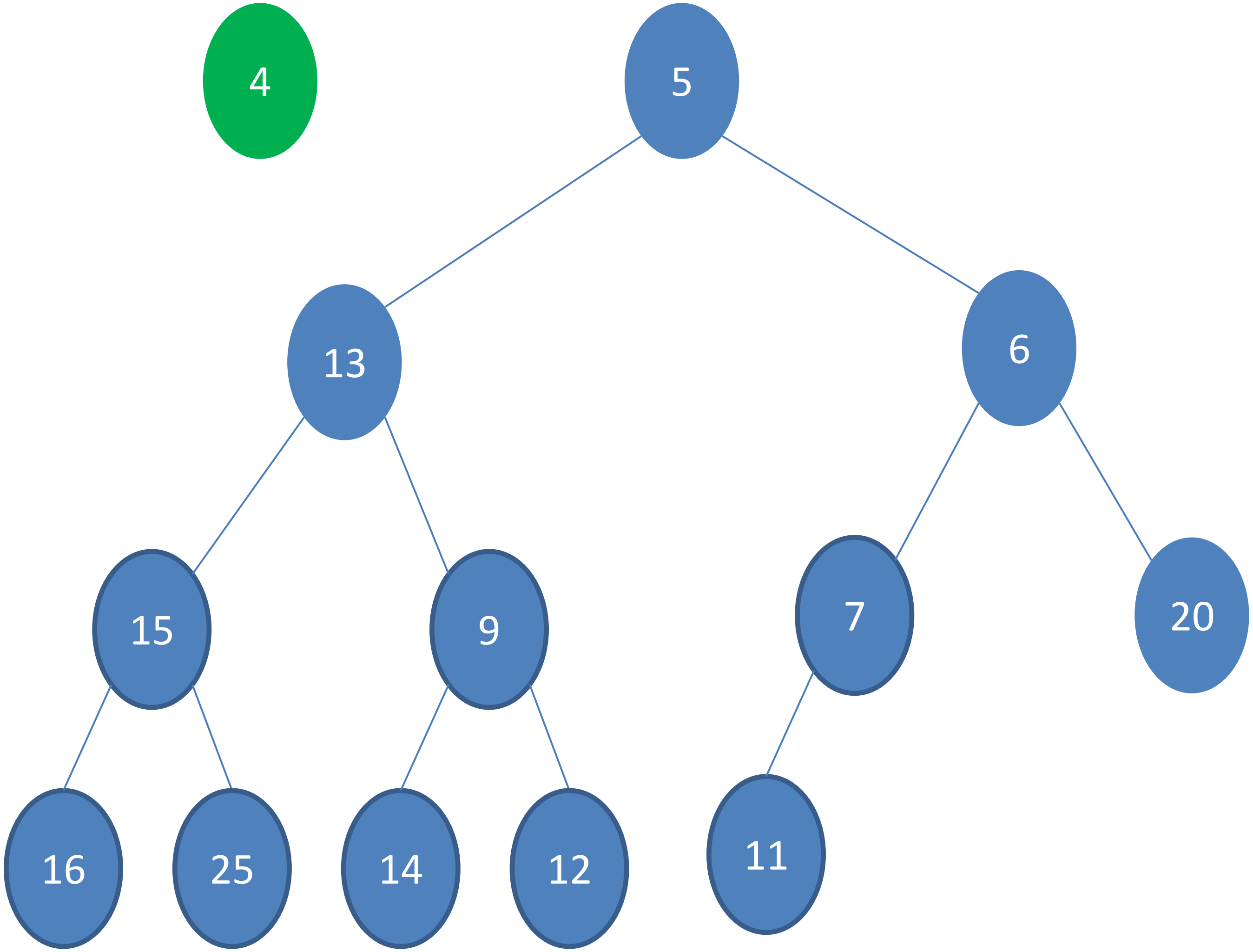
heap



If greater then swap with smallest child

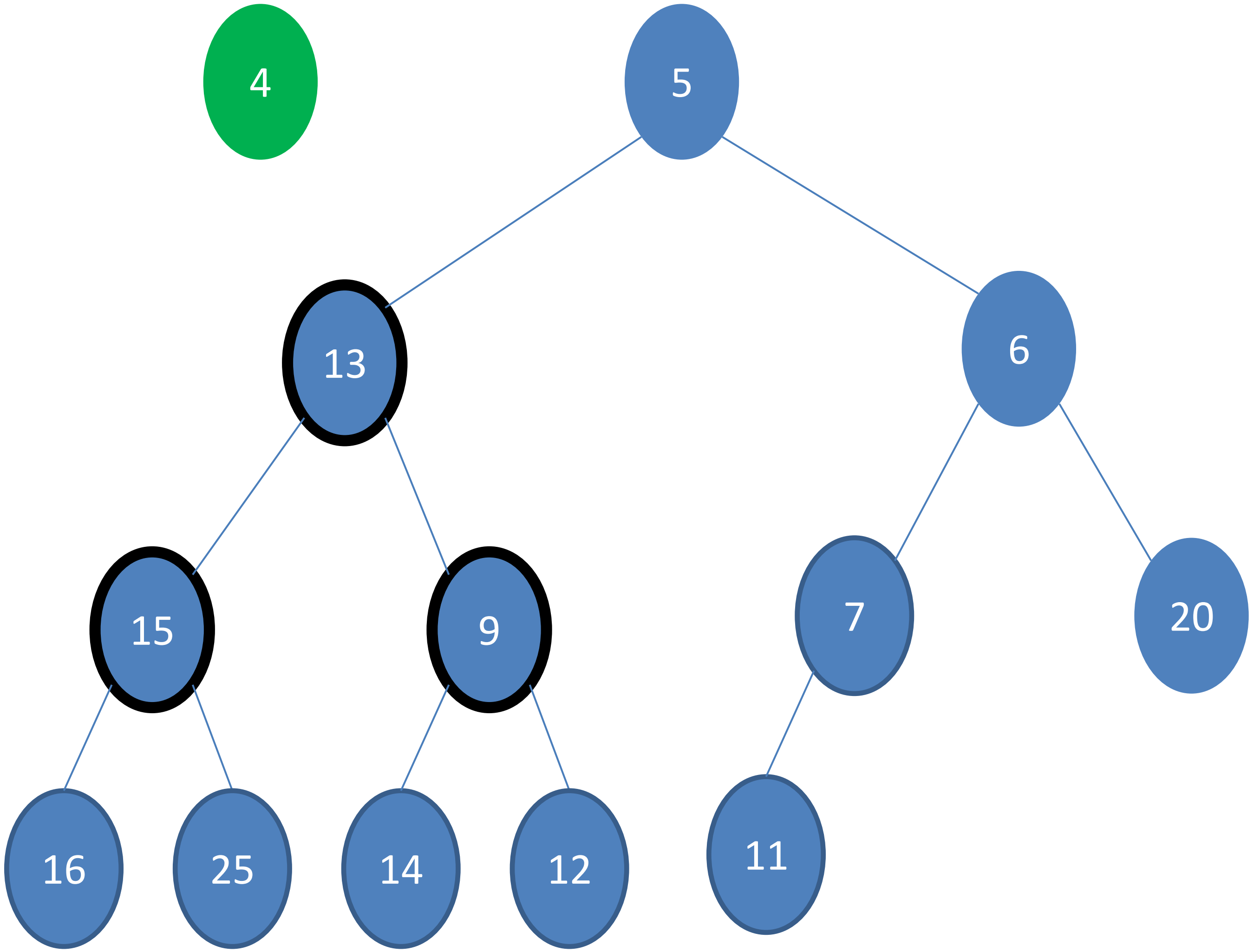
Example: removal from a heap

heap



Example: removal from a heap

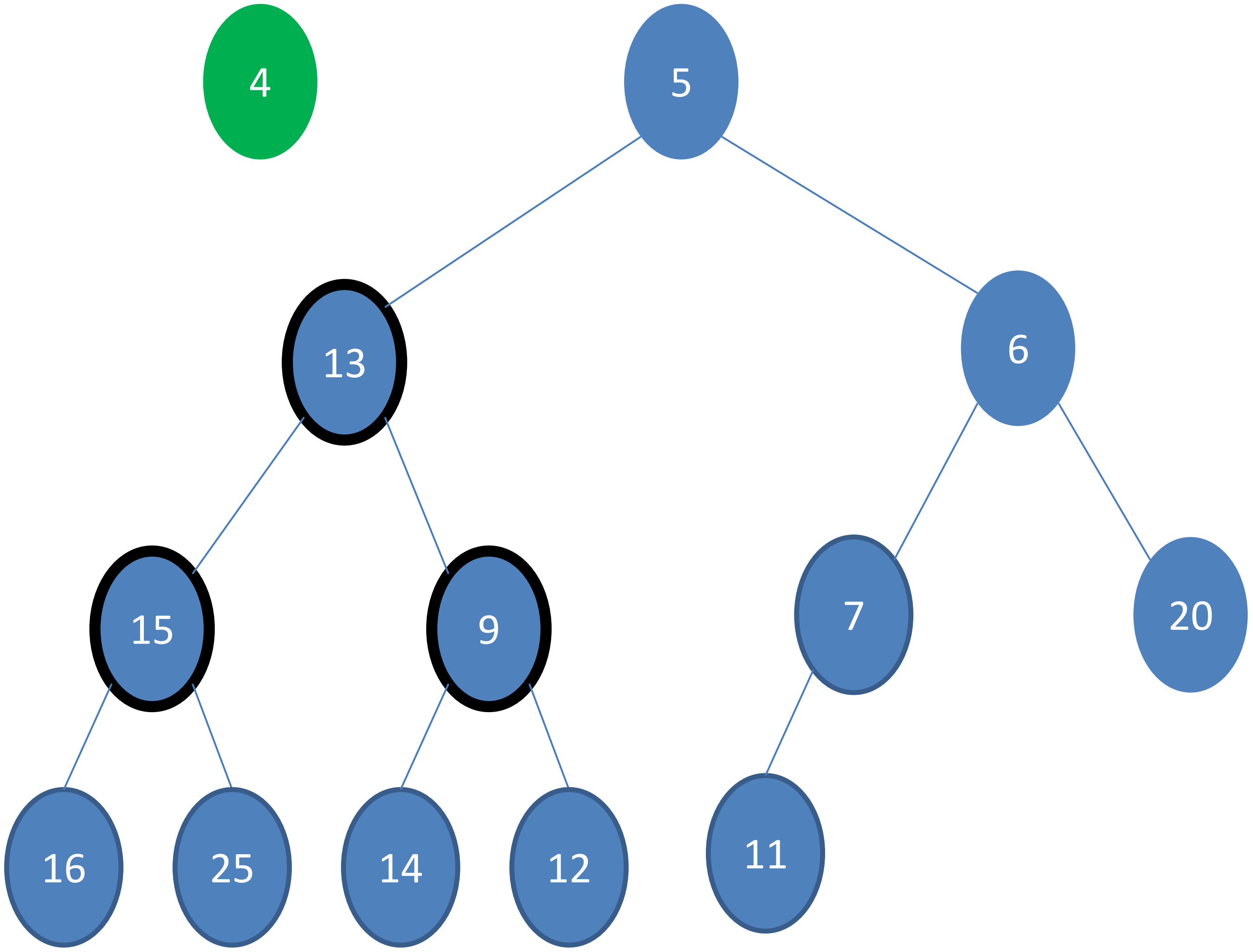
heap



Compare current node with its children

Example: removal from a heap

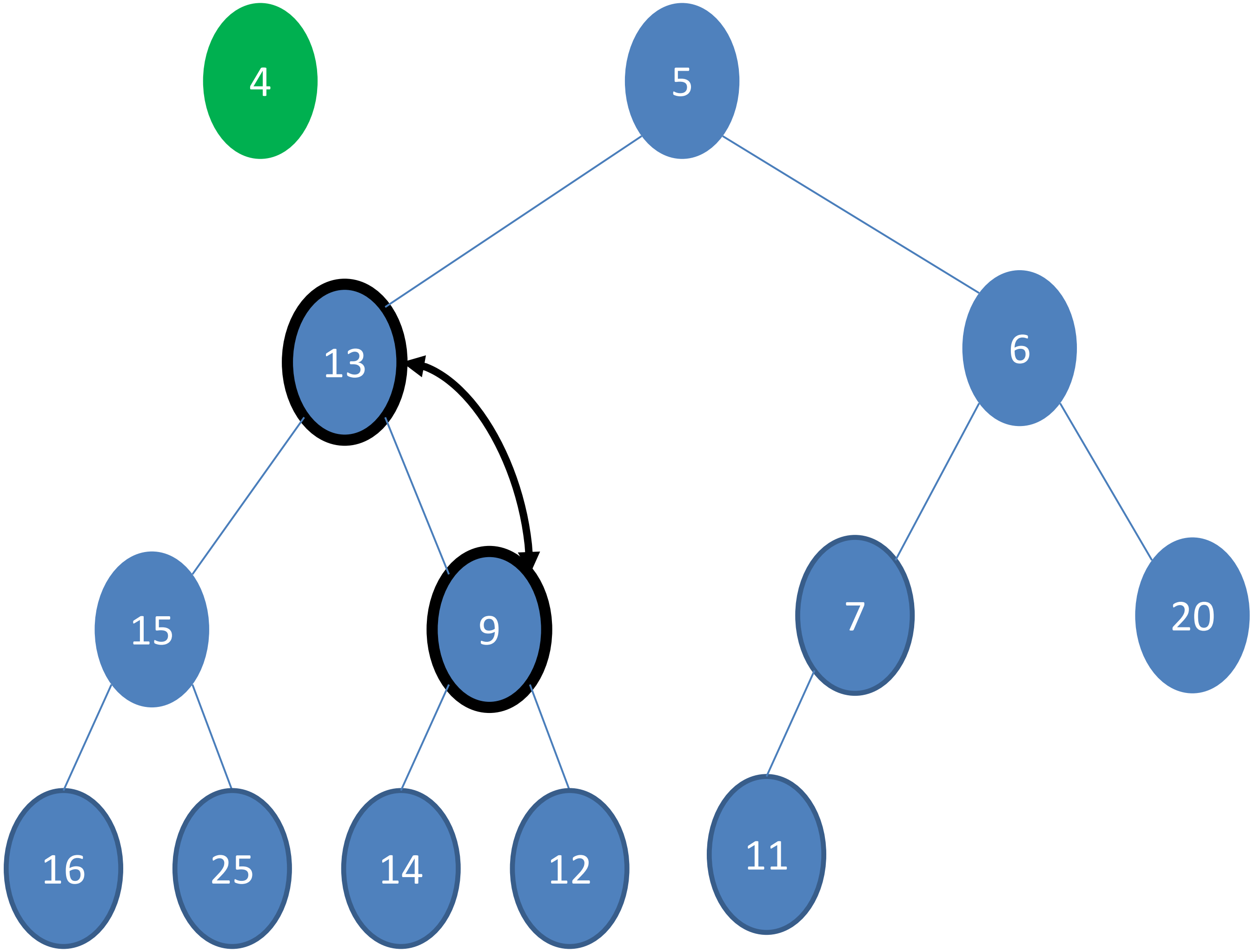
heap



If greater then swap with smallest child

Example: removal from a heap

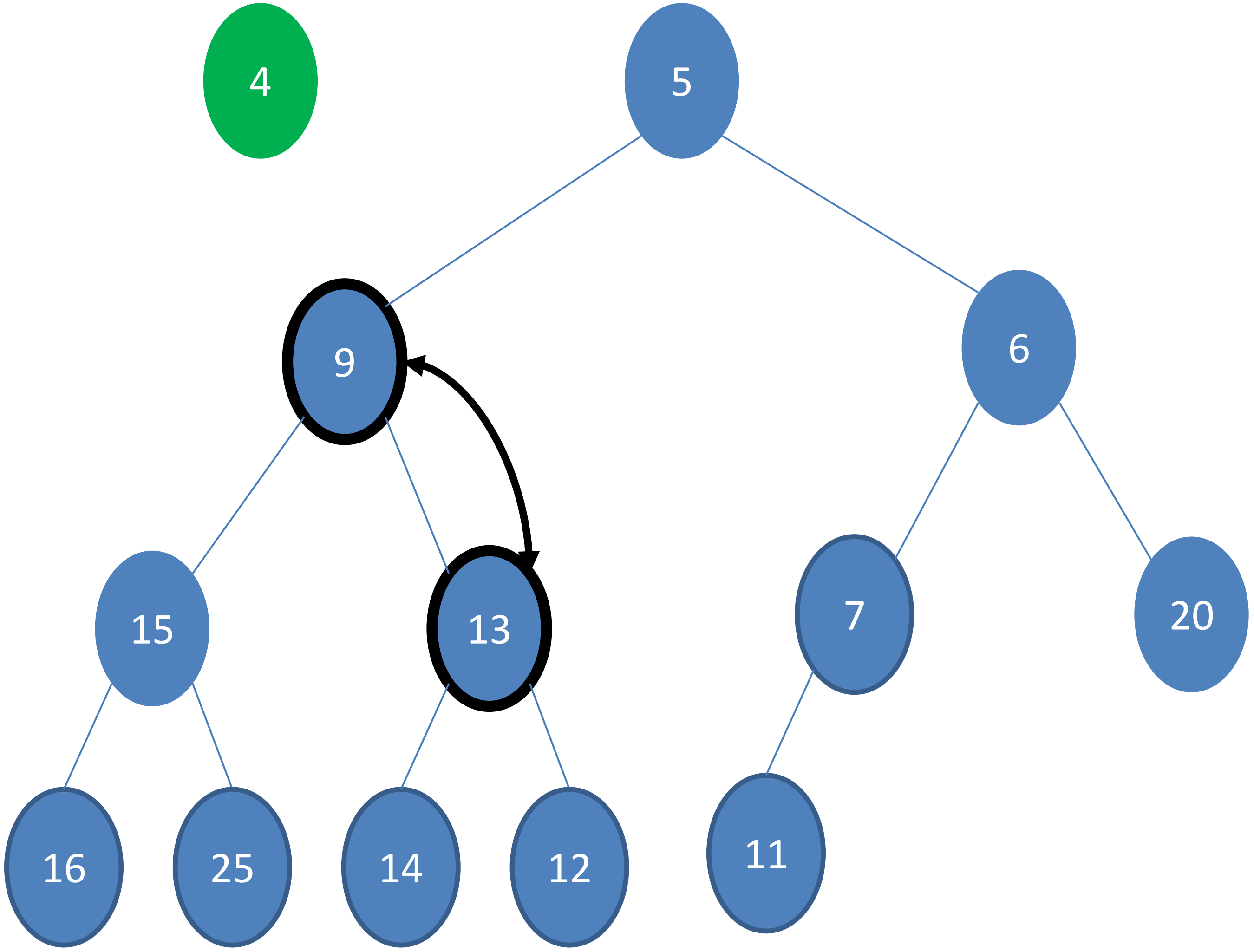
heap



If greater then swap with smallest child

Example: removal from a heap

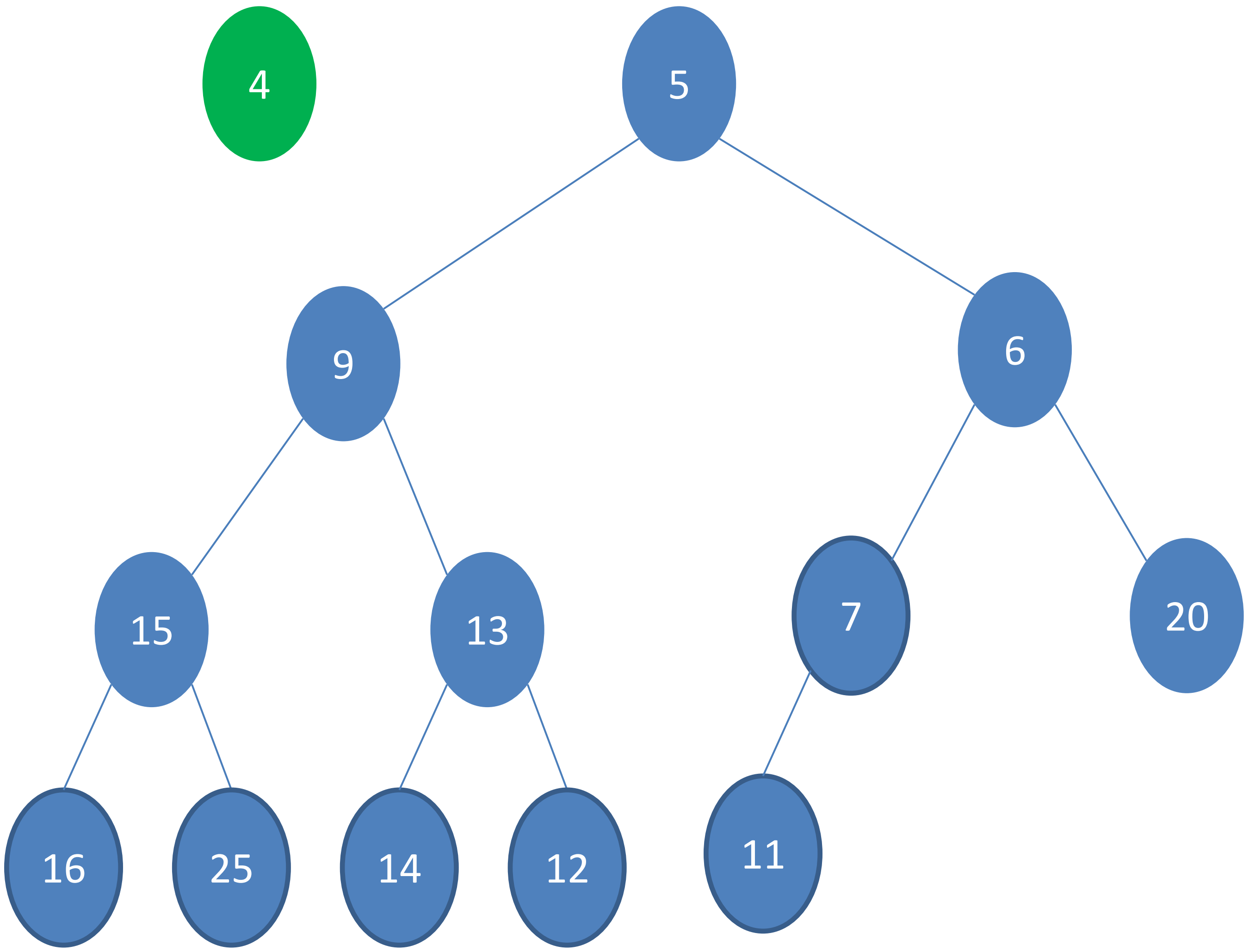
heap



If greater then swap with smallest child

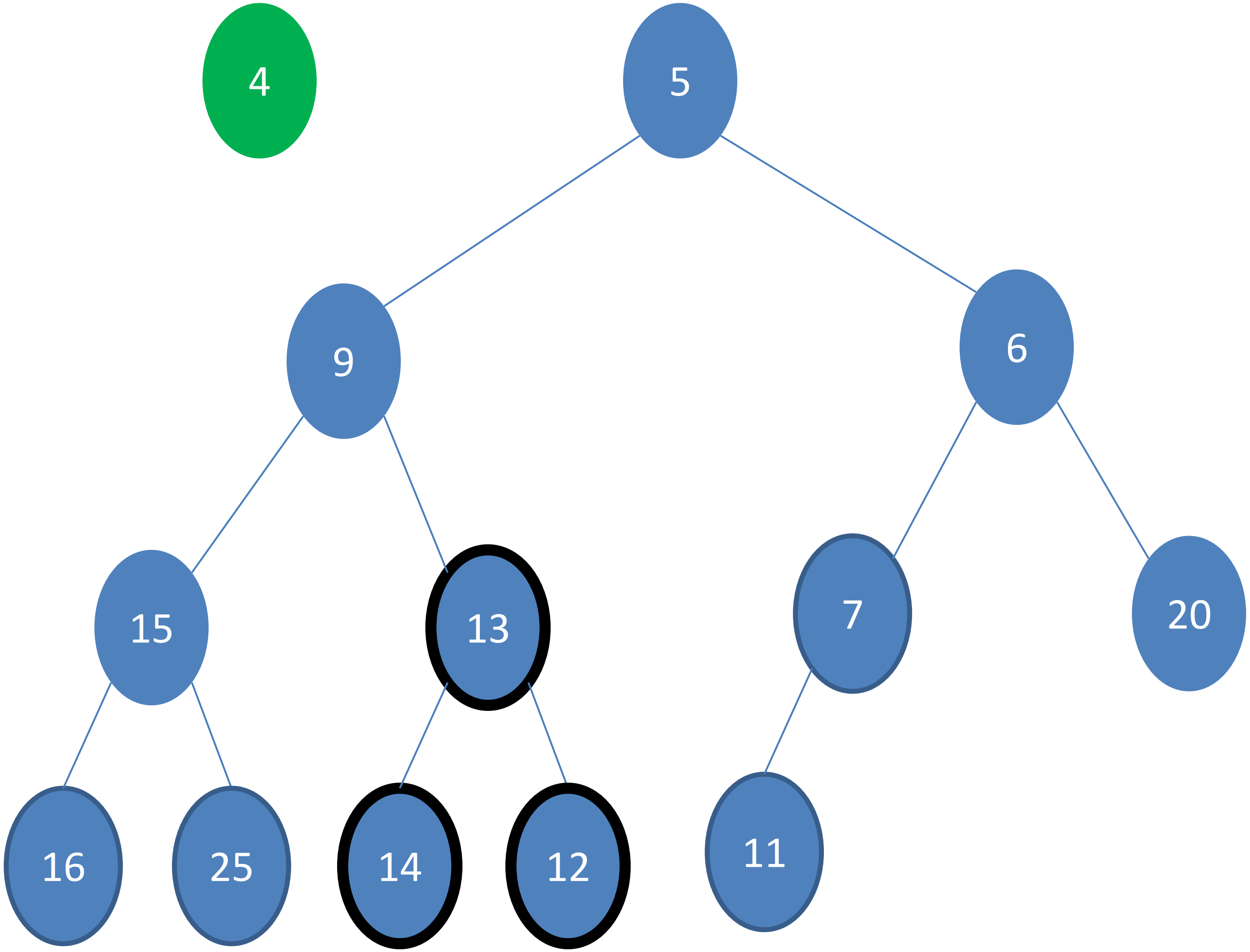
Example: removal from a heap

heap



Example: removal from a heap

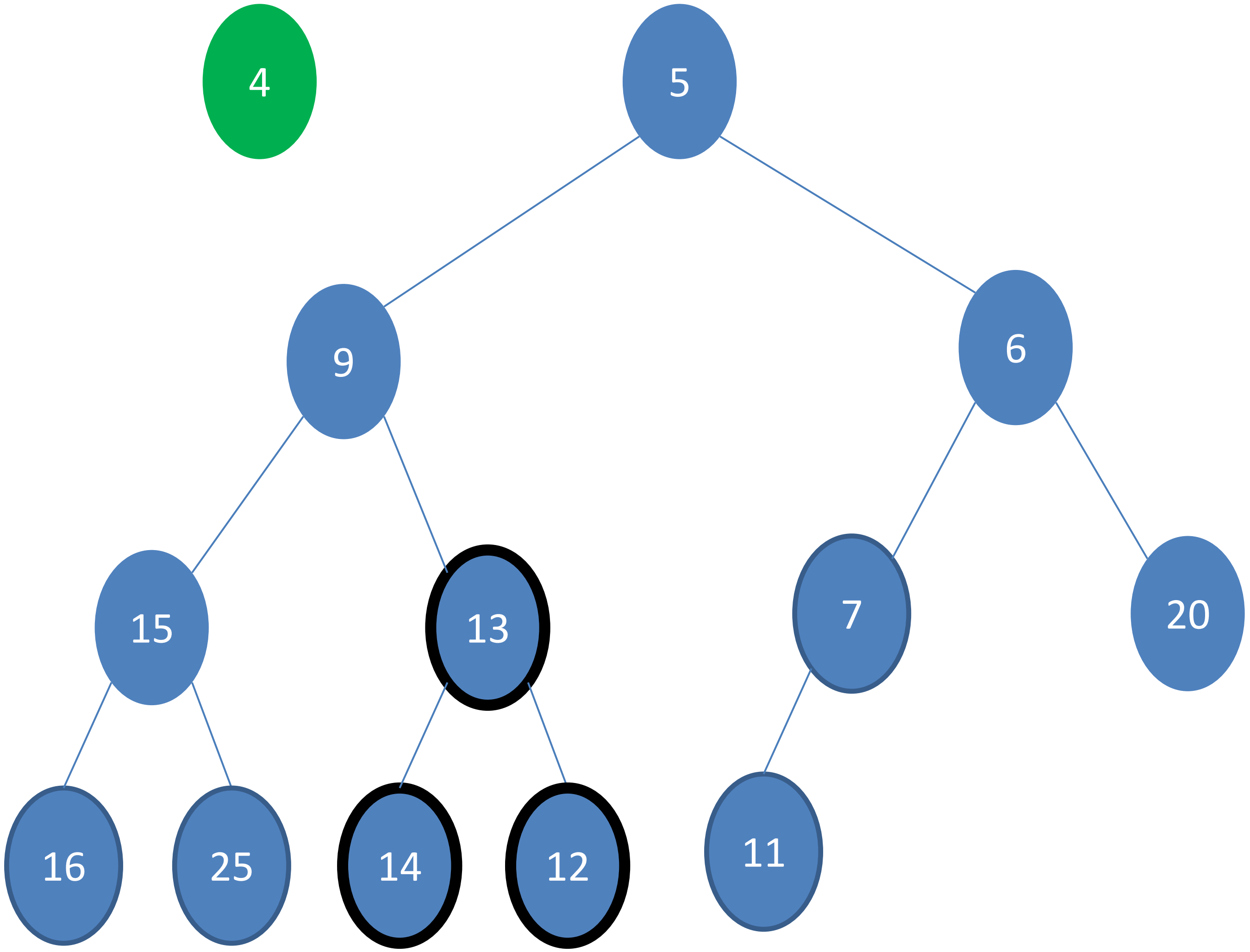
heap



Compare current node with its children

Example: removal from a heap

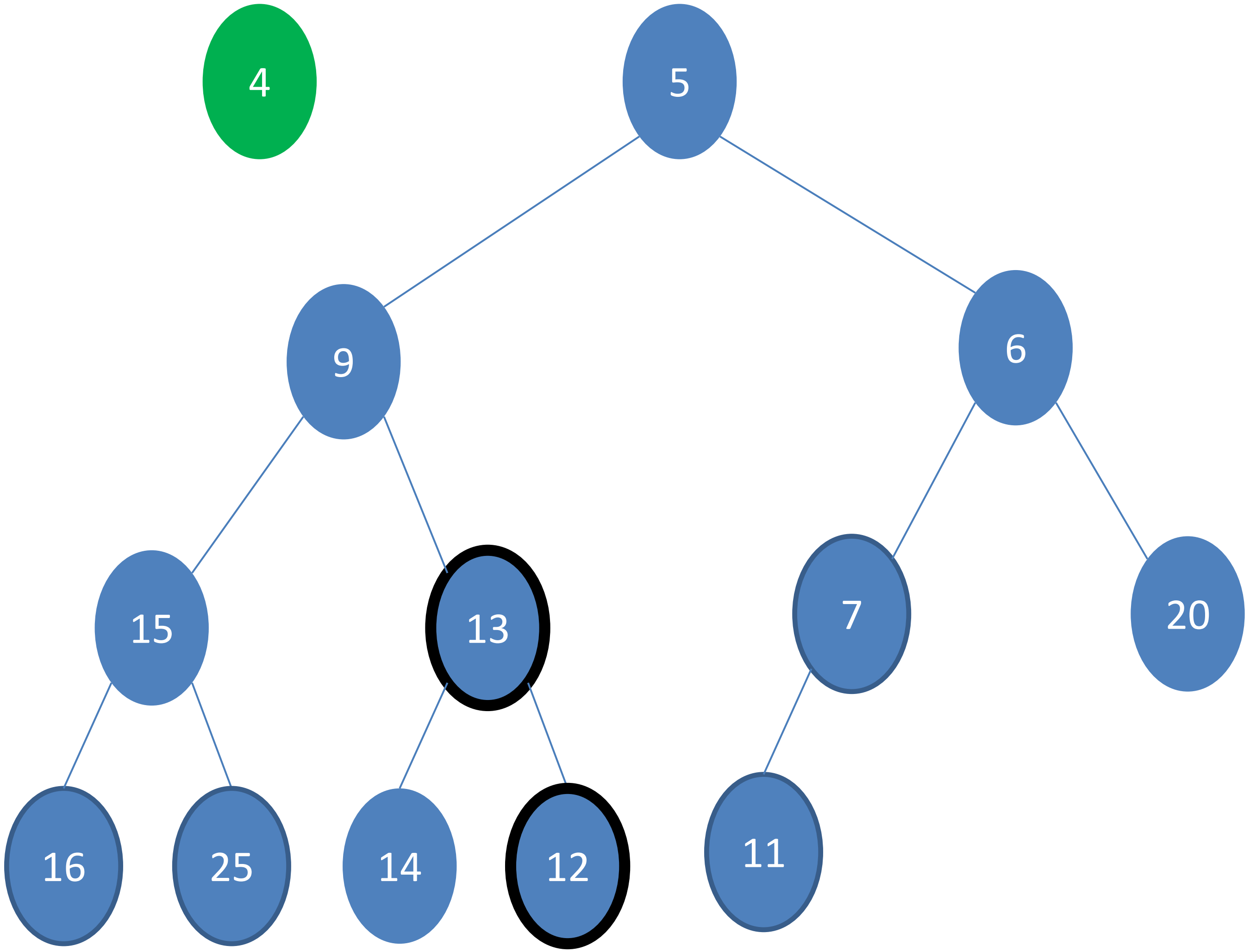
heap



If greater then swap with smallest child

Example: removal from a heap

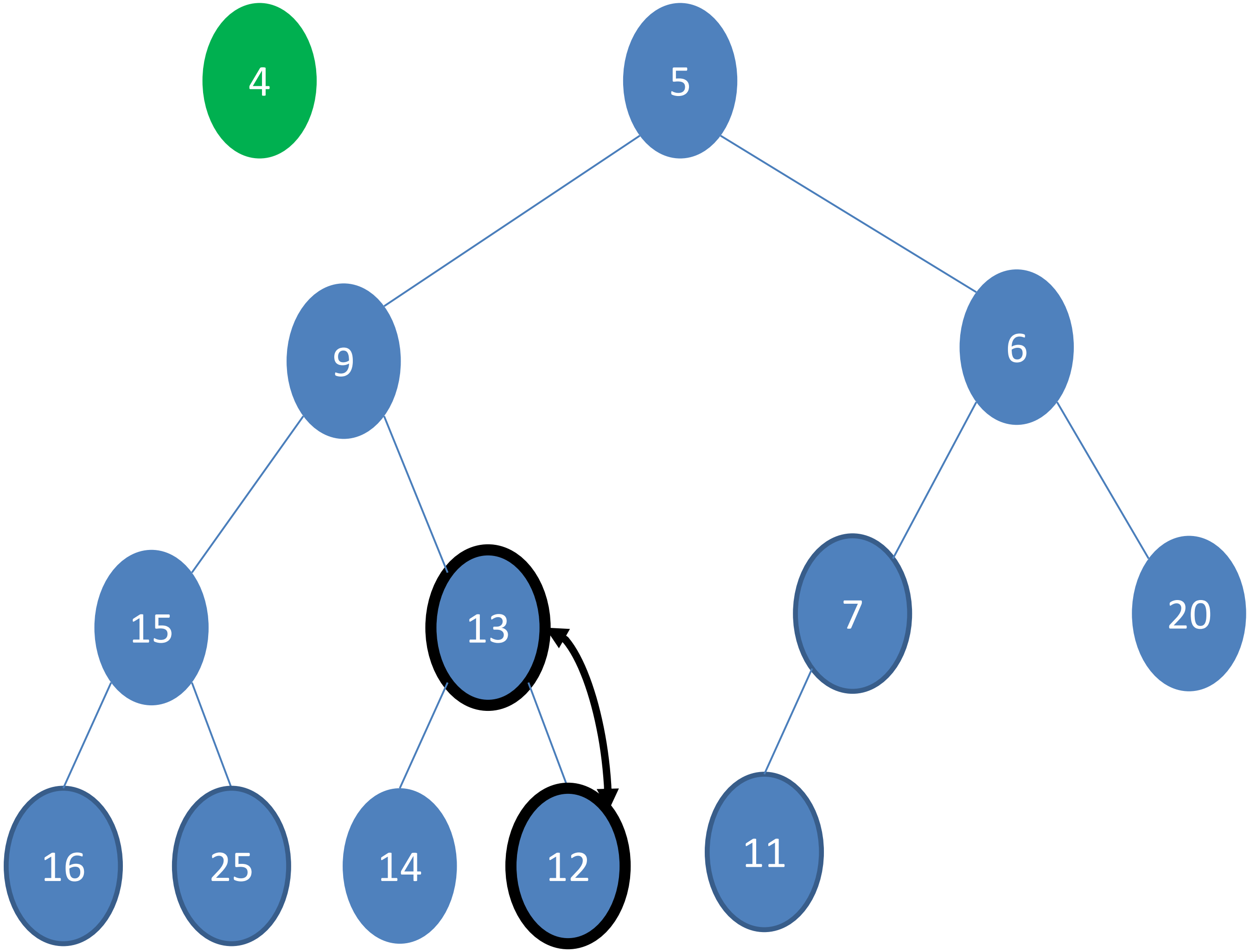
heap



If greater then swap with smallest child

Example: removal from a heap

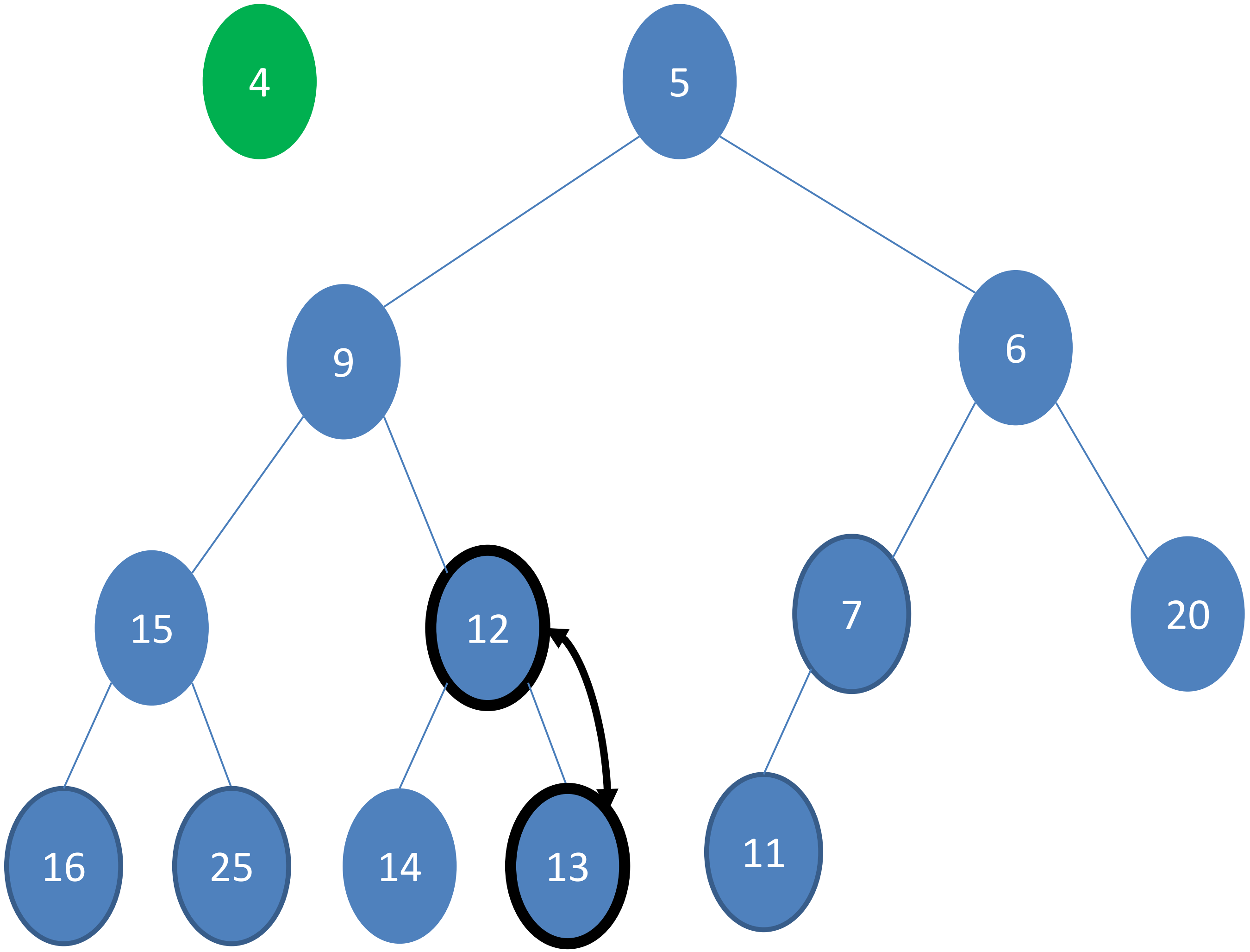
heap



If greater then swap with smallest child

Example: removal from a heap

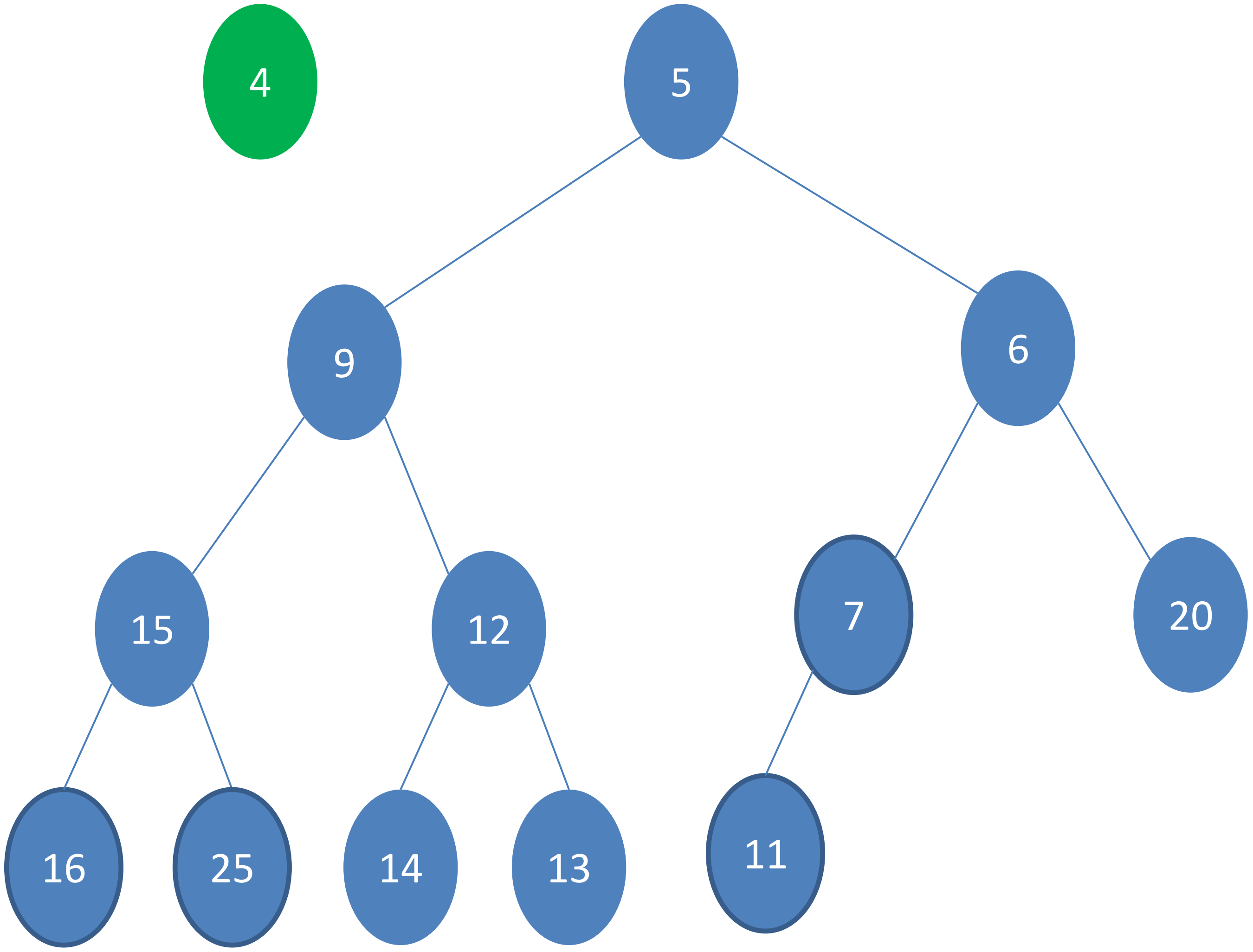
heap



If greater then swap with smallest child

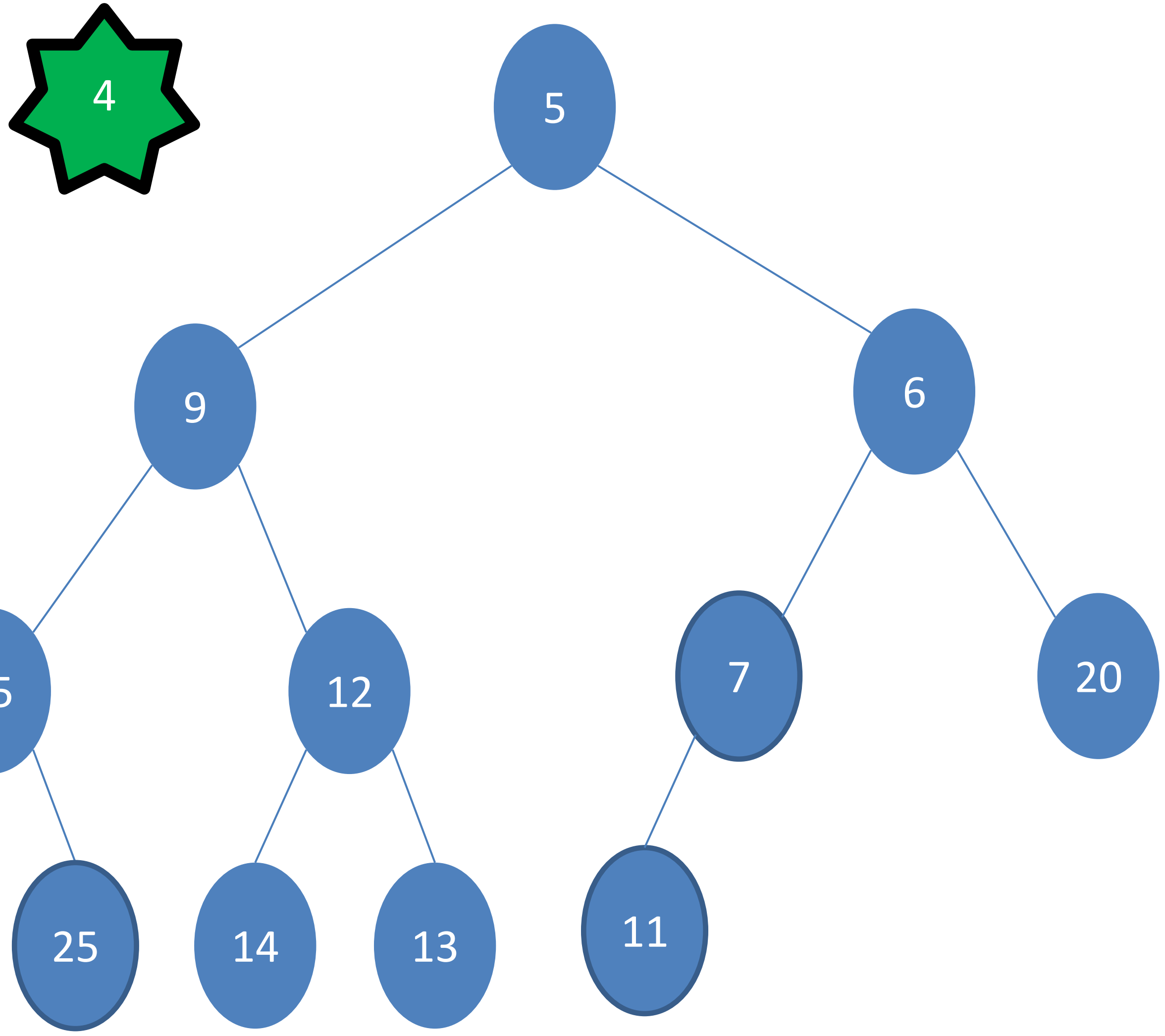
Example: removal from a heap

heap



Example: removal from a heap

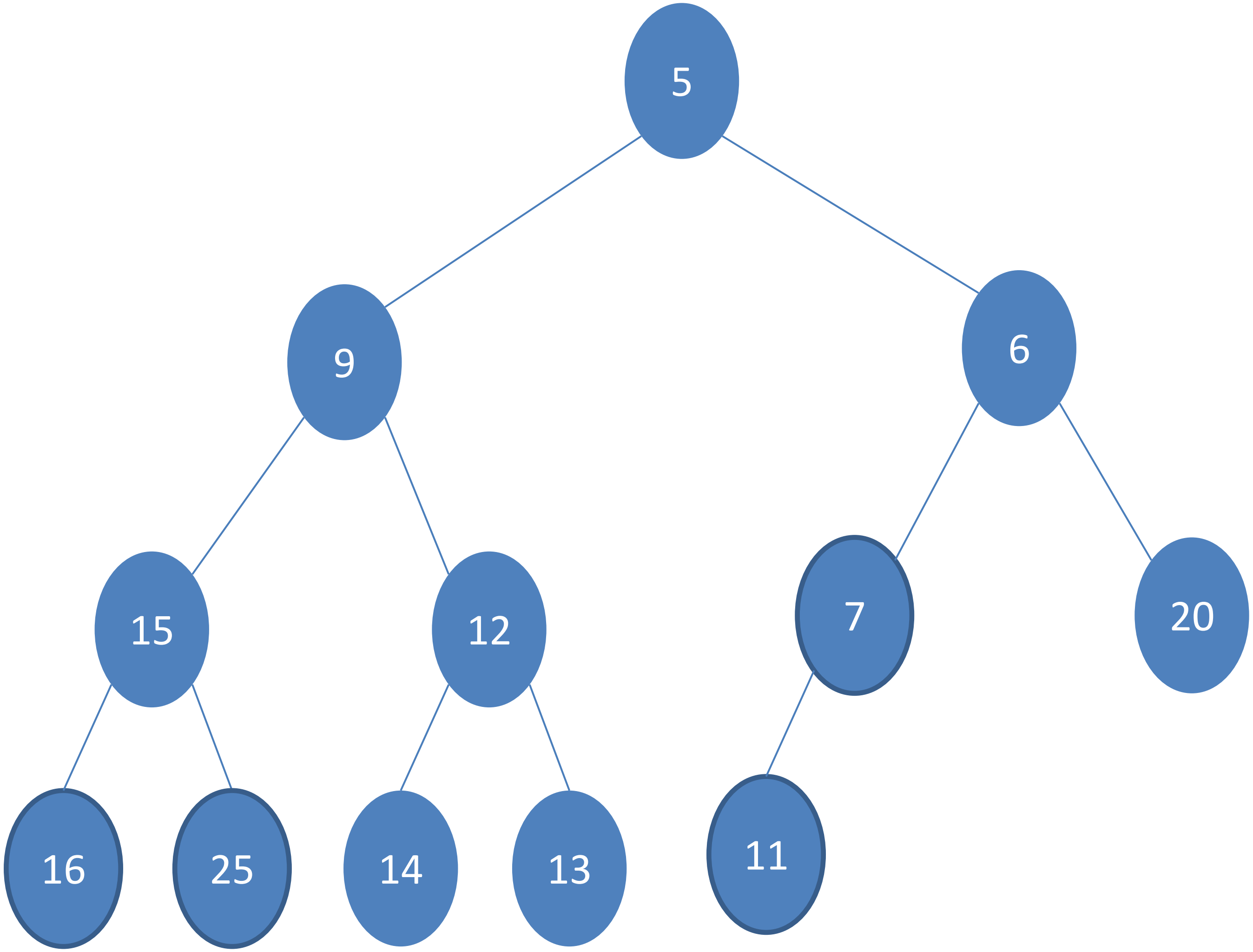
heap



Return result

Example: removal from a heap

heap



Done ☺

What we just saw

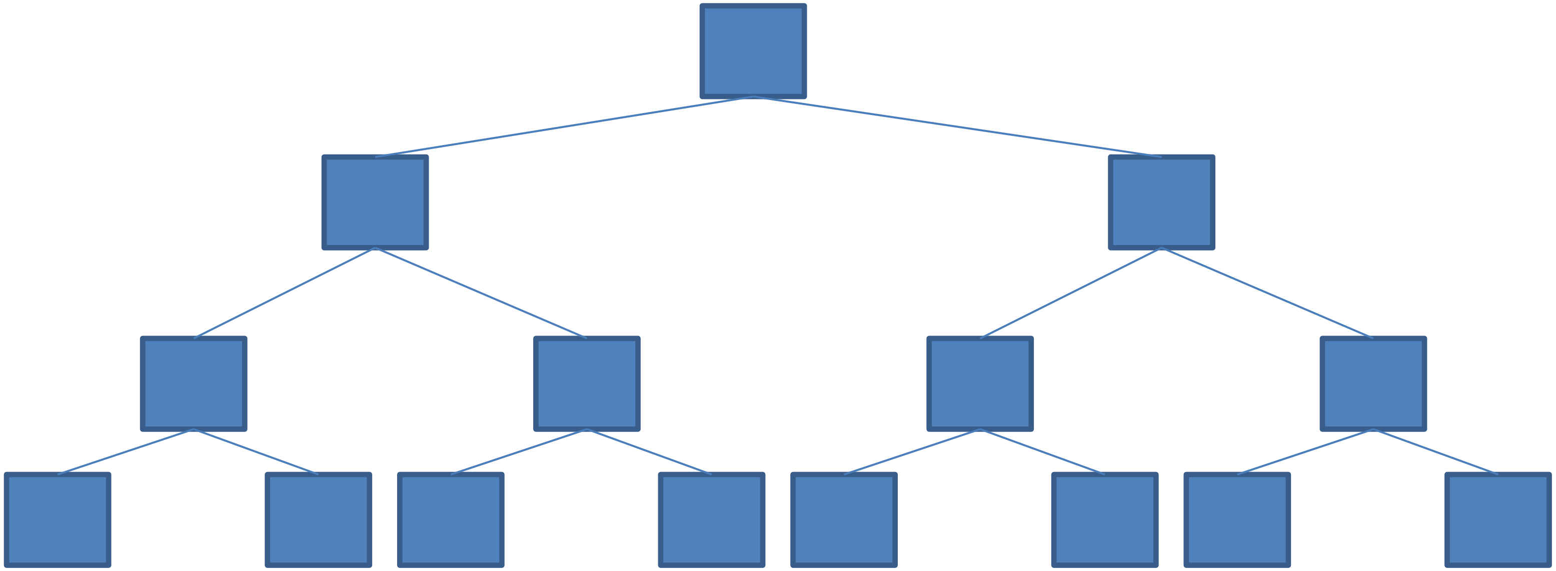
upheap bubbling: when we add to the heap

downheap bubbling: when we remove from the heap

Add and remove are O(log(n)) processes

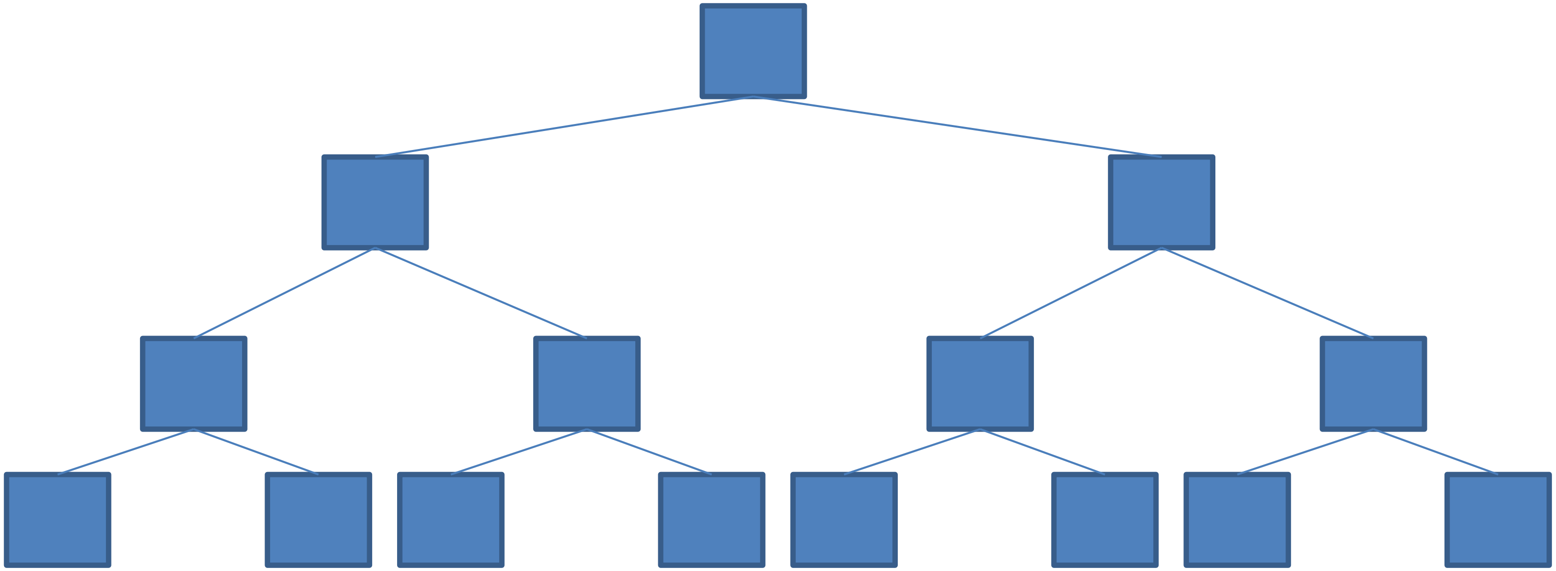
An implementation of a Heap data structure

An implementation of a Heap data structure



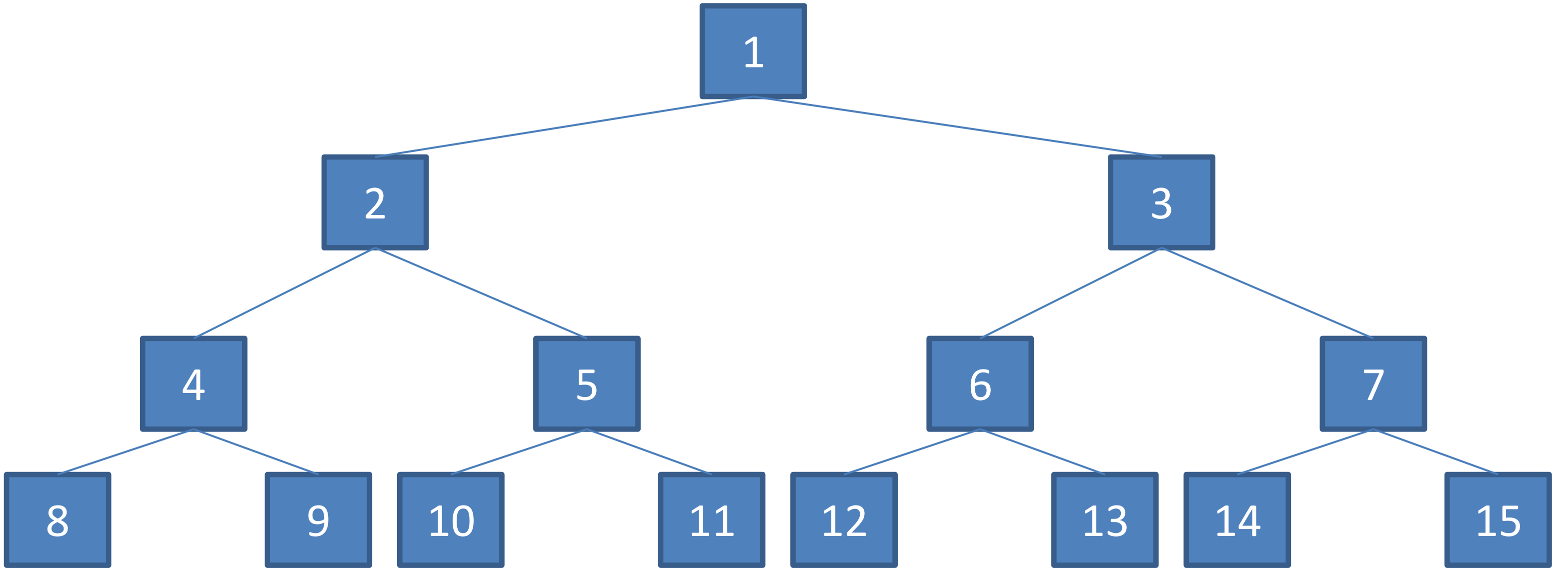
Number the vertices as follows

An implementation of a Heap data structure



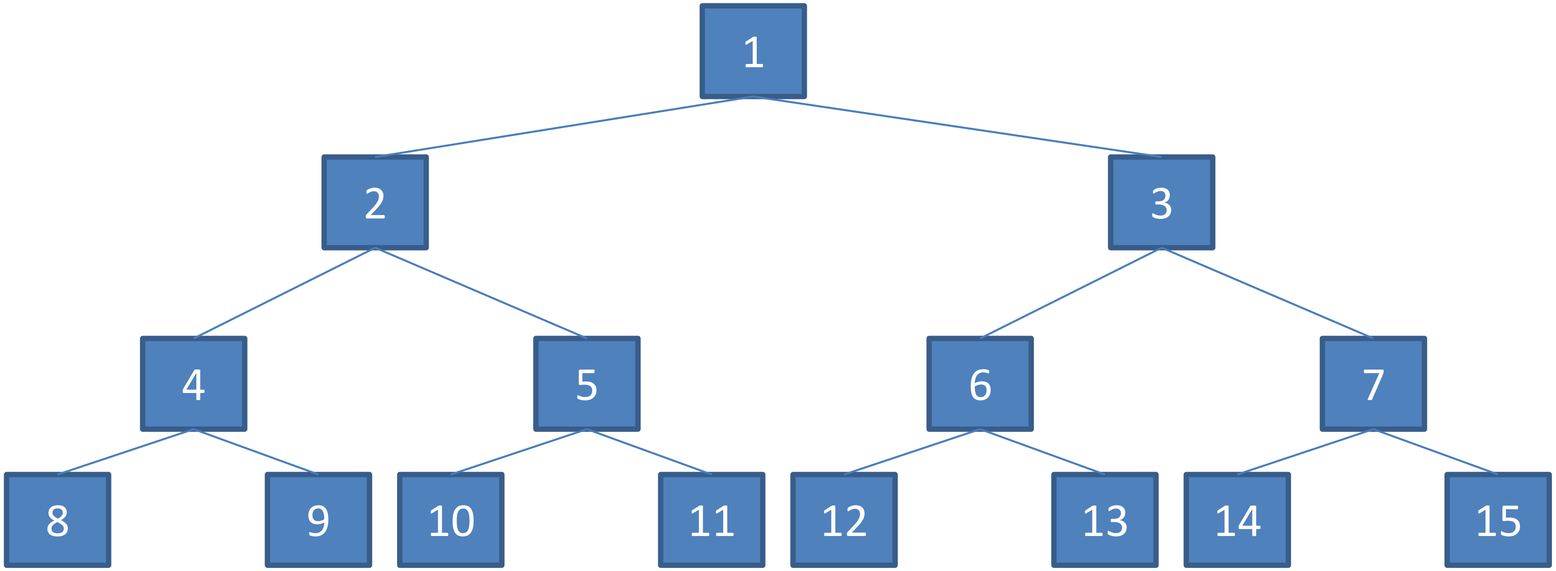
Number the vertices as follows

An implementation of a Heap data structure



Number the vertices as follows

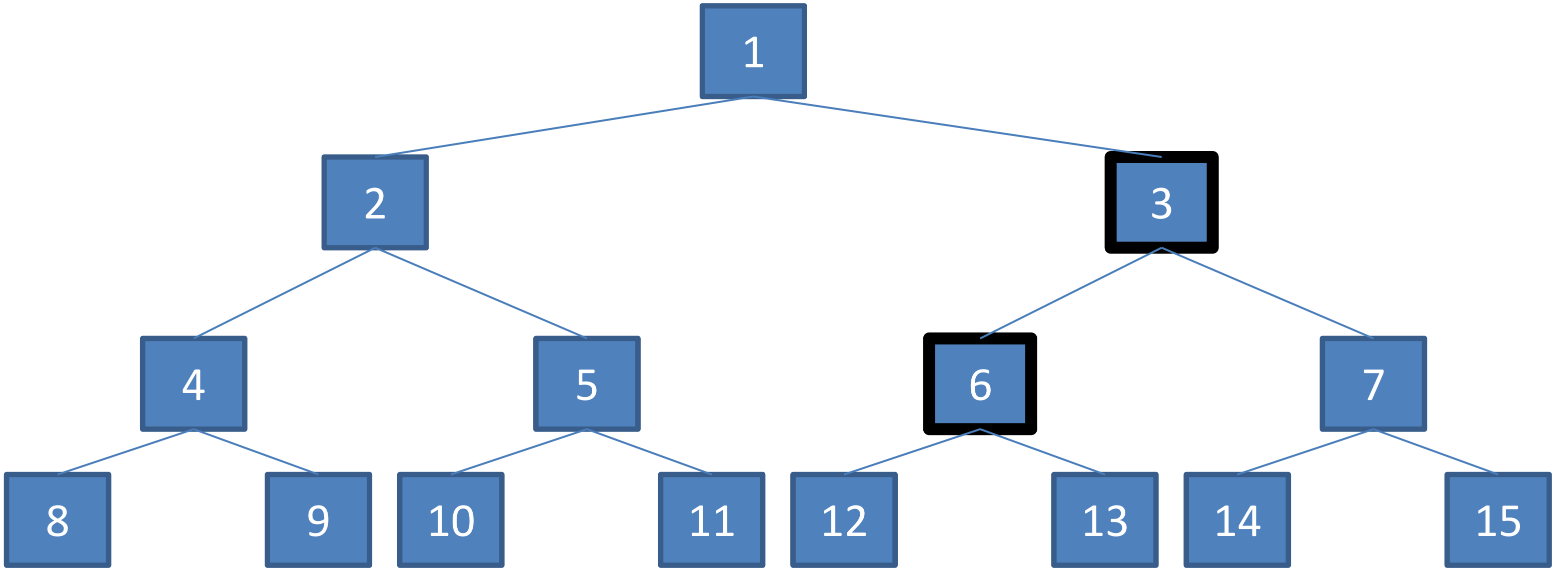
An implementation of a Heap data structure



Note: ***parent*** of node i is i/2 ....

Number the vertices as follows

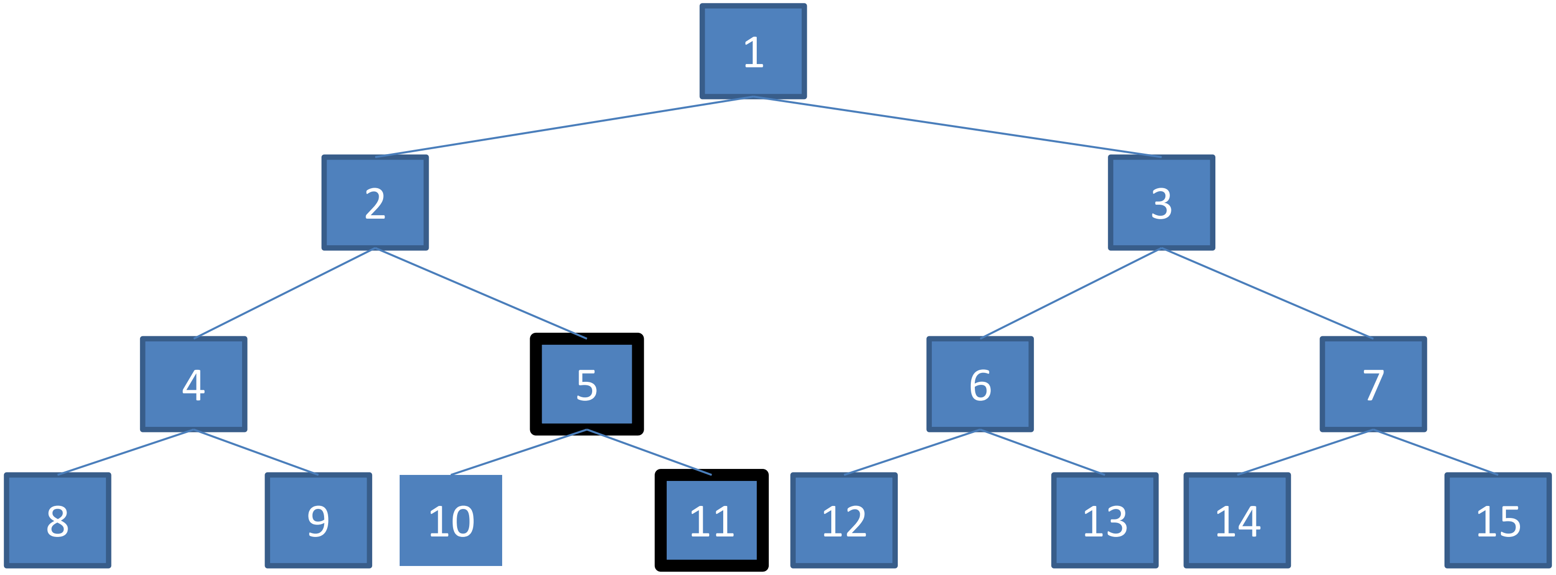
An implementation of a Heap data structure



Note: ***parent*** of node i is i/2 ....

Number the vertices as follows

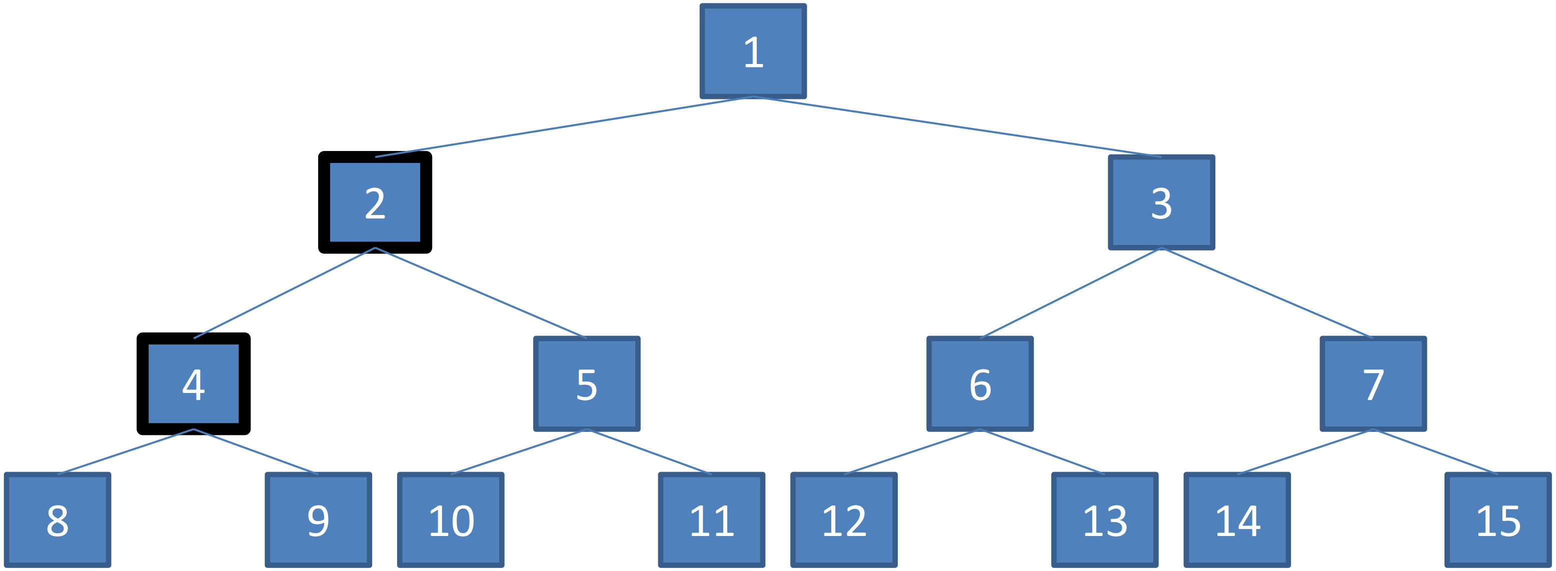
An implementation of a Heap data structure



Note: ***parent*** of node i is i/2 ....

Number the vertices as follows

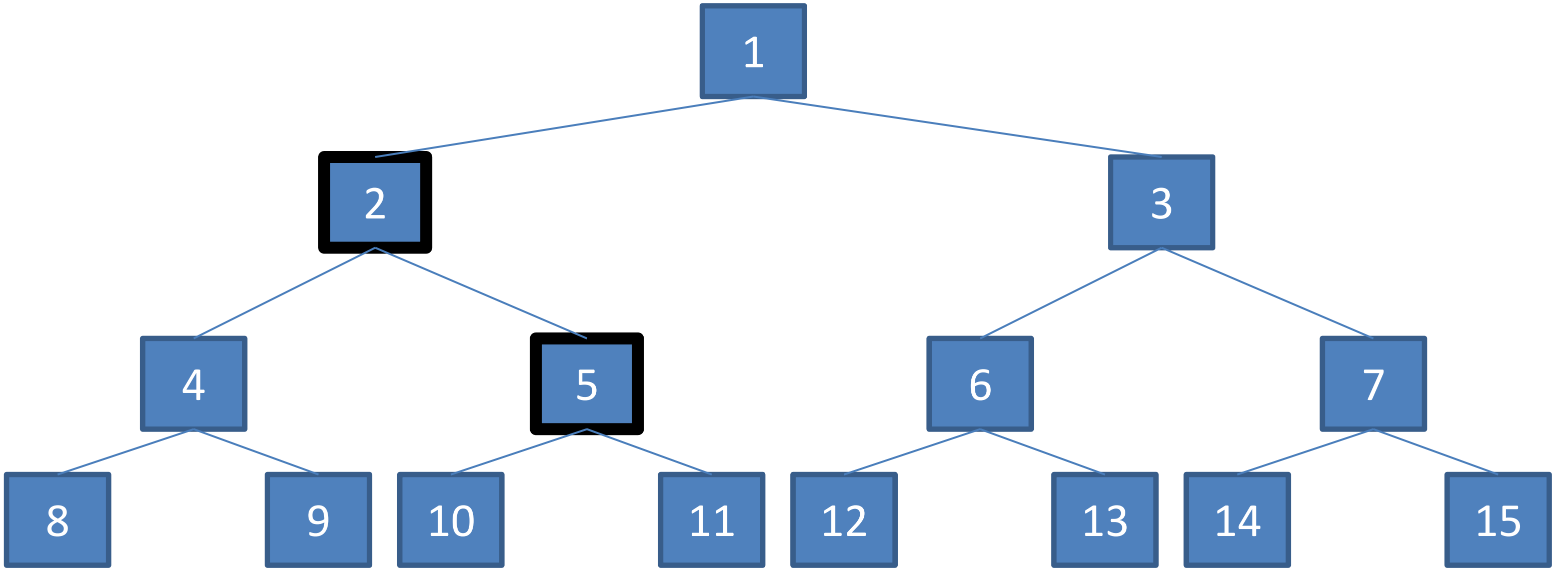
An implementation of a Heap data structure



Note: ***left child*** of i is i×2

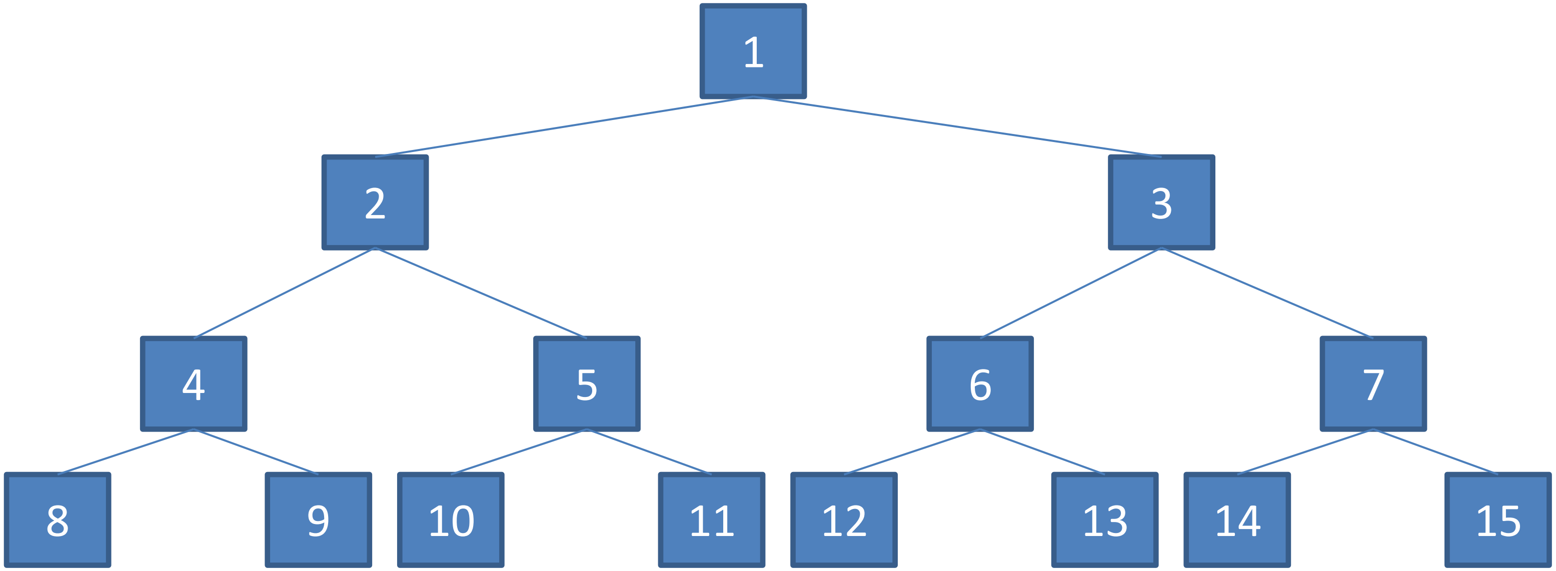
Number the vertices as follows

An implementation of a Heap data structure



Note: ***right child*** of i is (i×2) +1

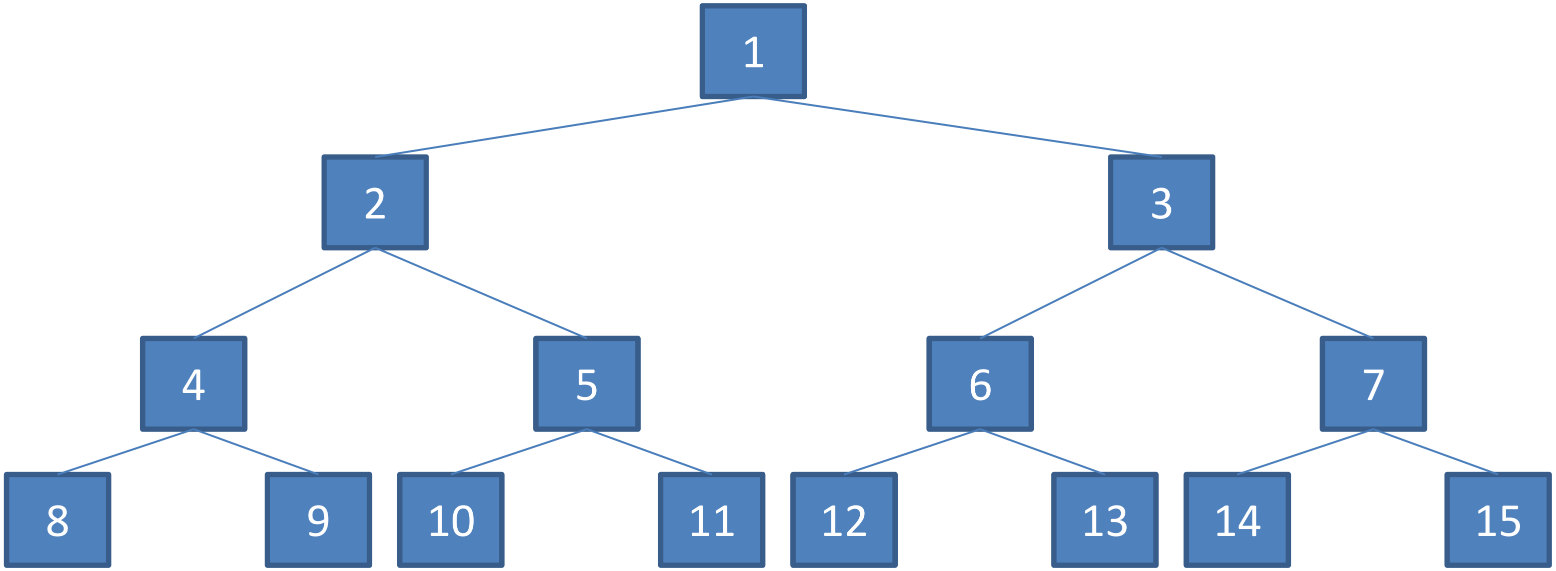
An implementation of a Heap data structure



Represent as a one *dimensional* array

Represent as a one *dimensional* array ***S***

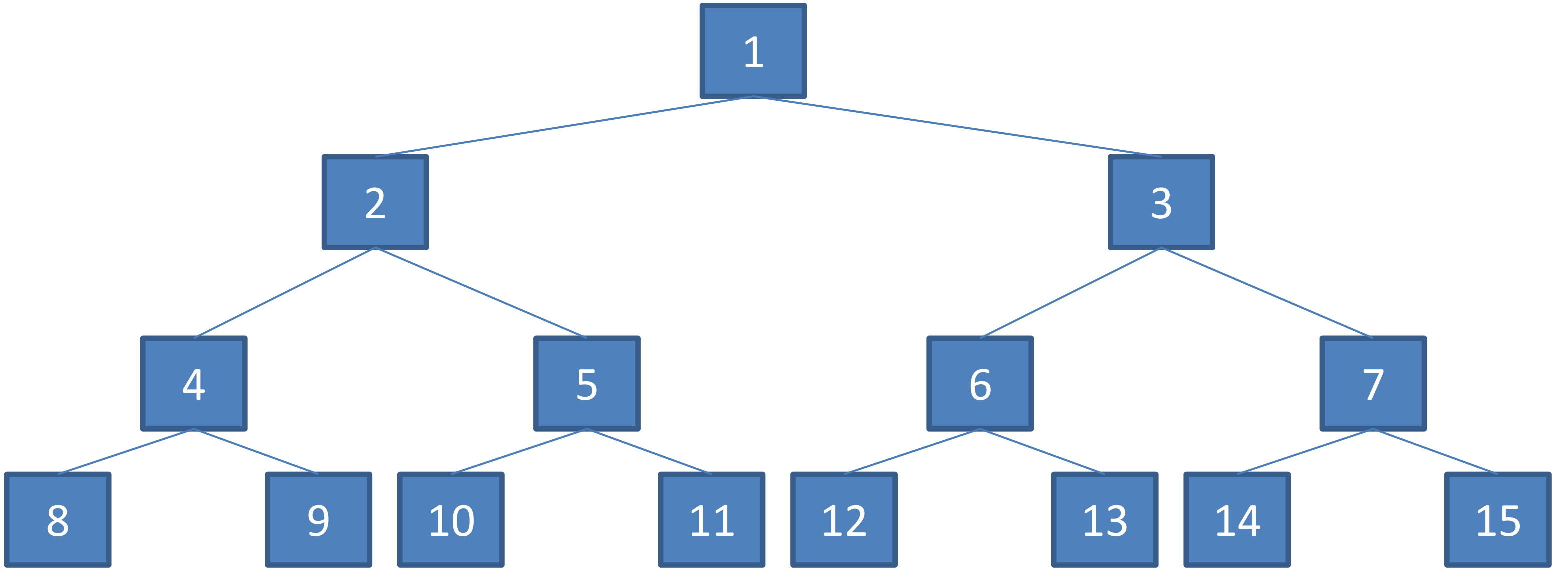
An implementation of a Heap data structure



S																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Represent as a one *dimensional* array ***S***

An implementation of a Heap data structure

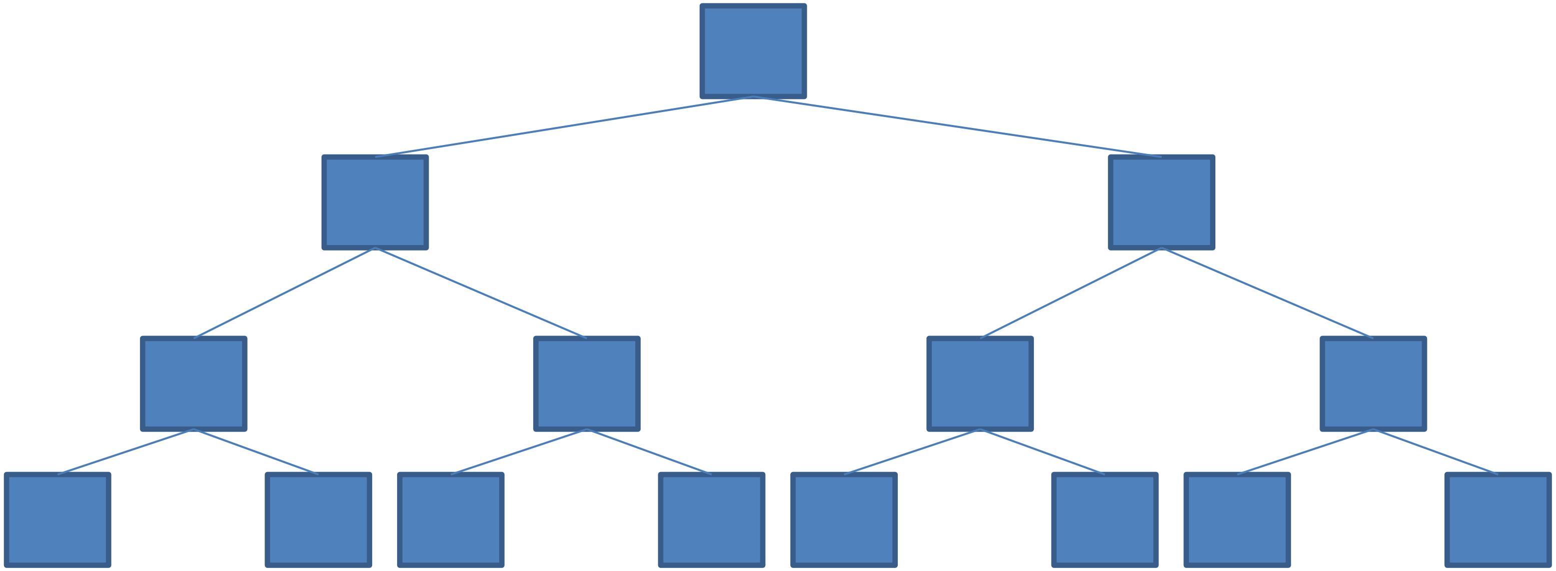


S	**															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

To simplify implementation we ***do not*** use S[0]

Represent as a one *dimensional* array ***S***

An implementation of a Heap data structure

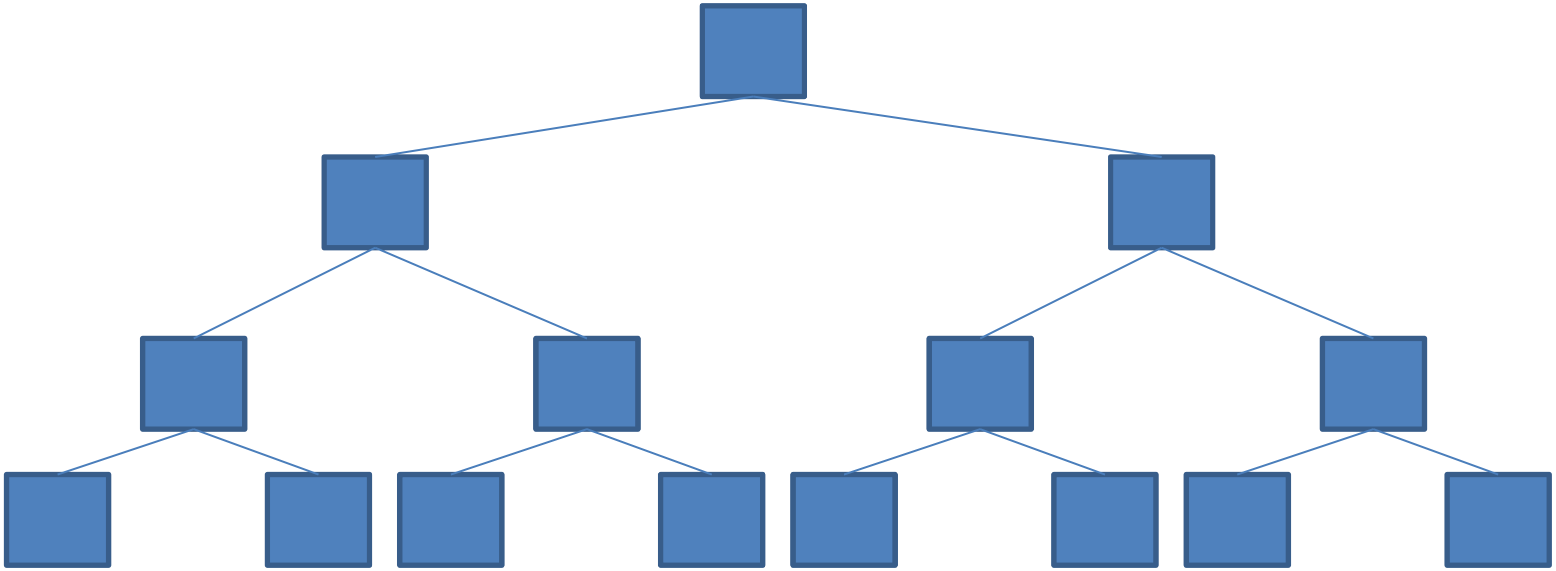


S	**															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Require two integer variables, ***last*** and ***capacity*** where last is initially 0
In our example ***capacity*** is 15

Represent as a one *dimensional* array ***S***

An implementation of a Heap data structure



S	**															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: 0

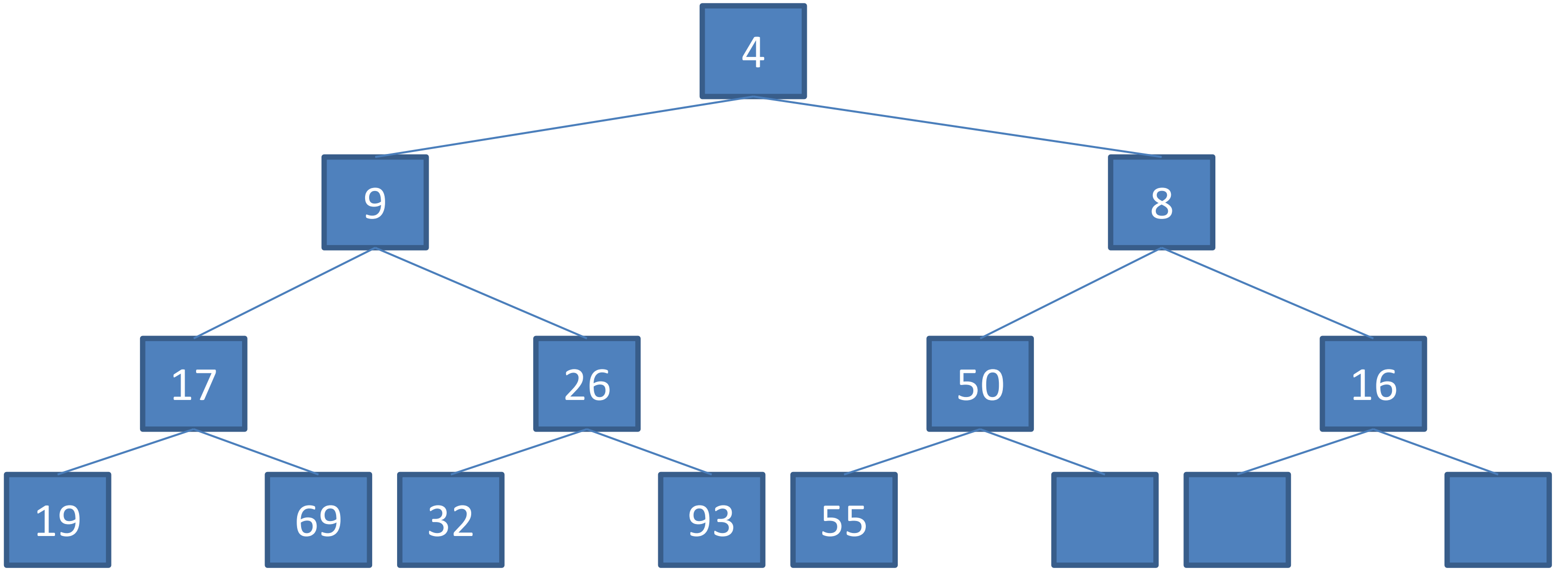
capacity: 15

An implementation of a Heap data structure

Consider the following heap H

Consider the following heap H

An implementation of a Heap data structure

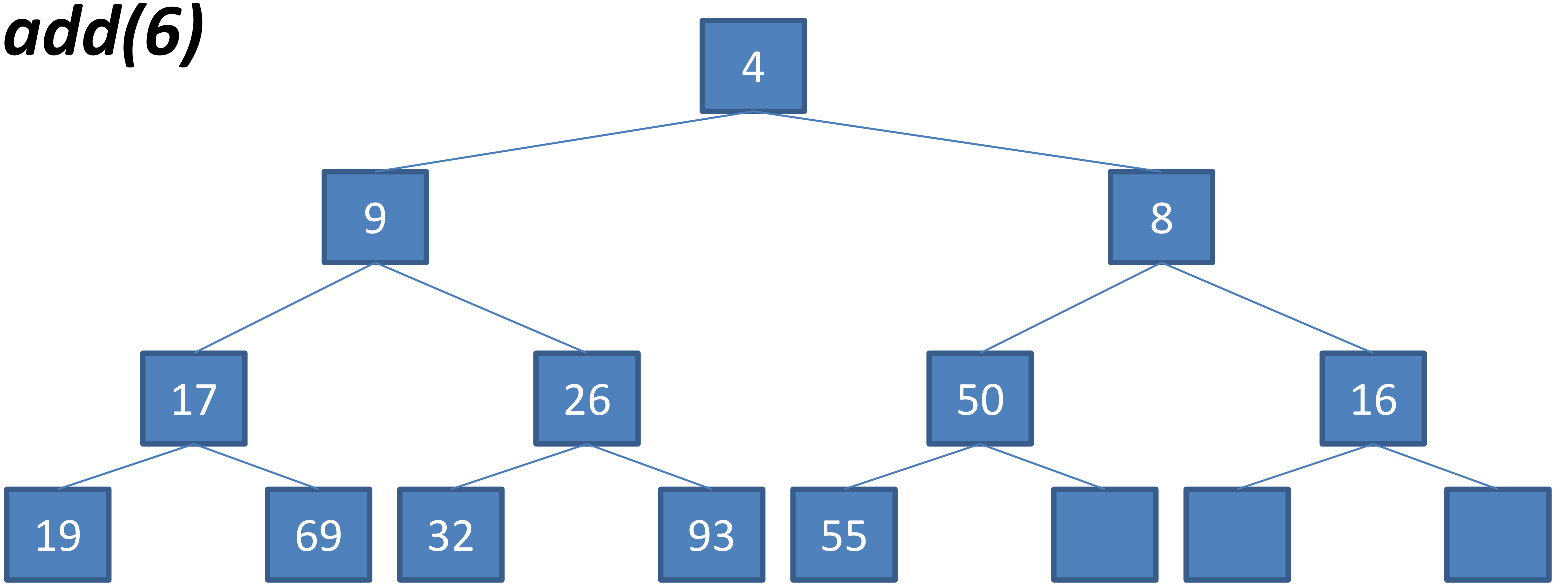


S	******	**4**	**9**	**8**	**17**	**26**	**50**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: 12

capacity: 15

H.add(6)



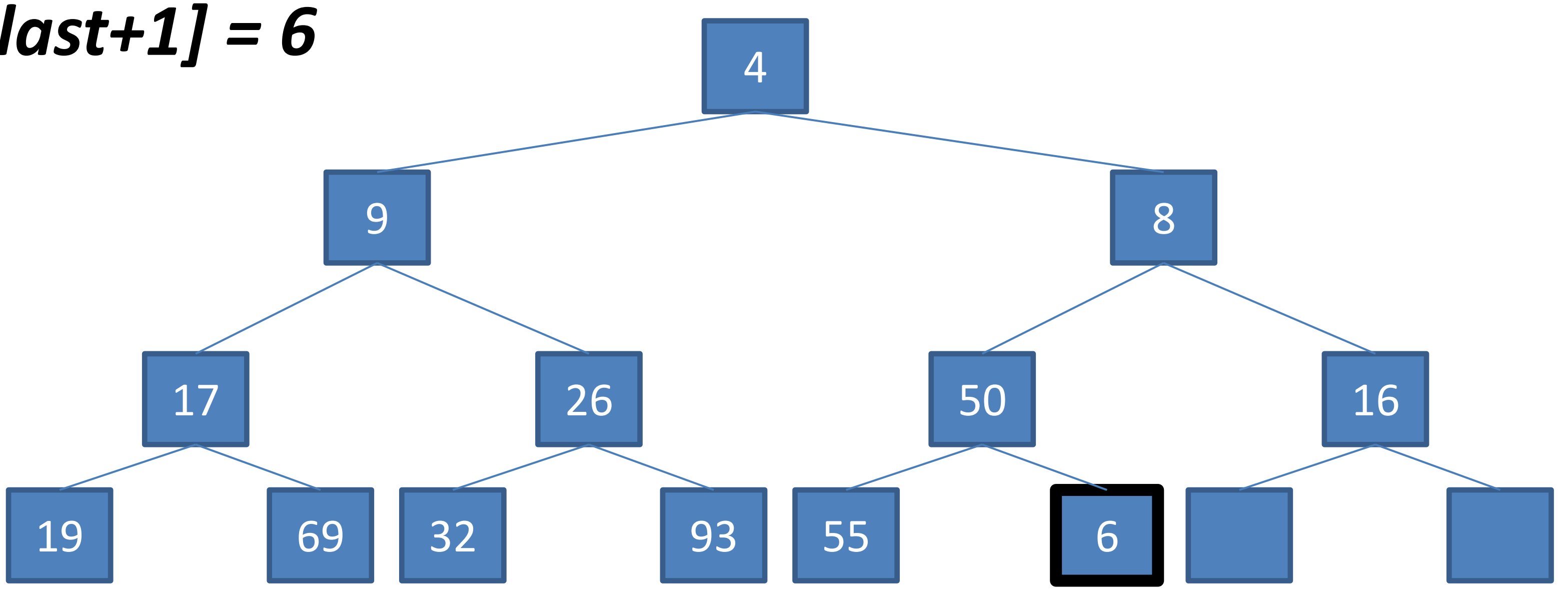
S	******	**4**	**9**	**8**	**17**	**26**	**50**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: 12

capacity: 15

An implementation of a Heap data structure

S[last+1] = 6



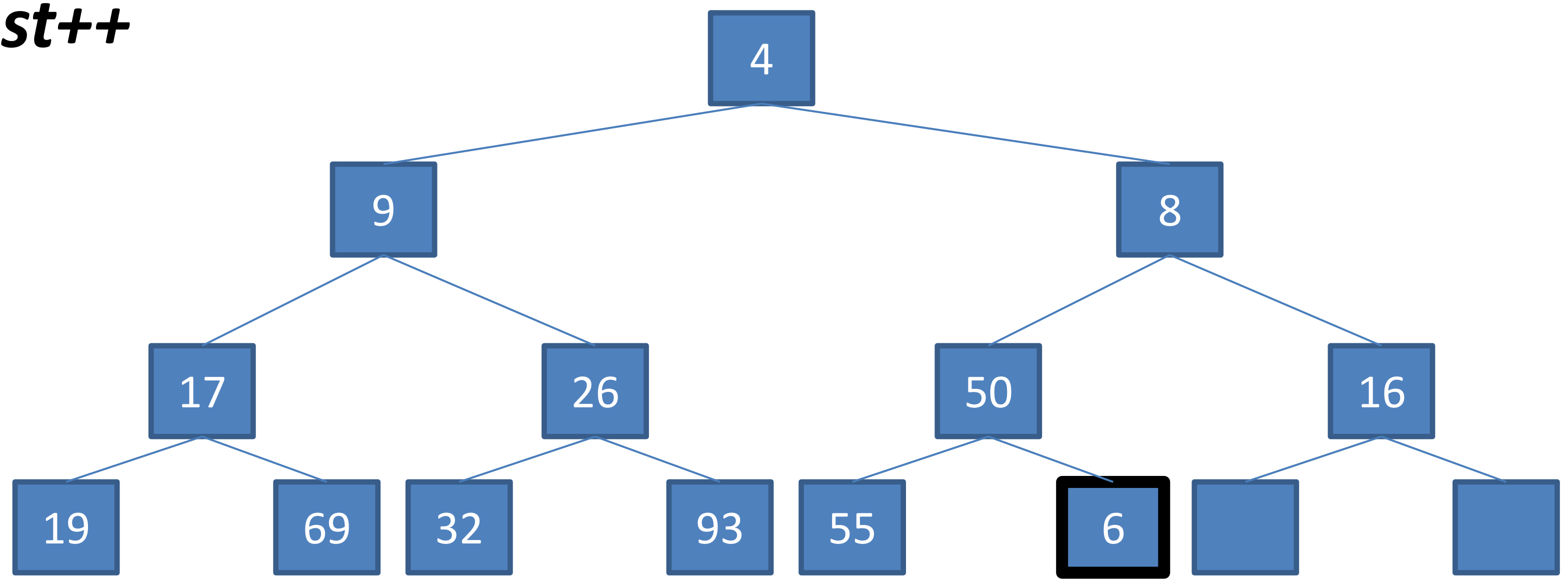
S	**	4	9	8	17	26	50	16	19	69	32	93	55	***6***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: 12

capacity: 15

An implementation of a Heap data structure

last++



S

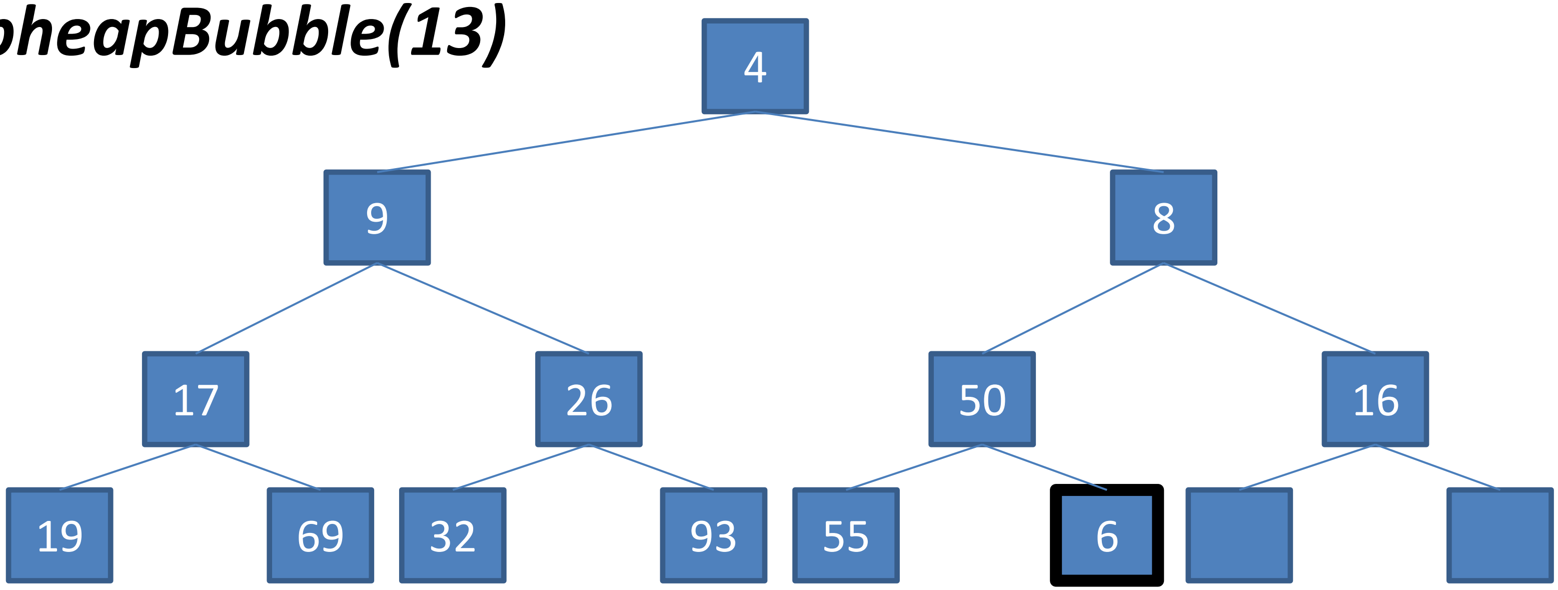
**	4	9	8	17	26	50	16	19	69	32	93	55	***6***		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

An implementation of a Heap data structure

upheapBubble(13)

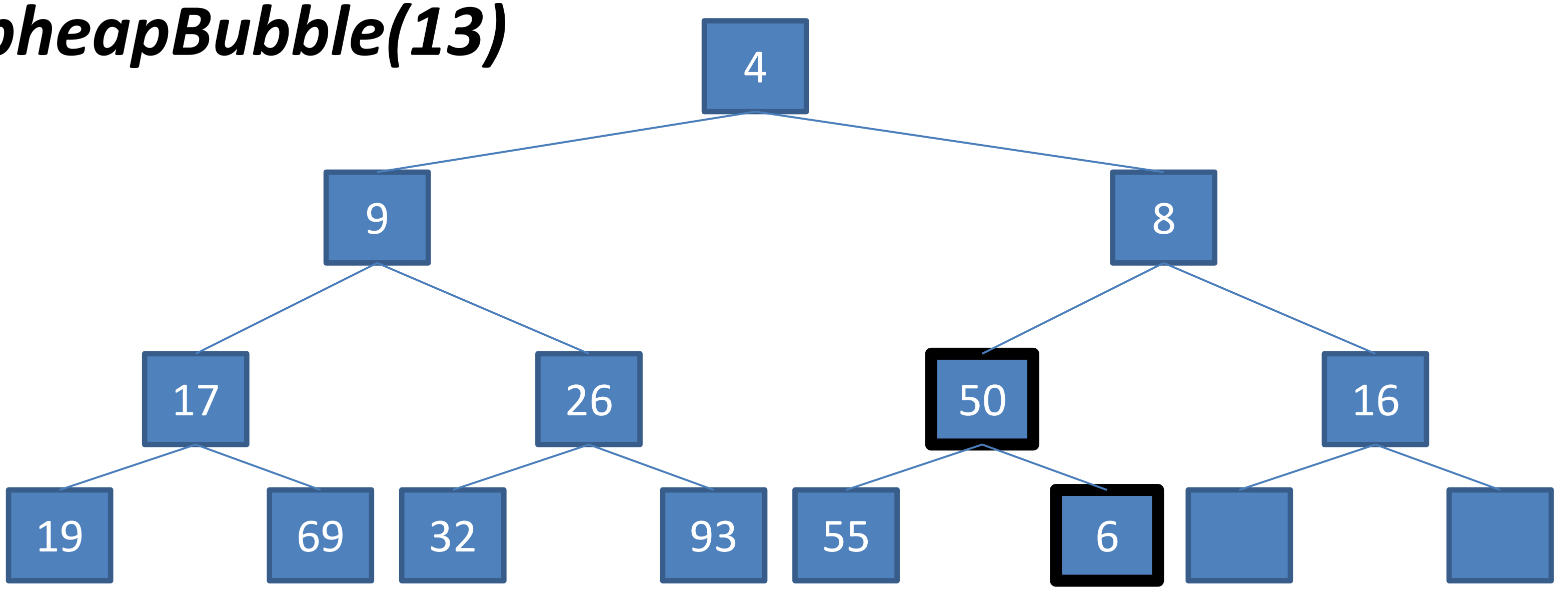


S	******	**4**	**9**	**8**	**17**	**26**	**50**	**16**	**19**	**69**	**32**	**93**	**55**	***6***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(13)



S	**	**4**	**9**	**8**	**17**	**26**	***50***	**16**	**19**	**69**	**32**	**93**	**55**	***6***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

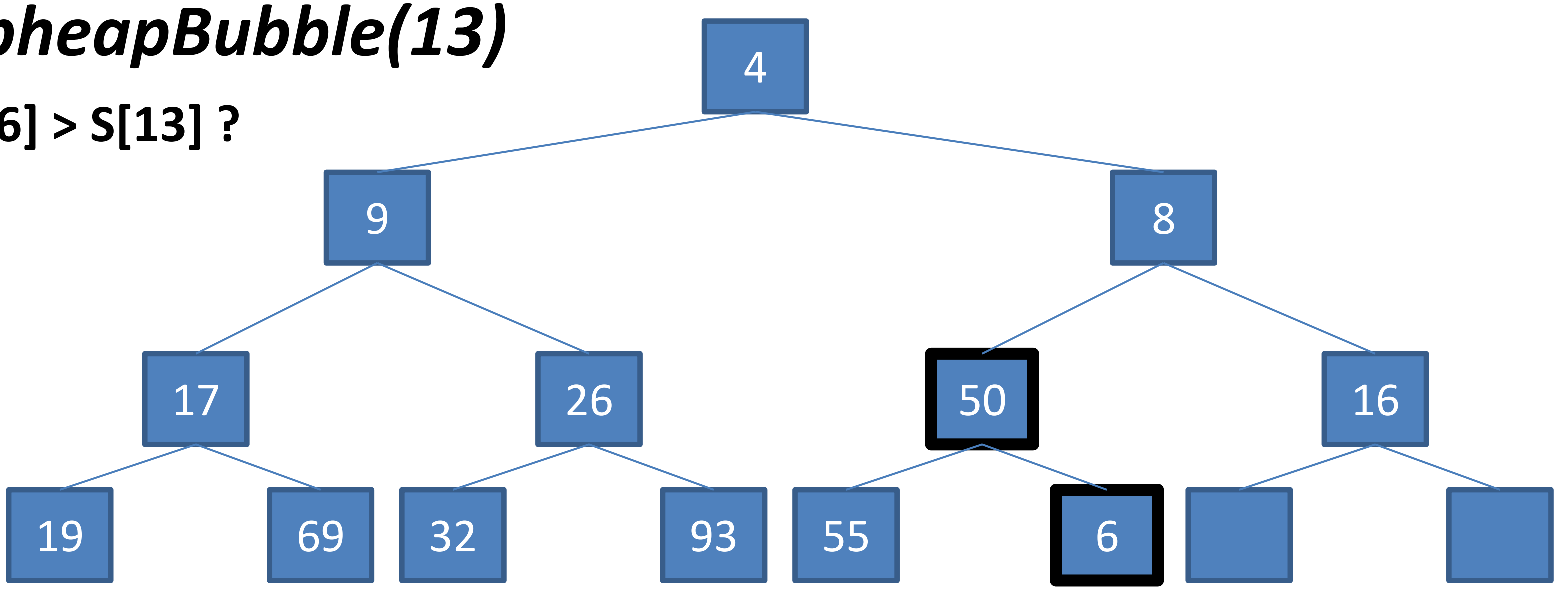
last: ***13***

capacity: 15

An implementation of a Heap data structure

upheapBubble(13)

S[6] > S[13] ?



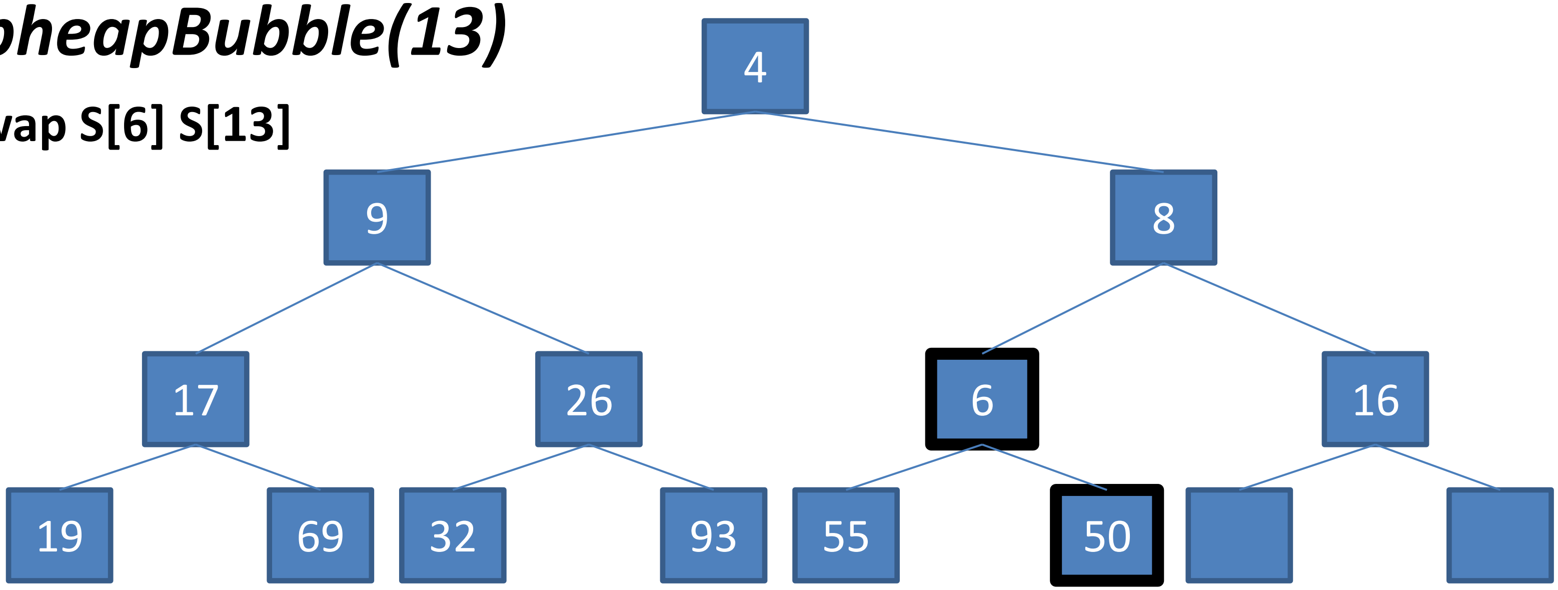
S	**	4	9	8	17	26	***50***	16	19	69	32	93	55	***6***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(13)

swap S[6] S[13]

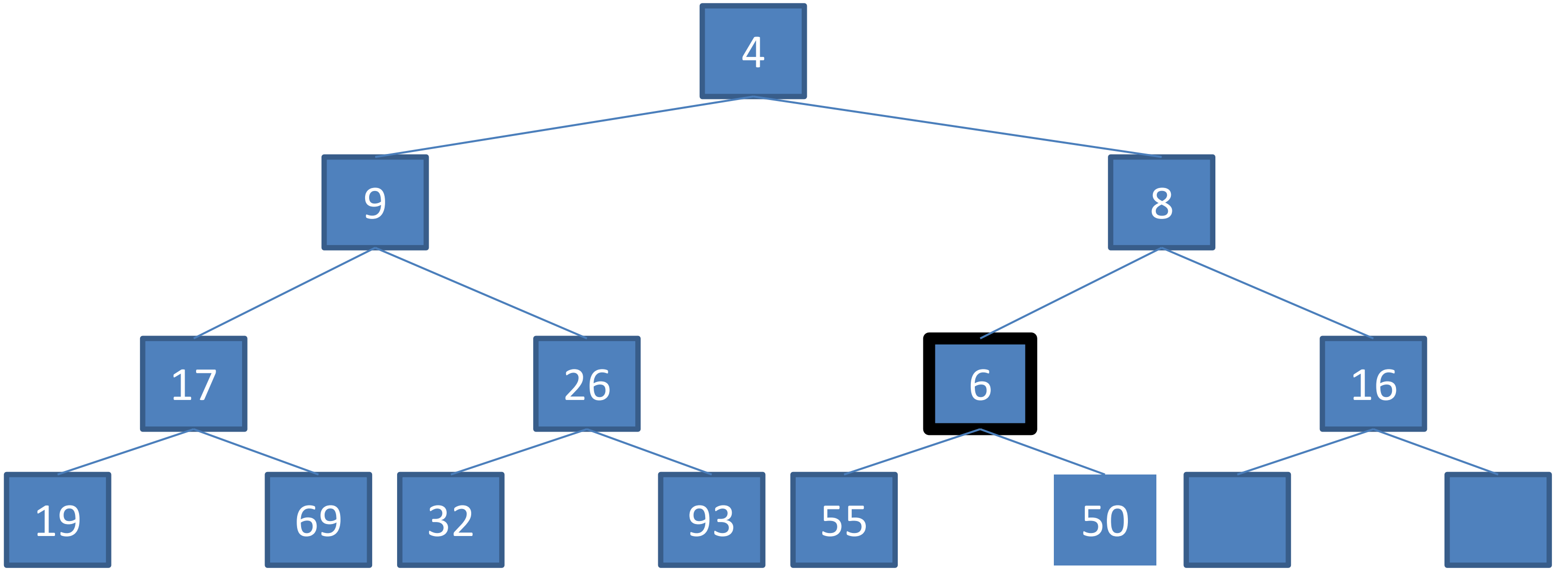


S	**	4	9	8	17	26	***6***	16	19	69	32	93	55	***50***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
							↑							↑		

last: ***13***

capacity: 15

An implementation of a Heap data structure

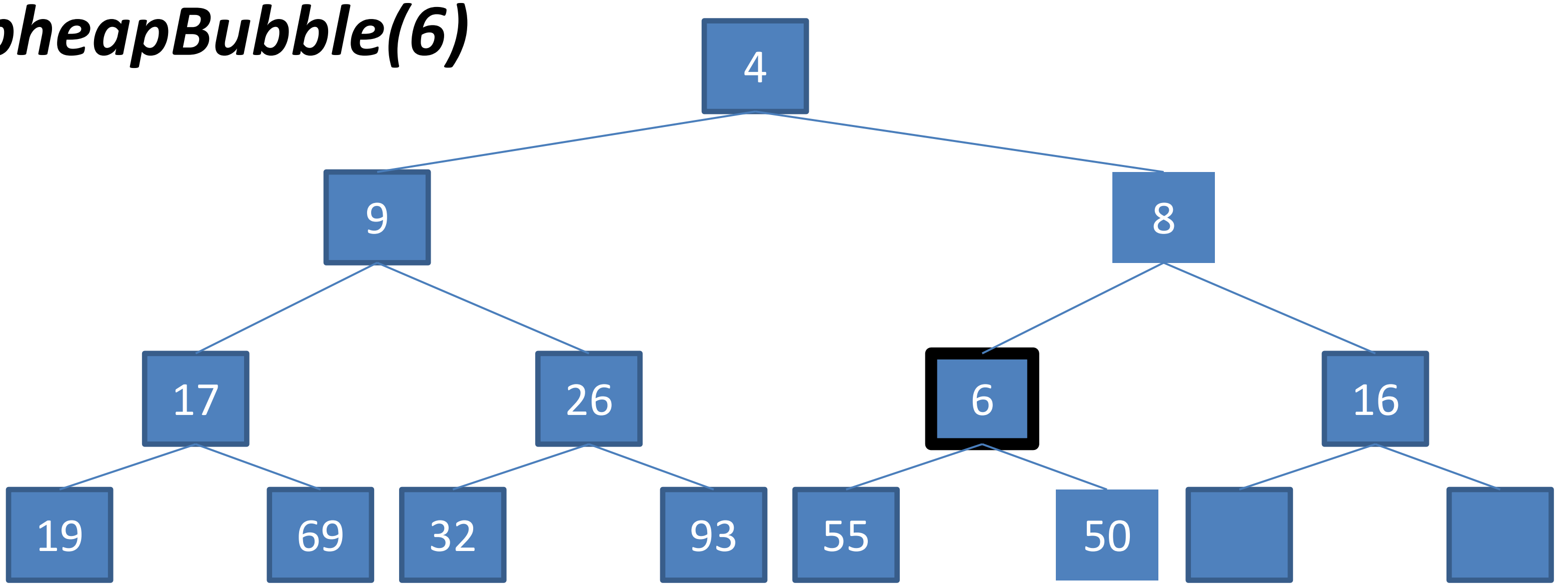


S	**	**4**	**9**	**8**	**17**	**26**	***6***	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(6)

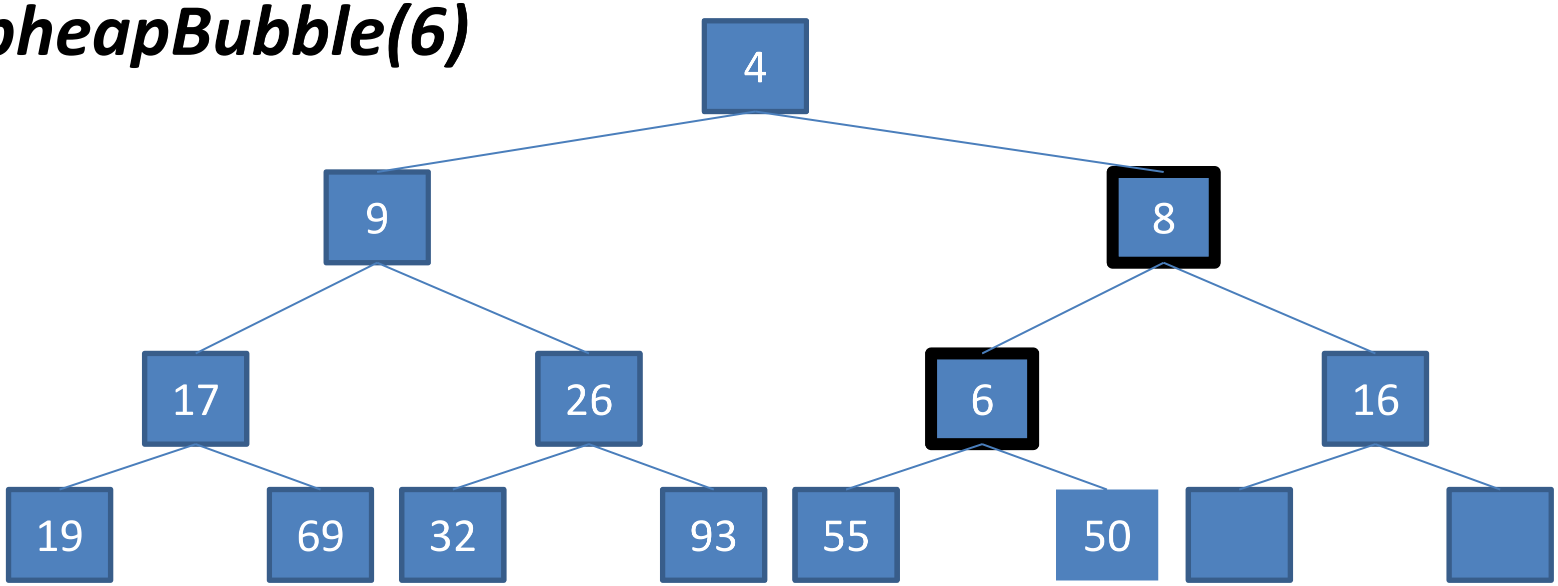


S	**	4	9	8	17	26	***6***	16	19	69	32	93	55	50		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(6)



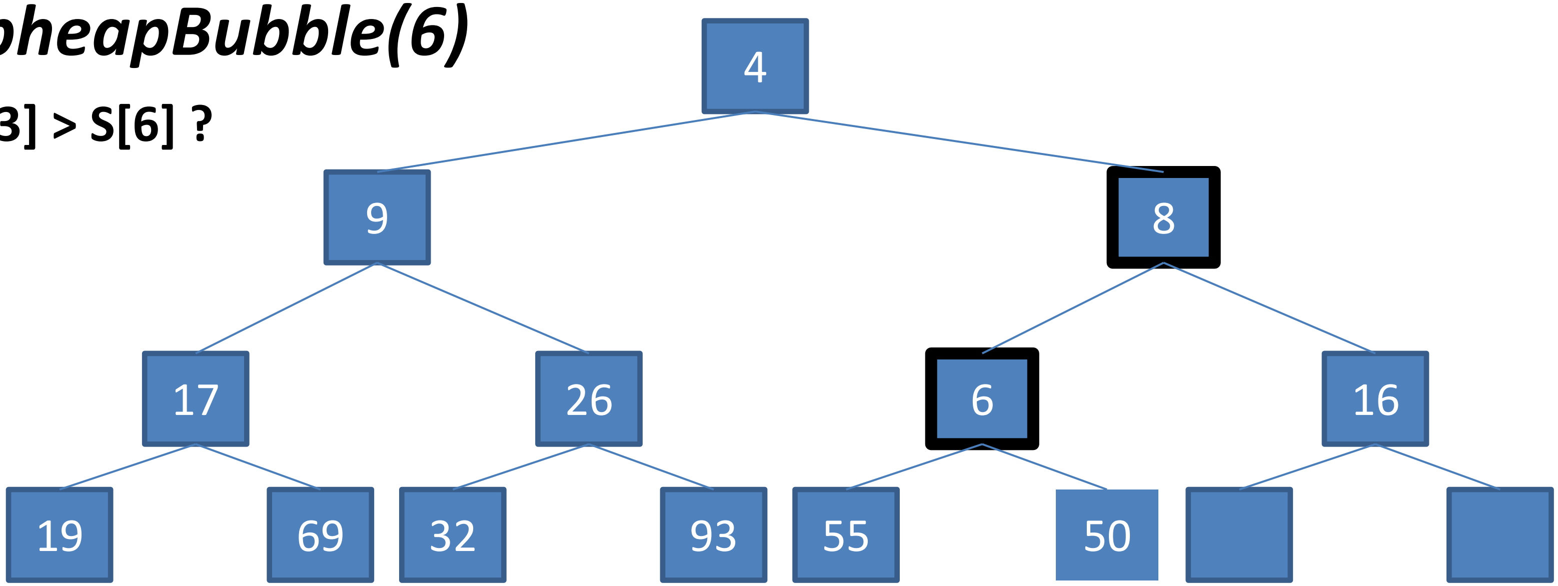
S	******	**4**	**9**	***8***	**17**	**26**	***6***	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(6)

S[3] > S[6] ?



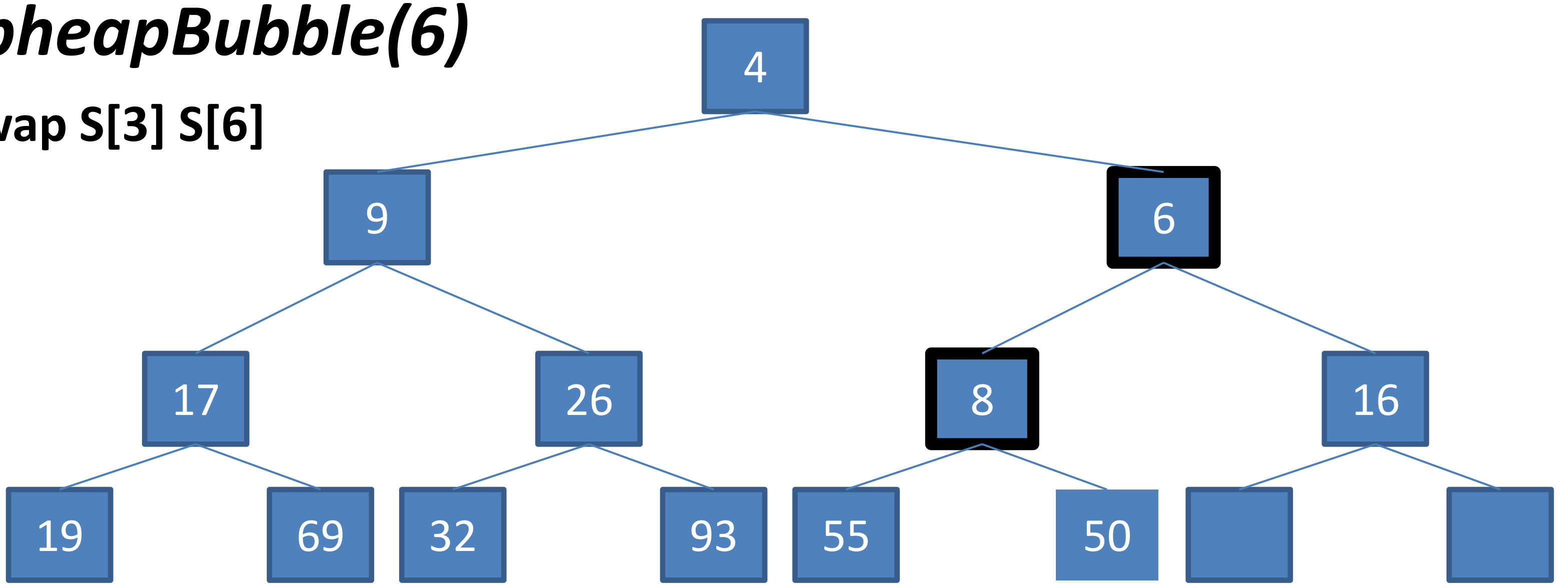
S	**	**4**	**9**	***8***	**17**	**26**	***6***	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

upheapBubble(6)

swap S[3] S[6]

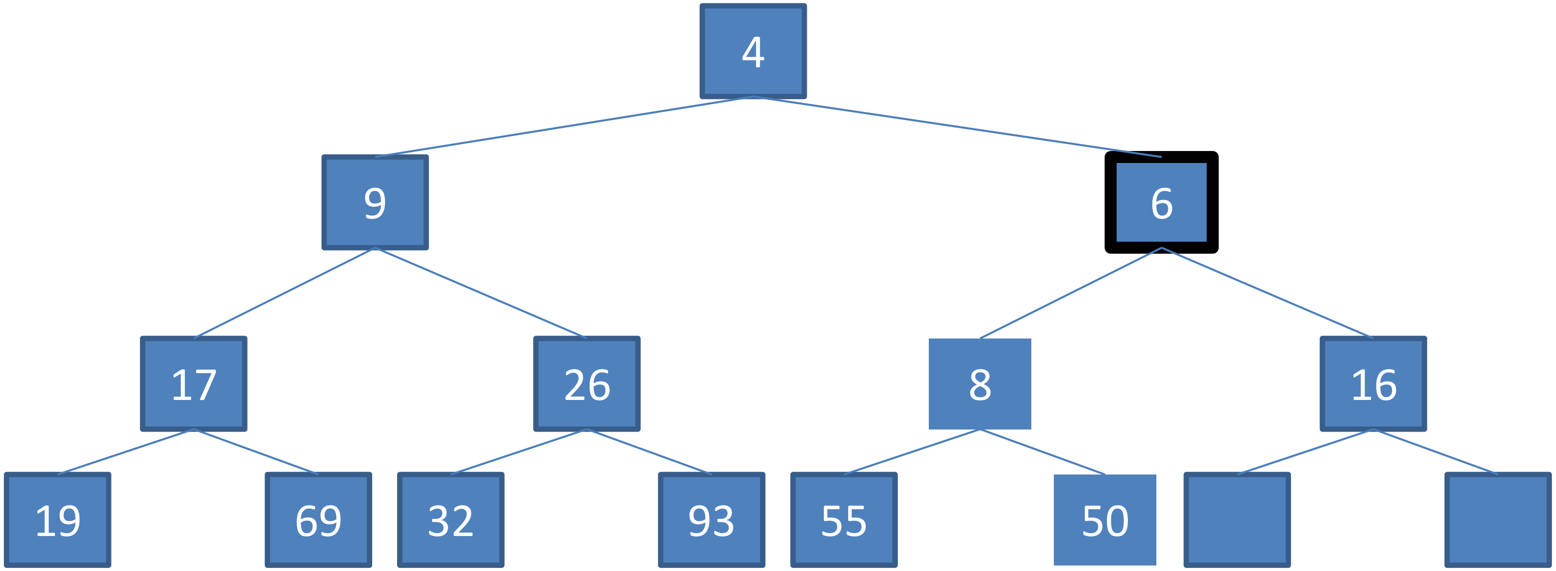


S																
	**	4	9	***6***	17	26	***8***	16	19	69	32	93	55	50		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

An implementation of a Heap data structure



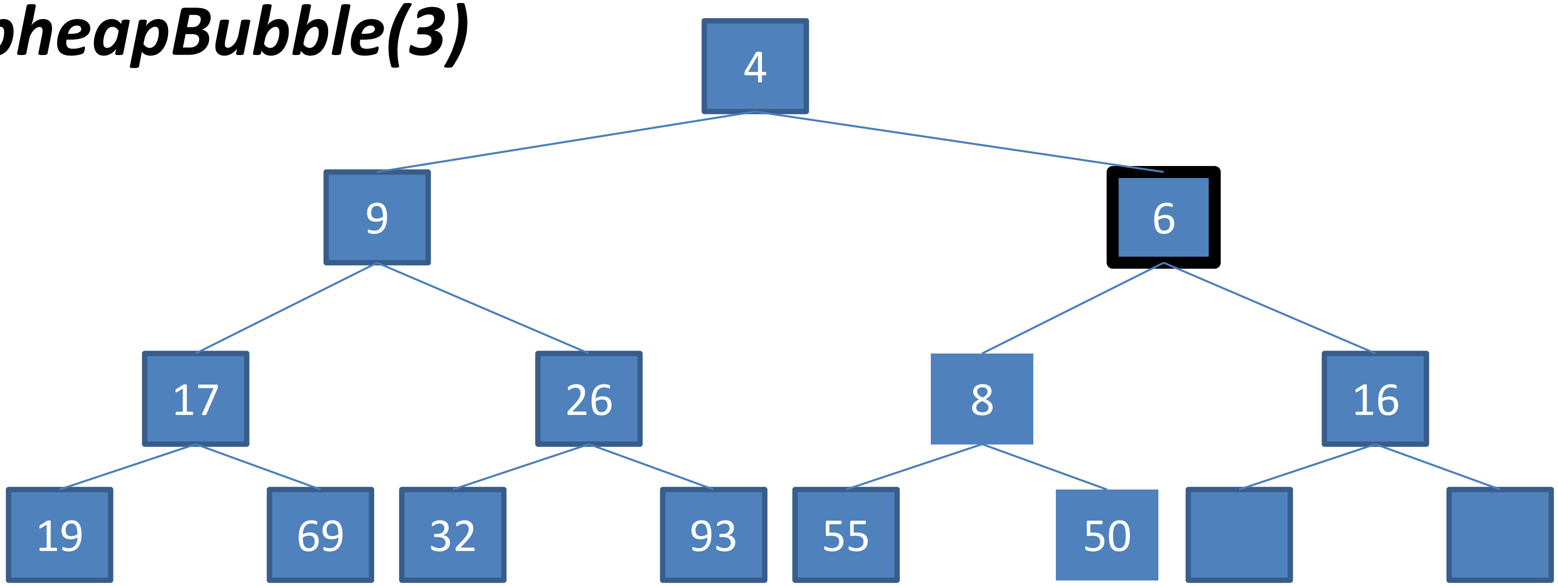
S	**	4	9	*6*	17	26	8	16	19	69	32	93	55	50		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

An implementation of a Heap data structure

upheapBubble(3)



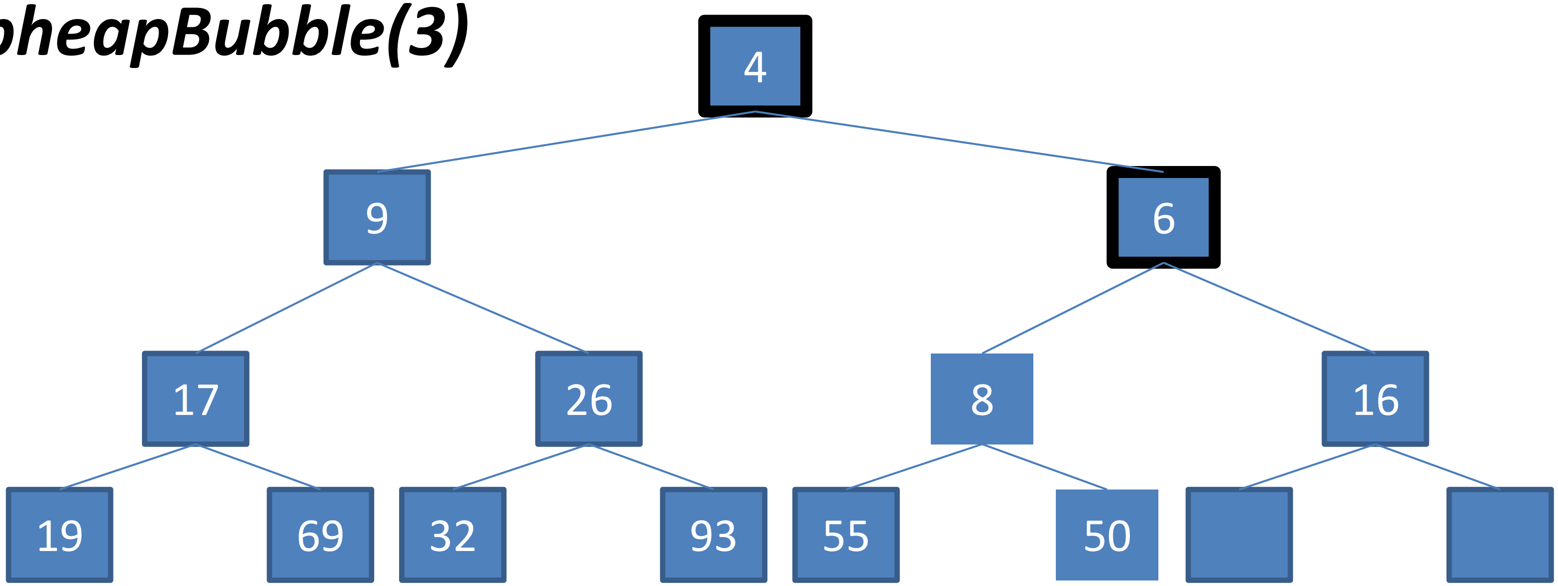
S	**	4	9	*6*	17	26	8	16	19	69	32	93	55	50		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

An implementation of a Heap data structure

upheapBubble(3)



S	**	***4***	9	***6***	17	26	8	16	19	69	32	93	55	50		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

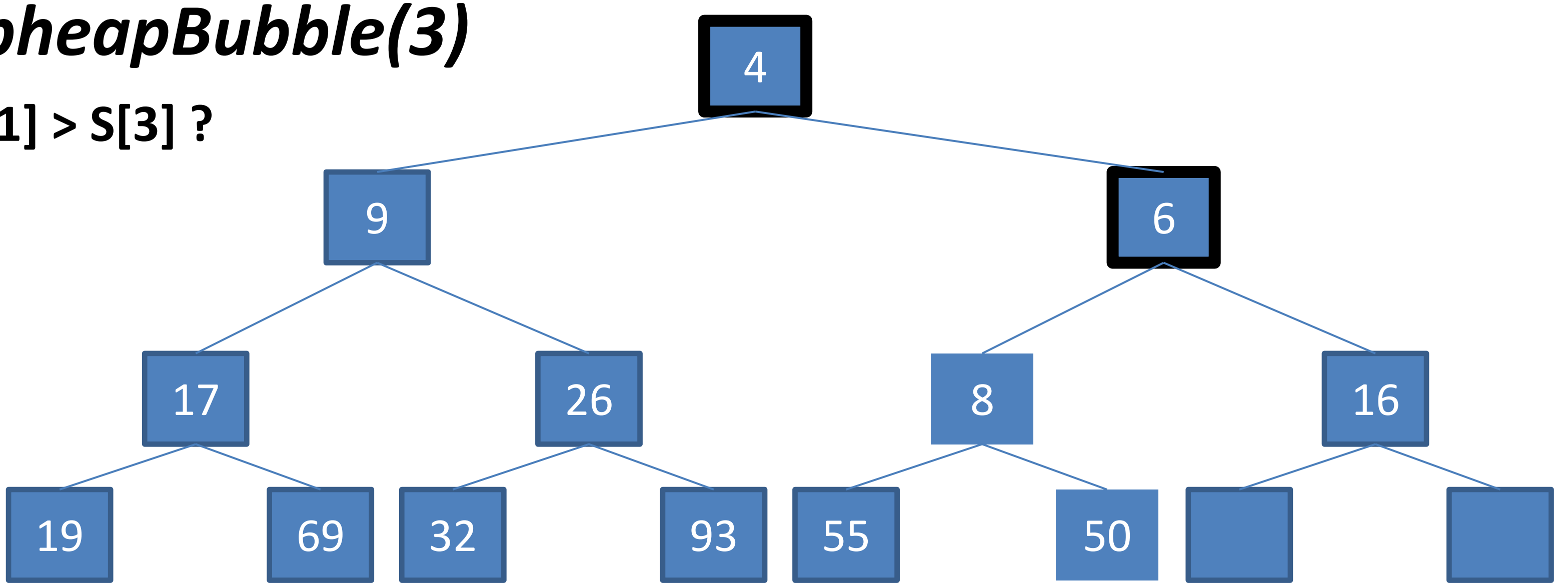
last: ***13***

capacity: 15

An implementation of a Heap data structure

upheapBubble(3)

S[1] > S[3] ?



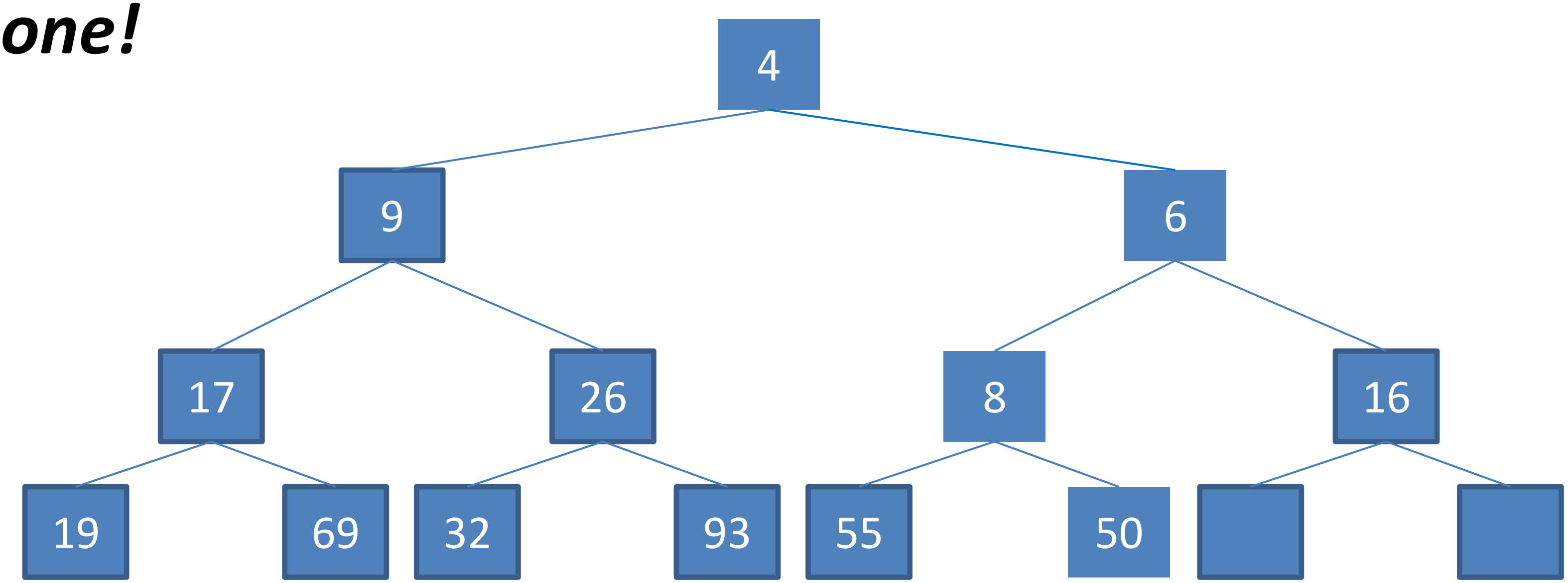
S	**	***4***	**9**	***6***	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***13***

capacity: 15

An implementation of a Heap data structure

Done!



S

**	4	9	6	17	26	8	16	19	69	32	93	55	50		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

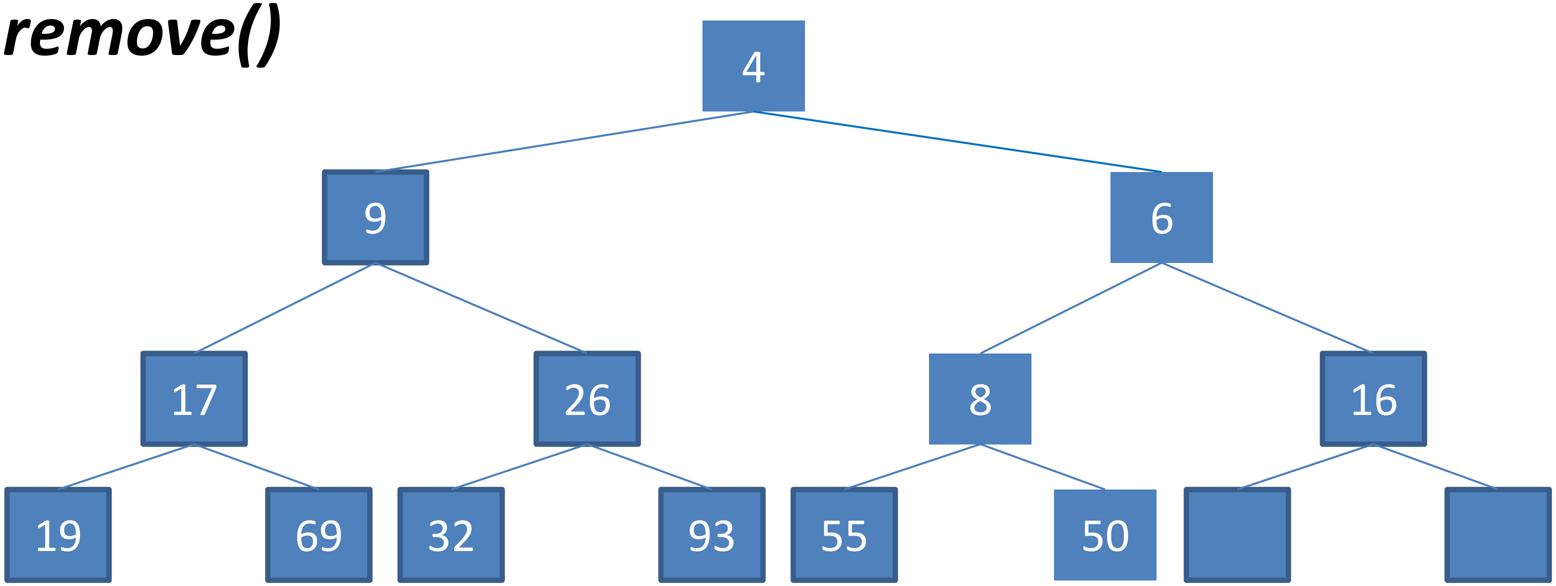
last: ***13***

capacity: 15

Removal from the heap H

An implementation of a Heap data structure

H.remove()



S	**	**4**	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

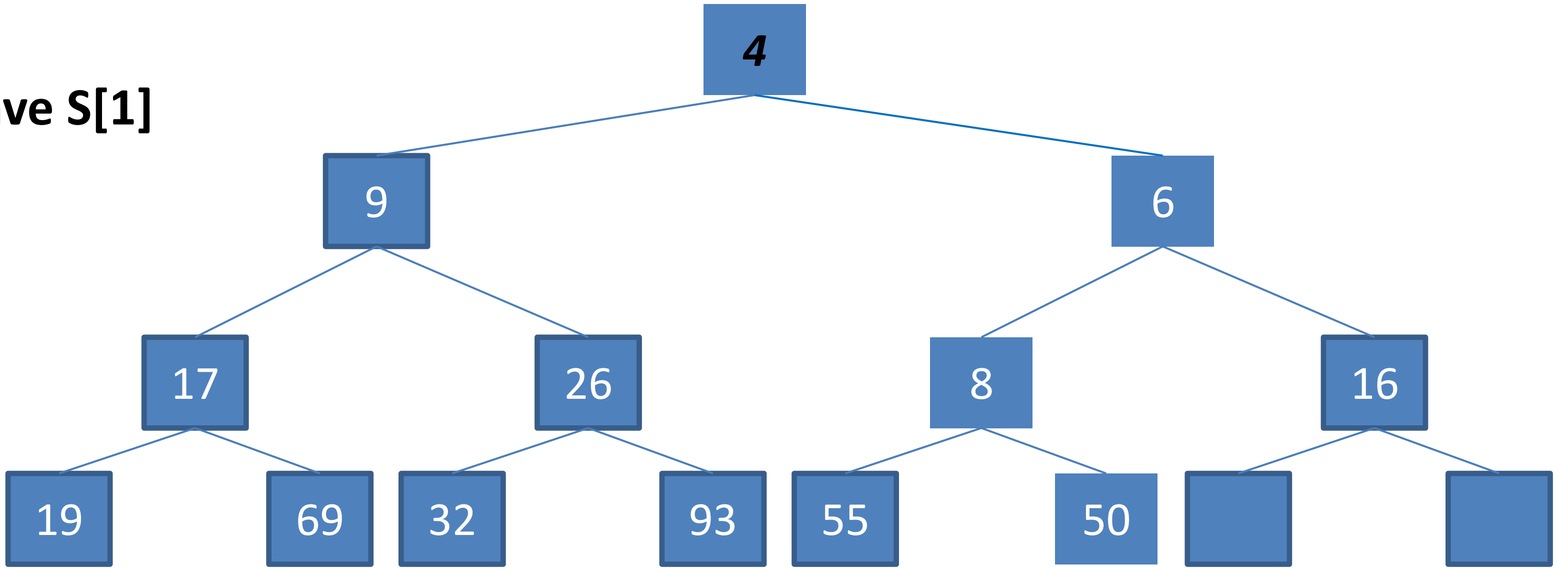
last: ***13***

capacity: 15

An implementation of a Heap data structure

4

Save S[1]



S	**	***4***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**	**50**		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

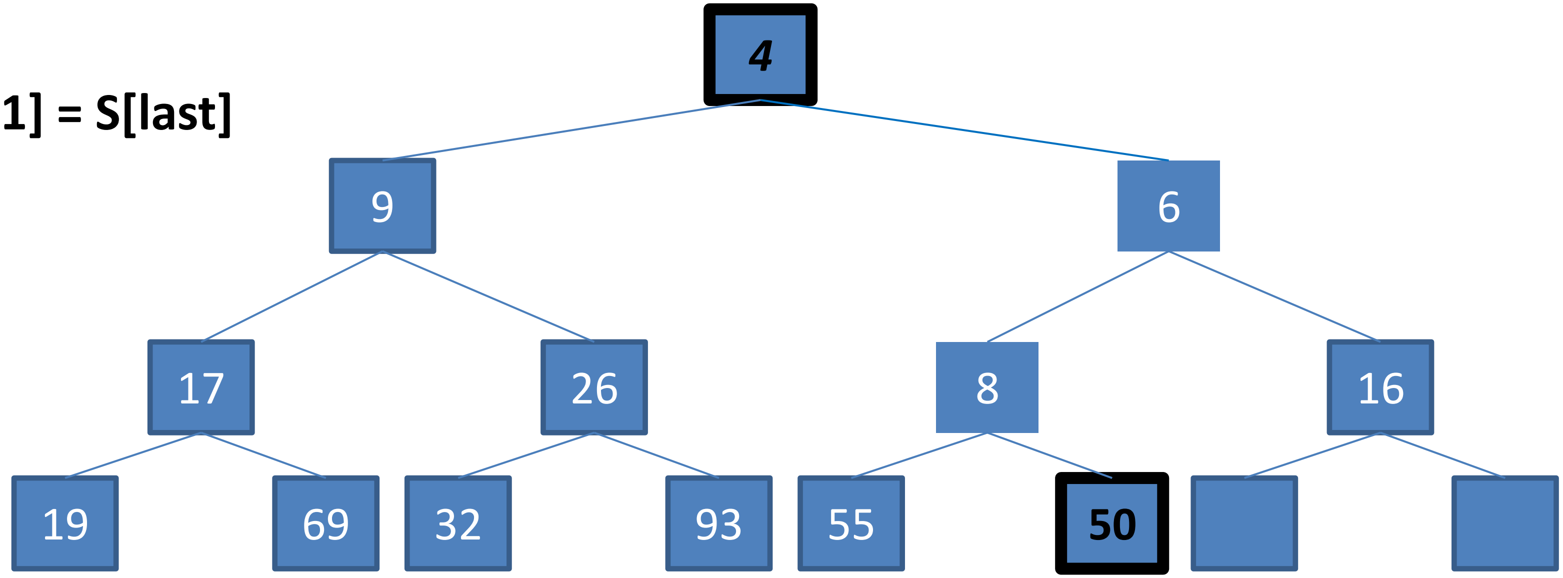
last: ***13***

capacity: 15

An implementation of a Heap data structure

4

S[1] = S[last]



S	**	***4***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**	***50***		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

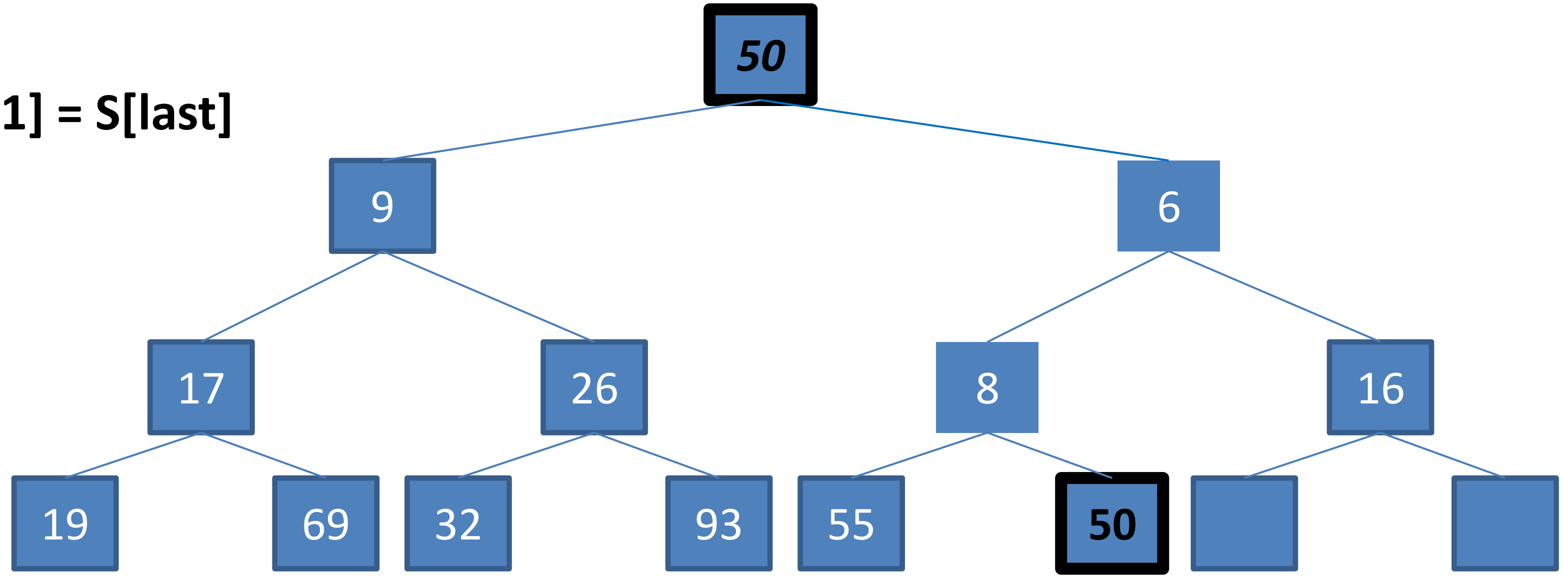
last: ***13***

capacity: 15

An implementation of a Heap data structure

4

S[1] = S[last]



S

**	*50*	9	6	17	26	8	16	19	69	32	93	55	*50*		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

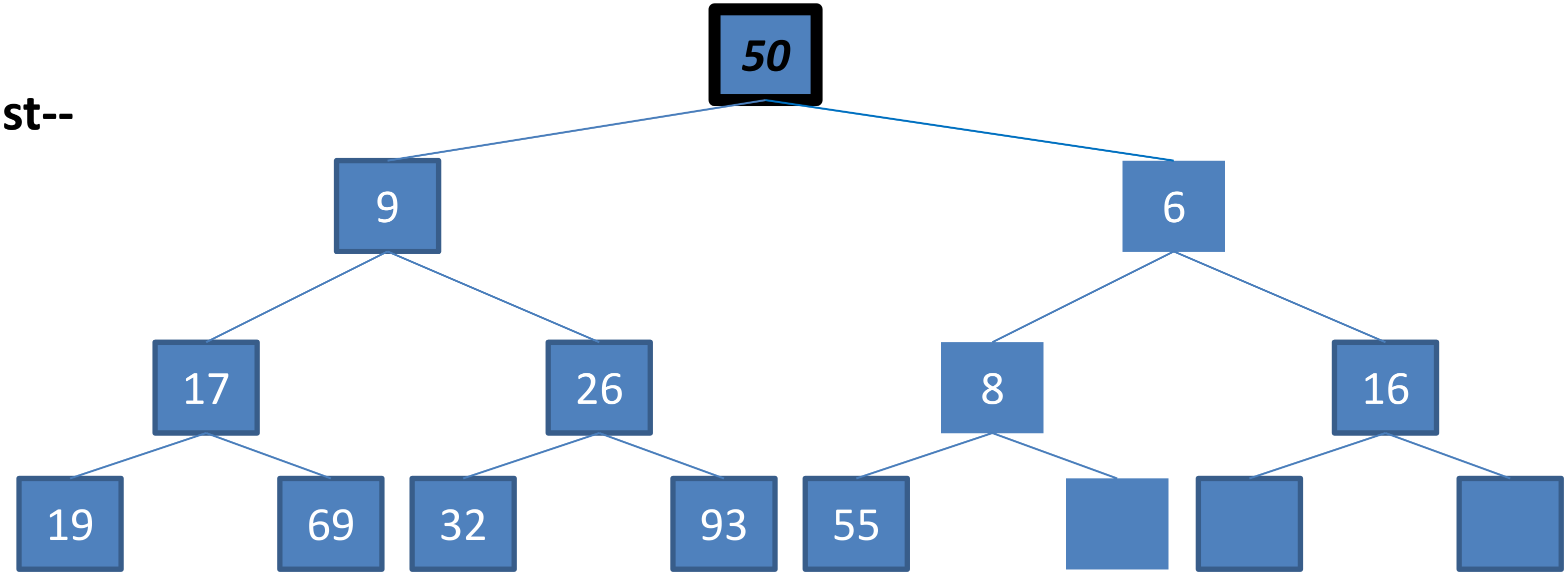
last: ***13***

capacity: 15

An implementation of a Heap data structure

4

last--



S	**	***50***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

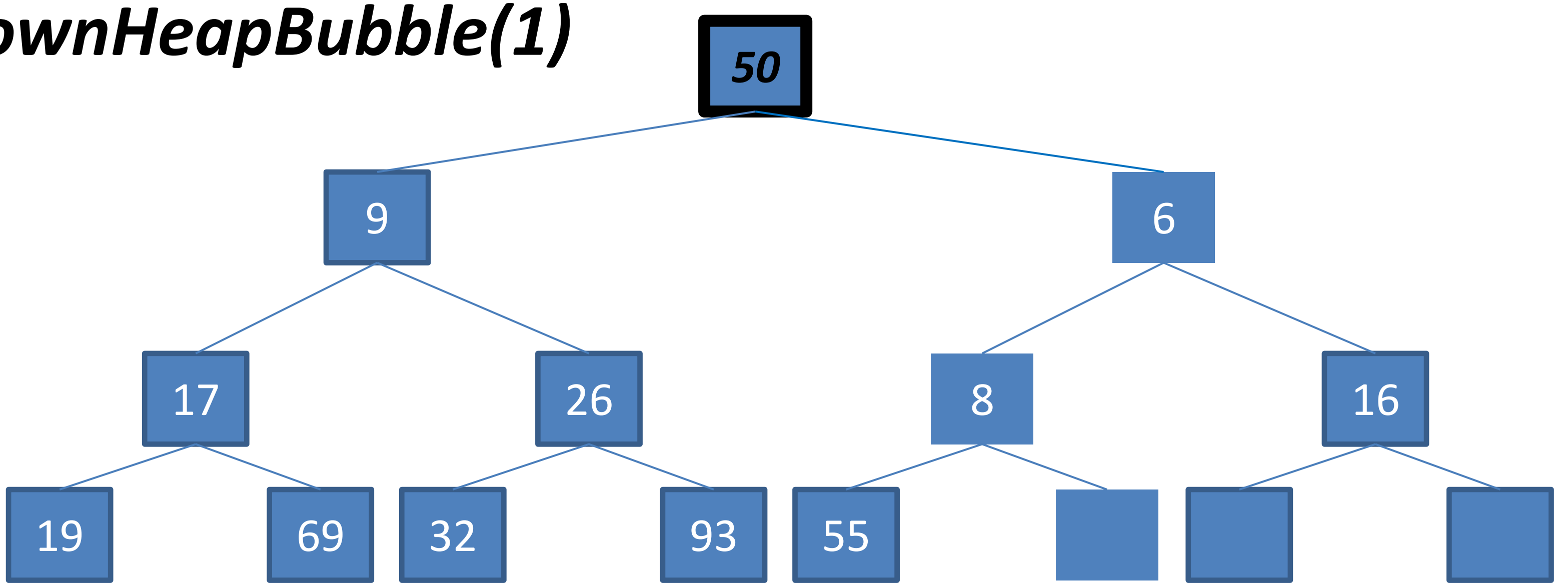
last: ***12***

capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(1)



S	**	***50***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

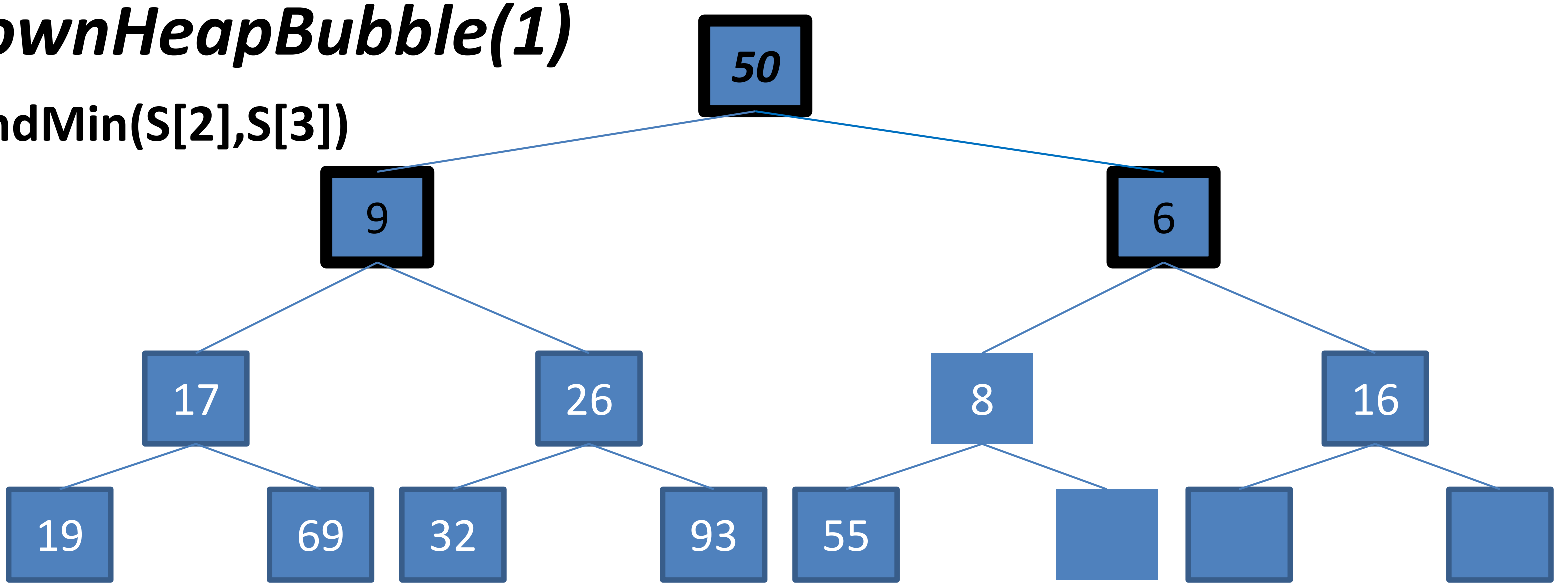
last: ***12***

capacity: 15

4

downHeapBubble(1)

findMin(S[2],S[3])



S																
	******	***50***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

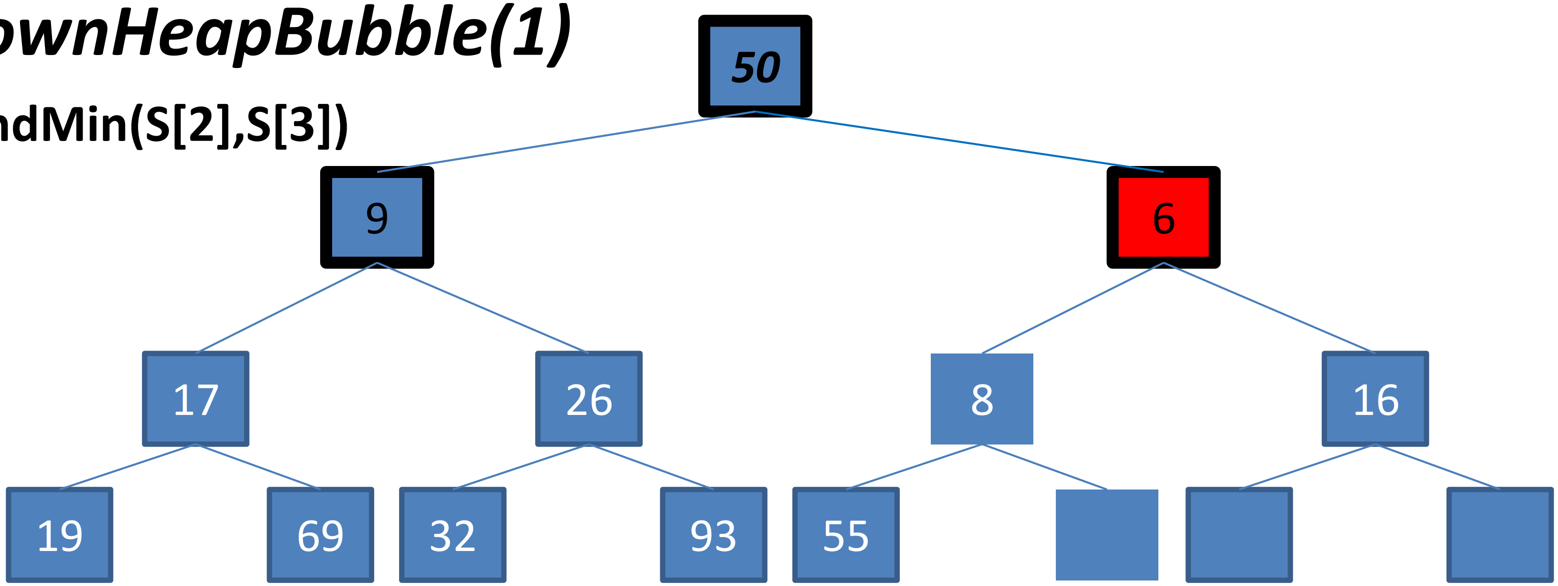
capacity: 15

An implementation of a Heap data structure

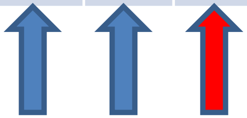
4

downHeapBubble(1)

findMin(S[2],S[3])



S	**	***50***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



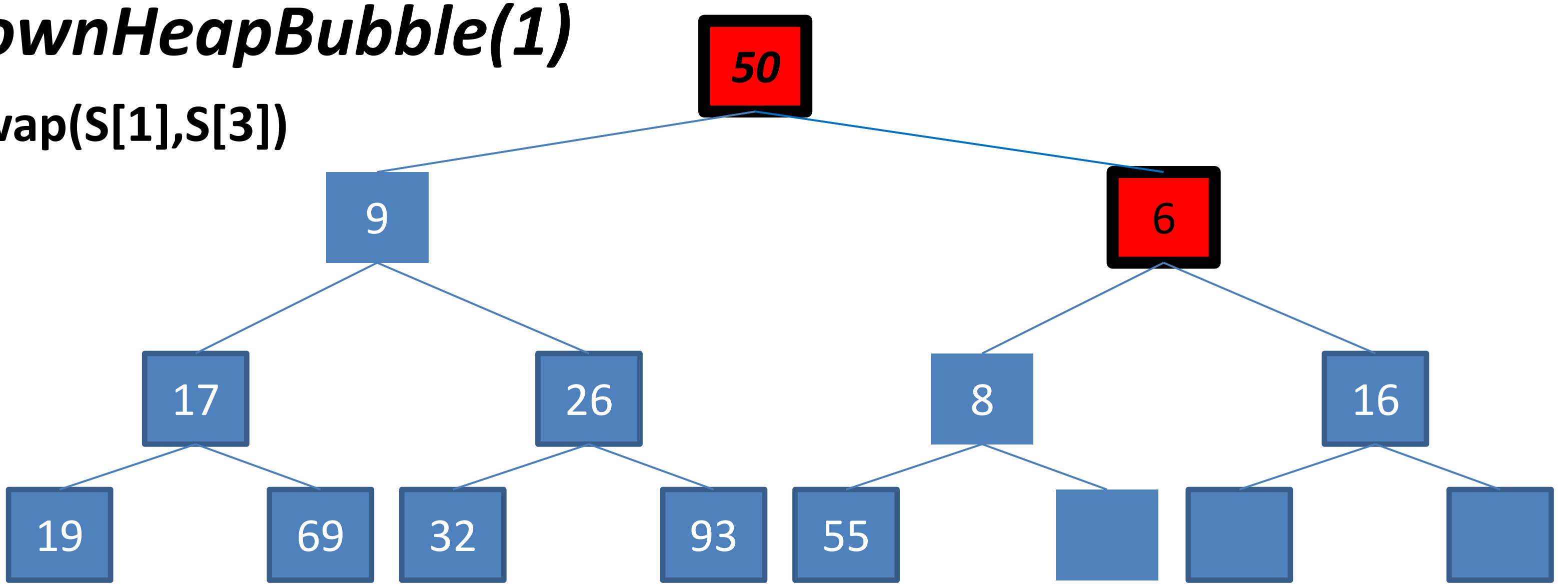
last: ***12***

capacity: 15

4

downHeapBubble(1)

swap(S[1],S[3])



S	**	***50***	**9**	**6**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

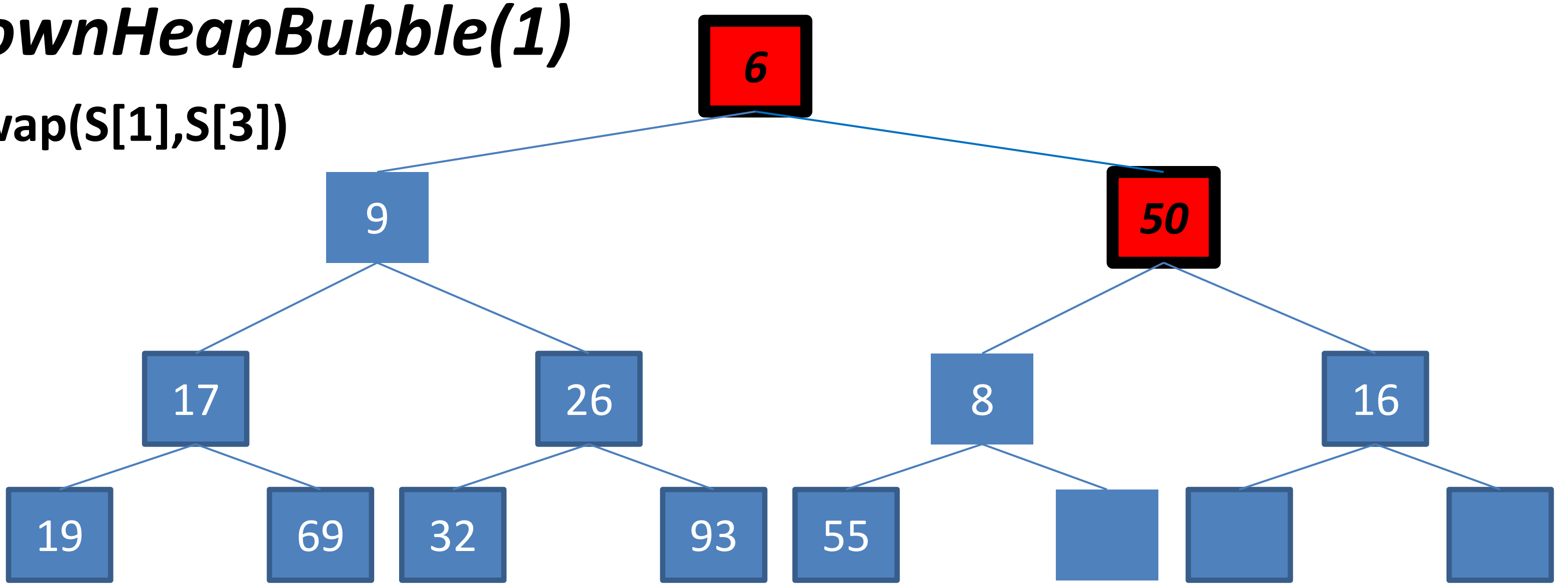
capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(1)

swap(S[1],S[3])



S	**	***6***	**9**	**50**	**17**	**26**	**8**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

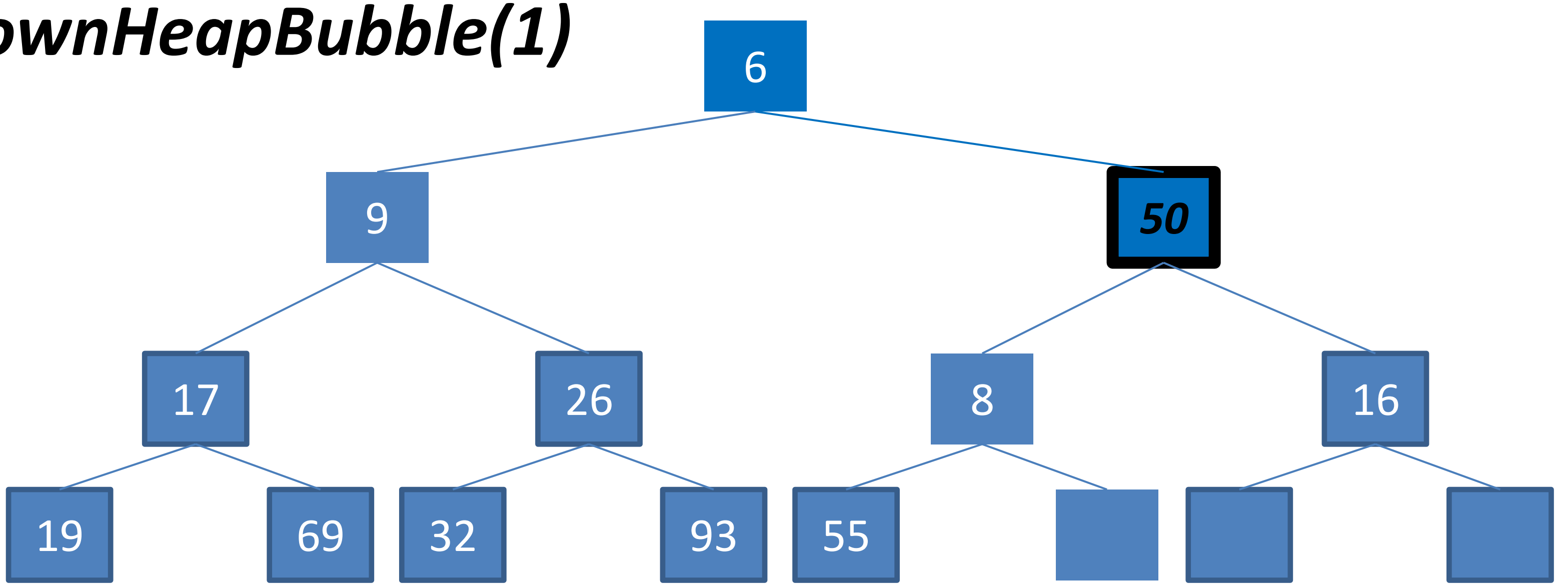
last: ***12***

capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(1)



S	**	6	9	50	17	26	8	16	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

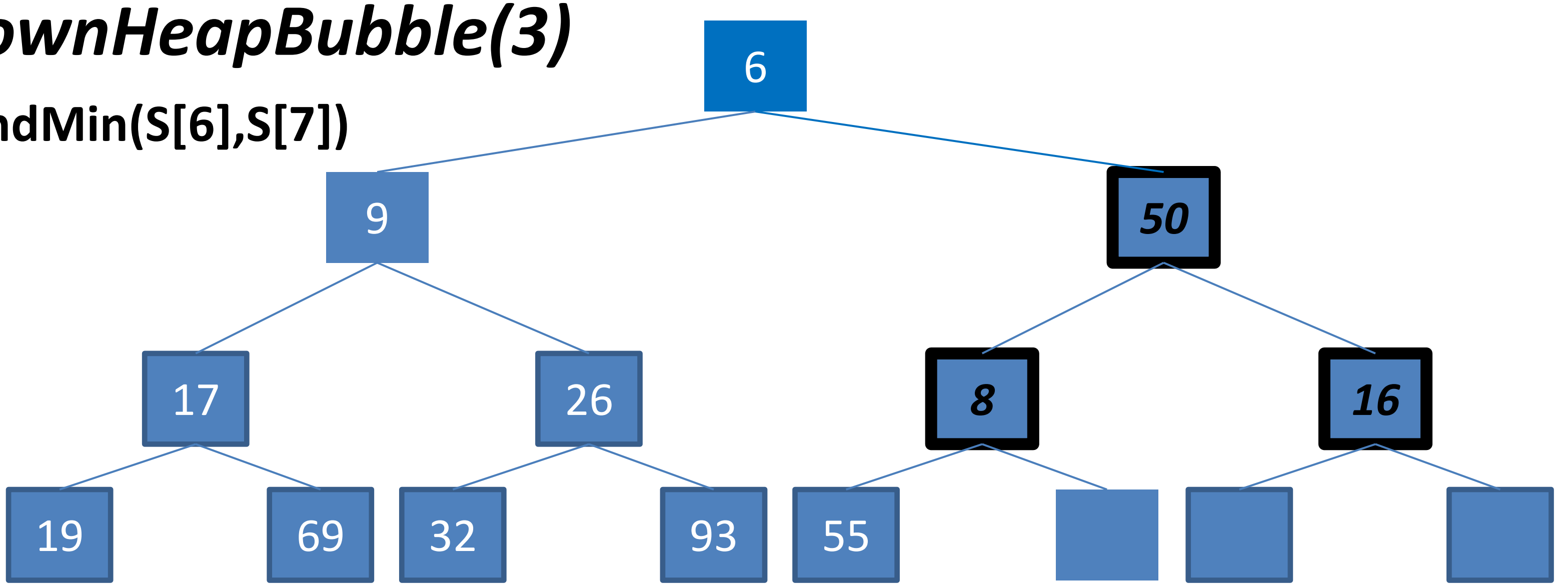
capacity: 15

An implementation of a Heap data structure

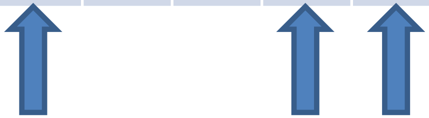
4

downHeapBubble(3)

findMin(S[6],S[7])



S	**	6	9	50	17	26	*8*	*16*	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



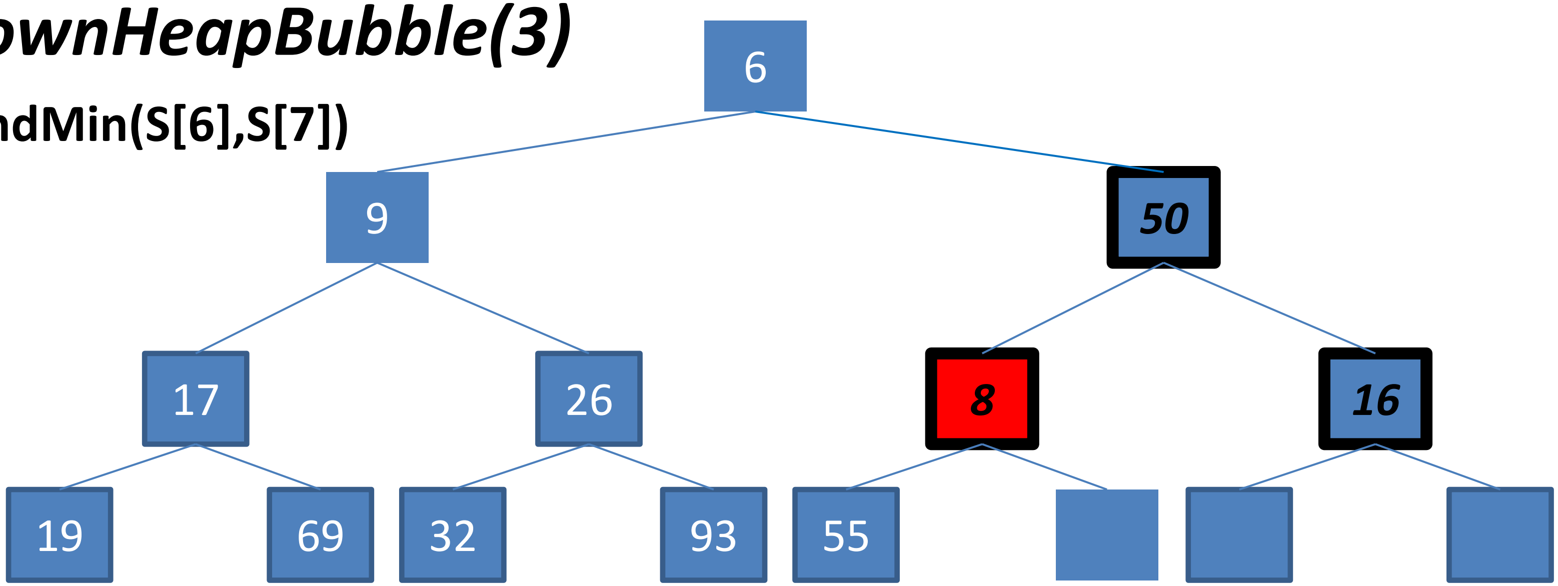
last: ***12***

capacity: 15

4

downHeapBubble(3)

findMin(S[6],S[7])



S															
**	6	9	50	17	26	*8*	*16*	19	69	32	93	55			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

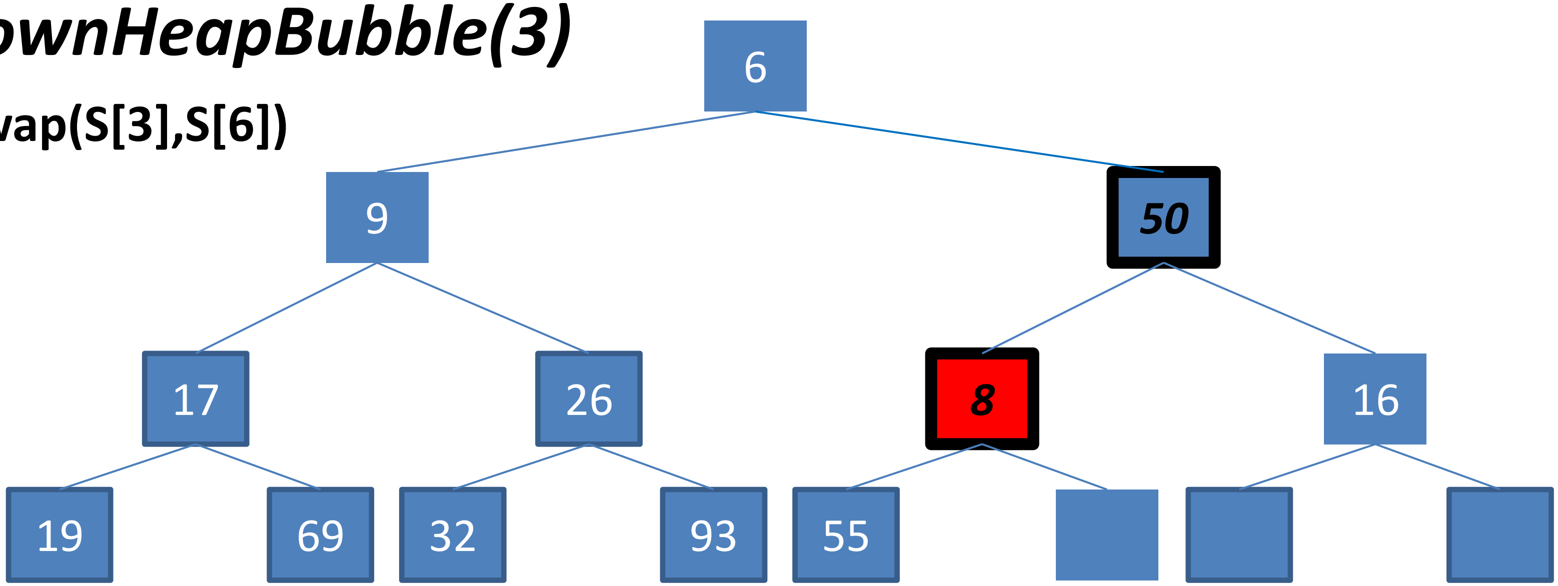
last: ***12***

capacity: 15

4

downHeapBubble(3)

swap(S[3],S[6])



S	**	6	9	50	17	26	*8*	16	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

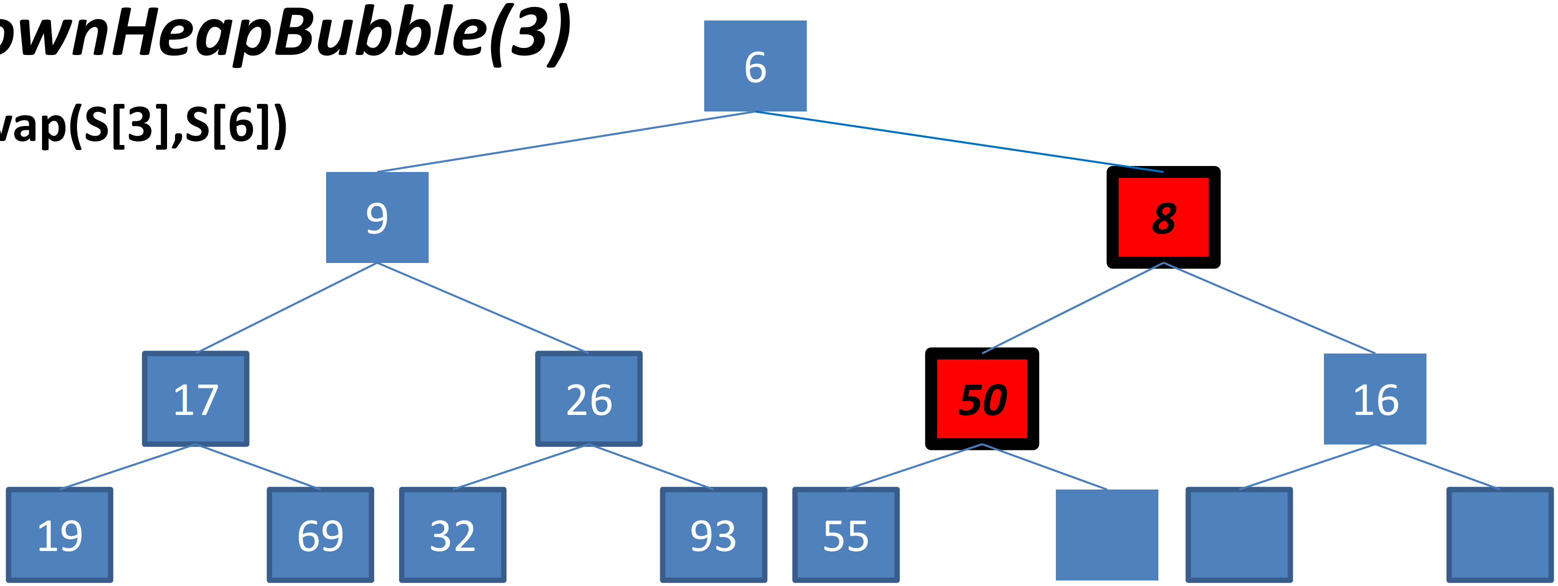
last: ***12***

capacity: 15

4

downHeapBubble(3)

swap(S[3],S[6])



S	**	6	9	8	17	26	*50*	16	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

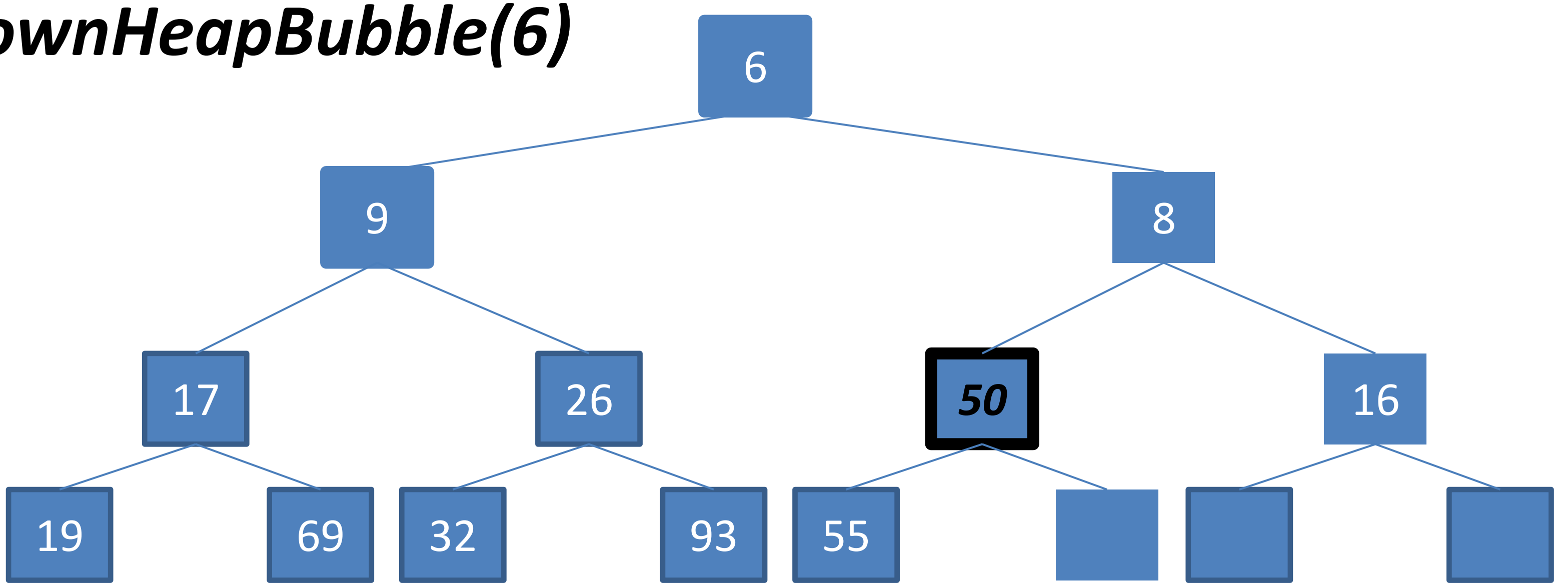
last: ***12***

capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(6)



S	**	6	9	8	17	26	***50***	16	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

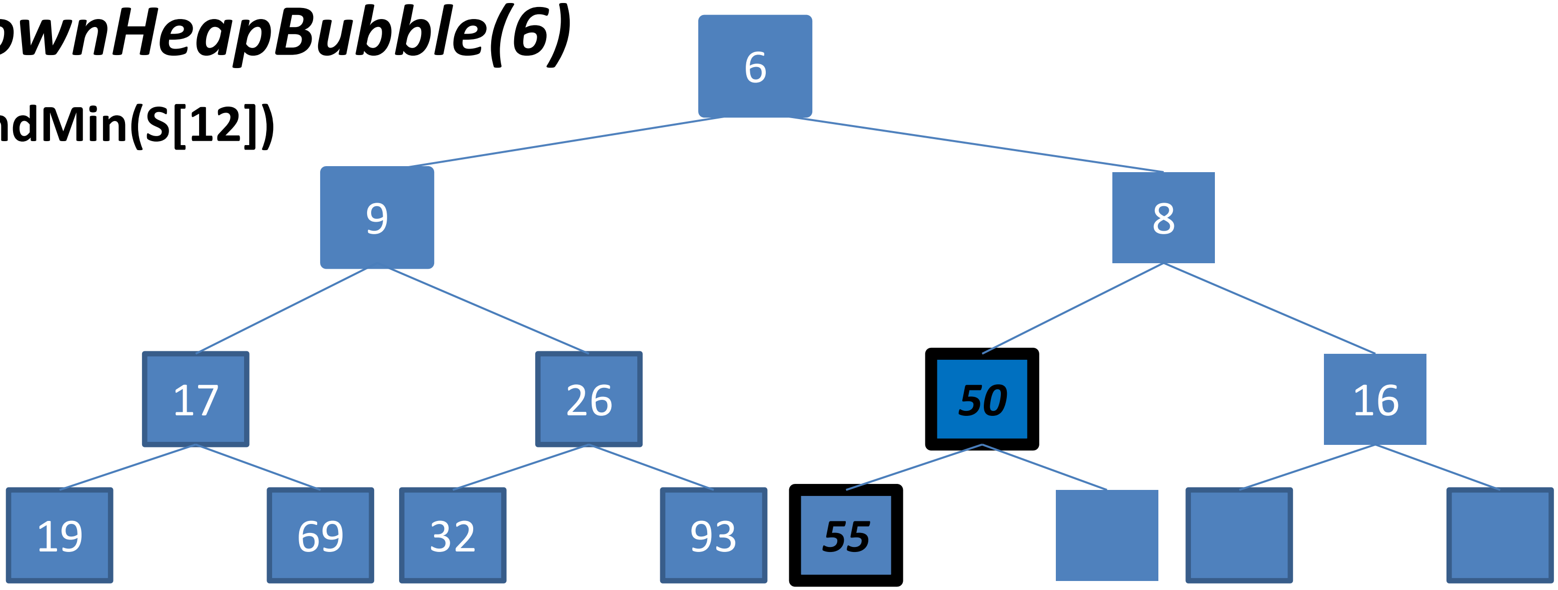
capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(6)

findMin(S[12])



S	**	6	9	8	17	26	*50*	16	19	69	32	93	*55*			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

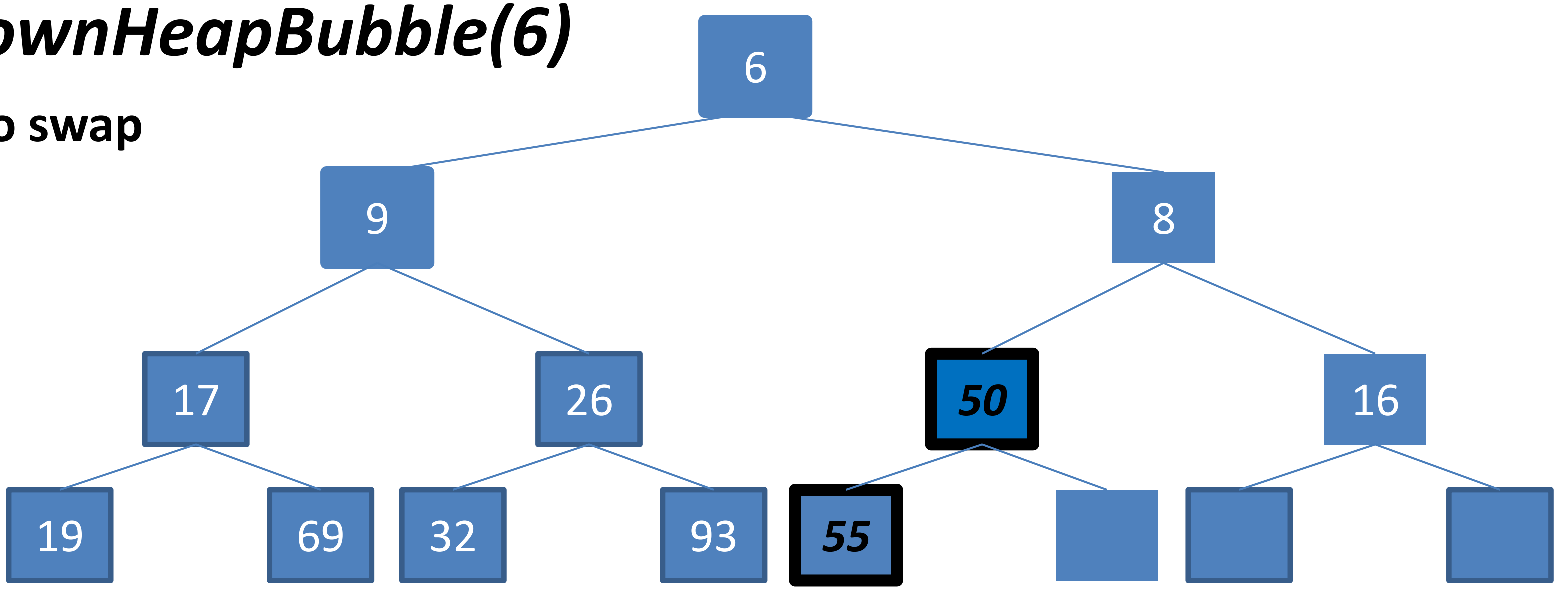
capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(6)

No swap



S	**	6	9	8	17	26	*50*	16	19	69	32	93	*55*			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

last: ***12***

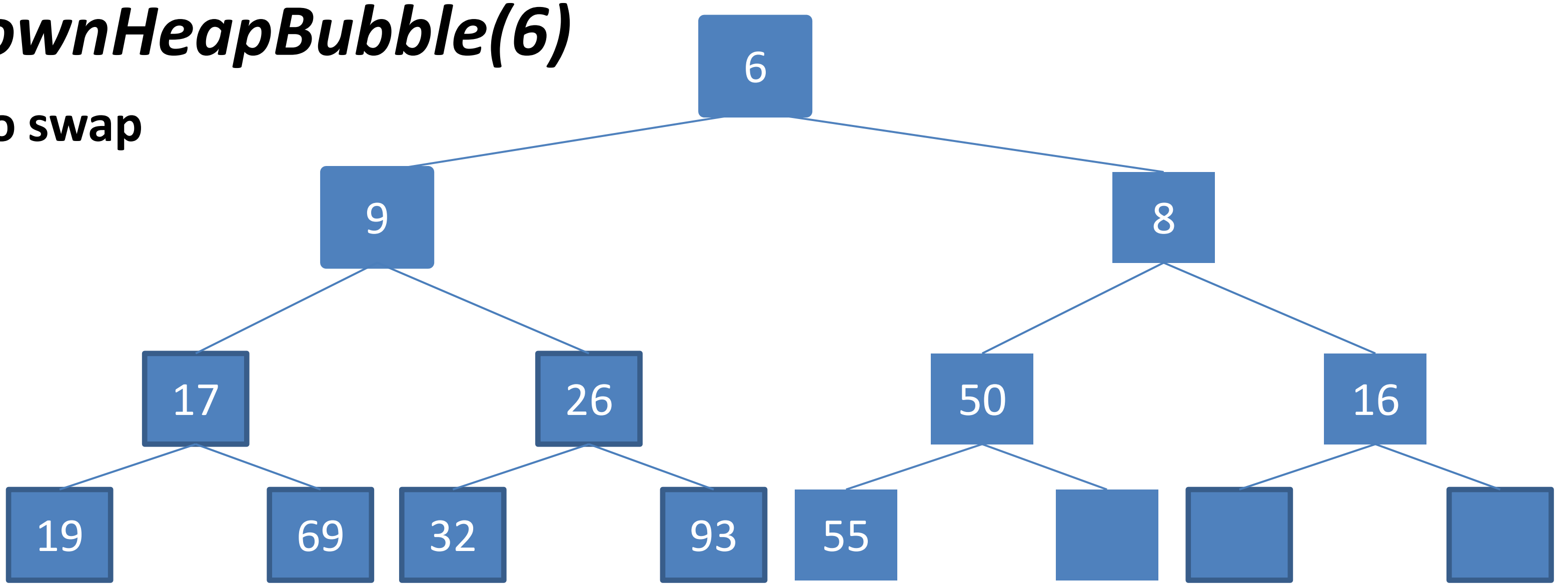
capacity: 15

An implementation of a Heap data structure

4

downHeapBubble(6)

No swap



S	**	6	9	8	17	26	50	16	19	69	32	93	55			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

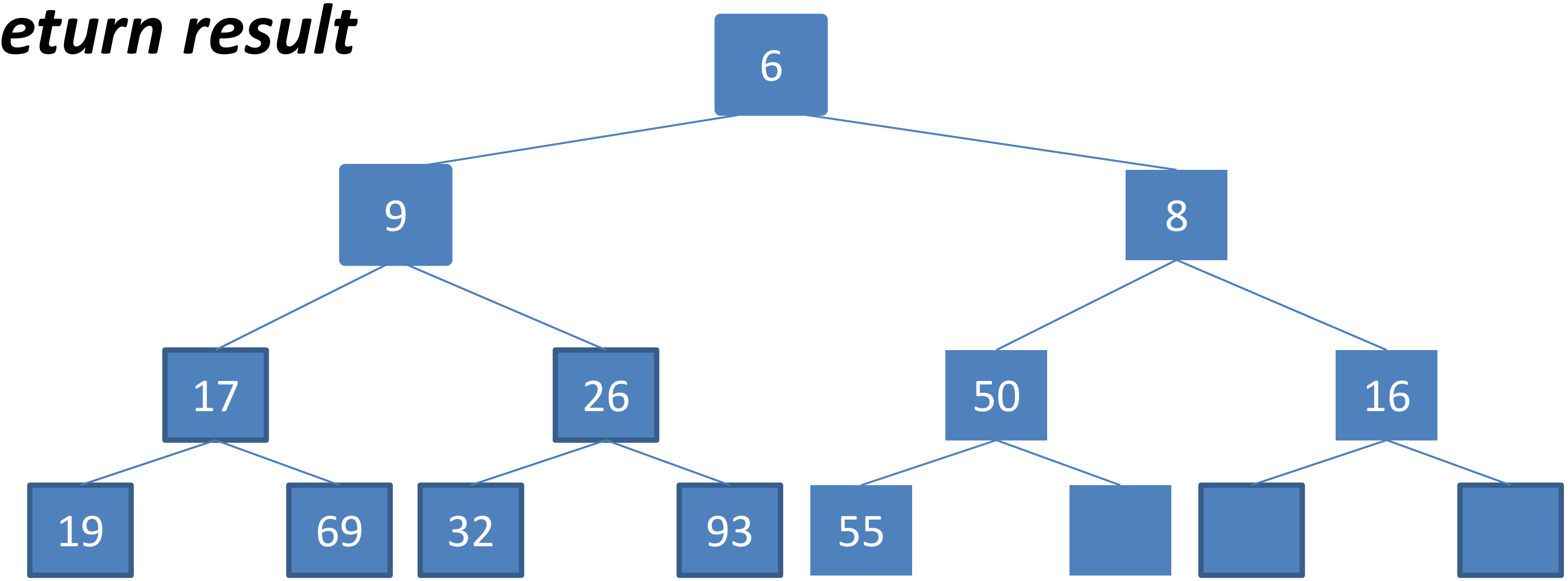
last: ***12***

capacity: 15

An implementation of a Heap data structure

4

Return result



S	**	**6**	**9**	**8**	**17**	**26**	**50**	**16**	**19**	**69**	**32**	**93**	**55**			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

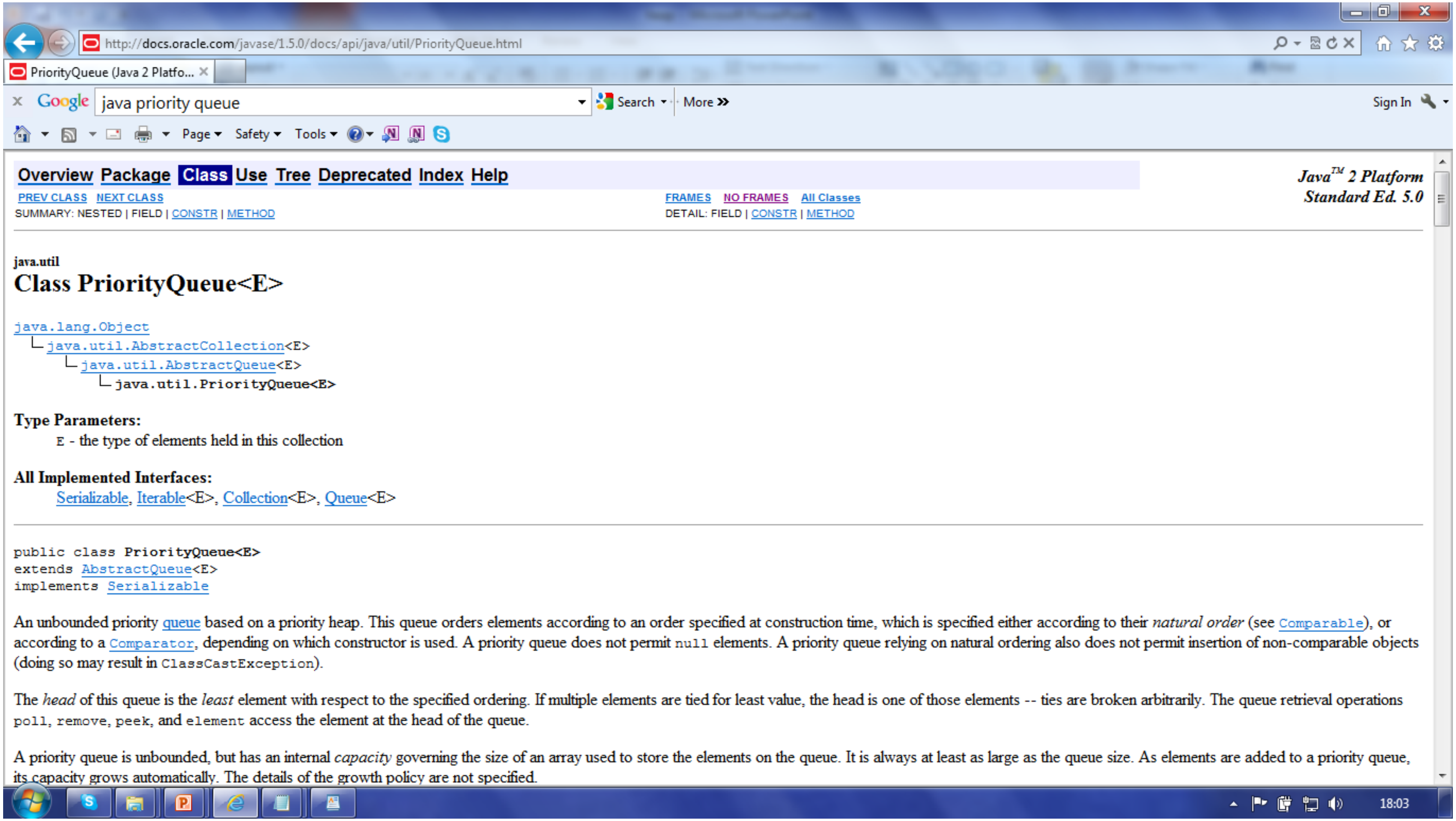
last: ***12***

capacity: 15

heap

Method	Time
size	O(1)
isEmpty	O(1)
insert	O(log(n))
removeMin	O(log(n))
min	O(1)

Have a look at priority queue as given in Java distribution


http://docs.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html
PriorityQueue (Java 2 Platfo...
Google
java priority queue
Search
More »
Sign In
Page
Safety
Tools
18:03

Overview Package **Class** Use Tree Deprecated Index Help

Java™ 2 Platform
Standard Ed. 5.0

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.util

Class PriorityQueue<E>

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractQueue<E>
          └ java.util.PriorityQueue<E>
```

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, Queue<E>

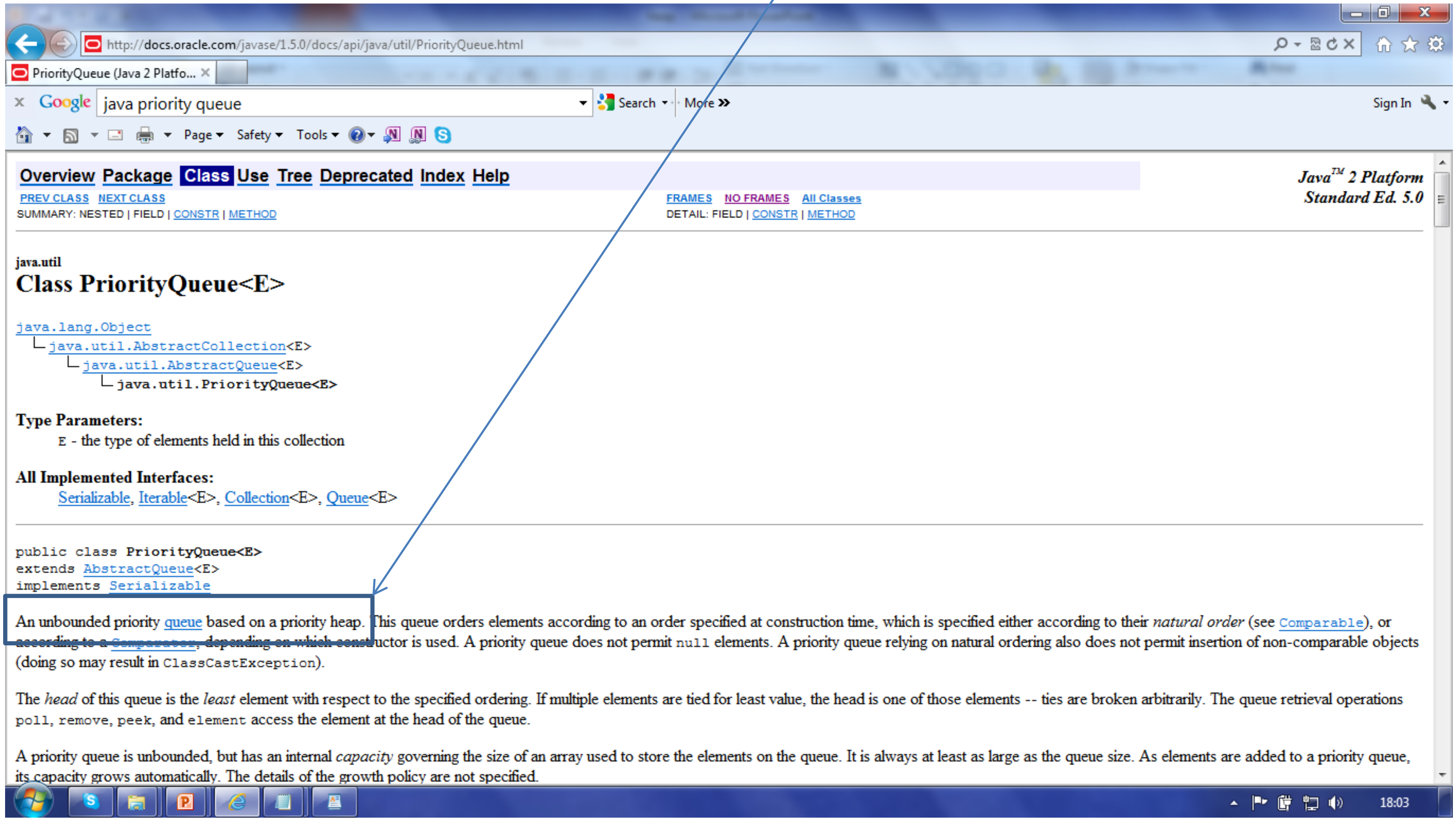
```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. This queue orders elements according to an order specified at construction time, which is specified either according to their *natural order* (see `Comparable`), or according to a `Comparator`, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations `poll`, `remove`, `peek`, and `element` access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

… based on a priority heap


http://docs.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html
PriorityQueue (Java 2 Platfo...
Google
java priority queue
Search
More »
Sign In
Page
Safety
Tools
18:03

Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

Java™ 2 Platform Standard Ed. 5.0

java.util
Class PriorityQueue<E>

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractQueue<E>
          └java.util.PriorityQueue<E>
```

Type Parameters:
E - the type of elements held in this collection

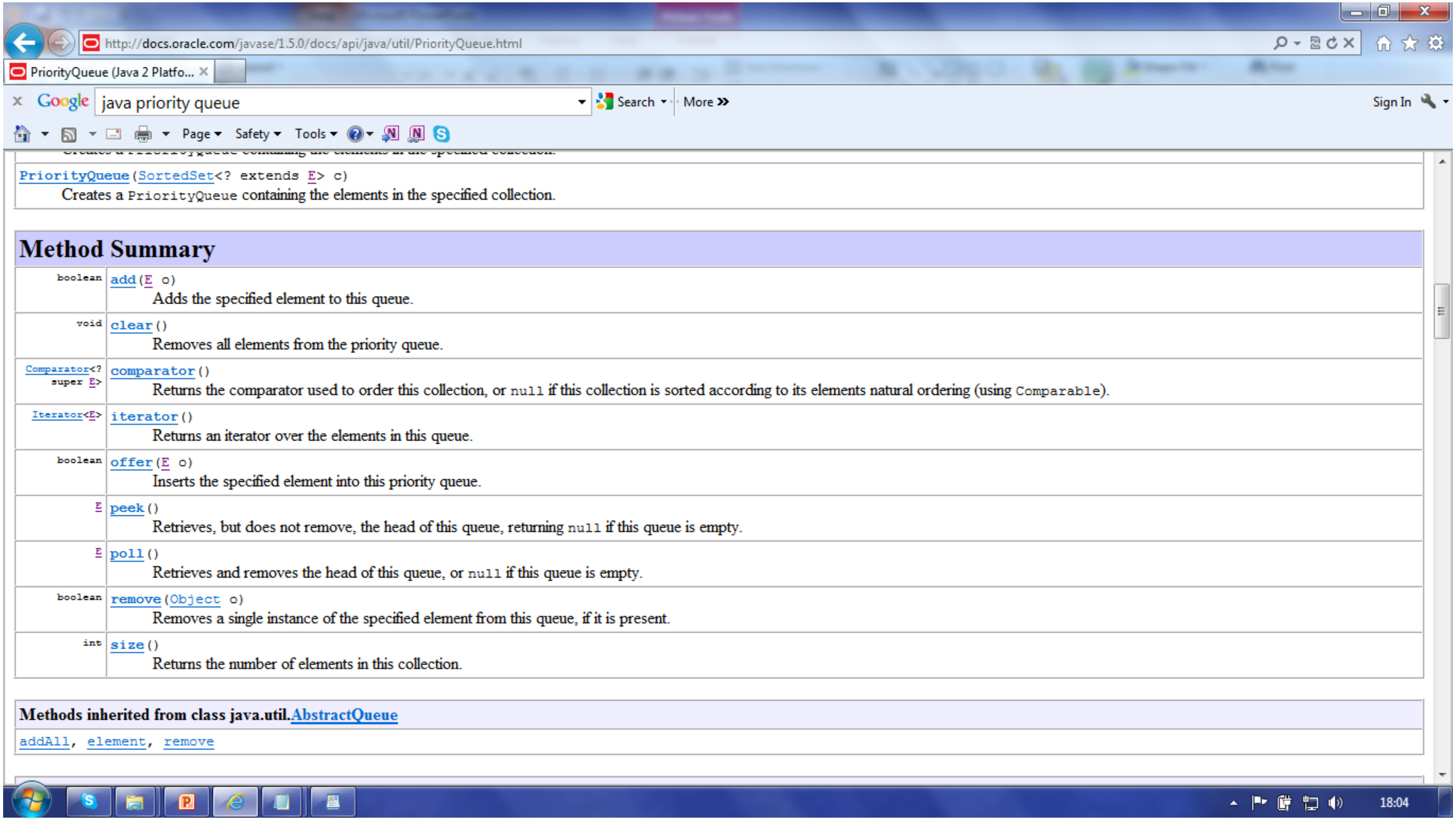
All Implemented Interfaces:
Serializable, Iterable<E>, Collection<E>, Queue<E>

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. This queue orders elements according to an order specified at construction time, which is specified either according to their *natural order* (see Comparable), or according to a Comparator, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.


http://docs.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html
PriorityQueue (Java 2 Platfo...
Google
java priority queue
Search
More »
Sign In
Page
Safety
Tools
18:04

Creates a PriorityQueue containing the elements in the specified collection.

`PriorityQueue(SortedSet<? extends E> c)`
Creates a `PriorityQueue` containing the elements in the specified collection.

Method Summary

`boolean`	`add(E o)` Adds the specified element to this queue.
`void`	`clear()` Removes all elements from the priority queue.
`Comparator<? super E>`	`comparator()` Returns the comparator used to order this collection, or `null` if this collection is sorted according to its elements natural ordering (using `Comparable`).
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this queue.
`boolean`	`offer(E o)` Inserts the specified element into this priority queue.
`E`	`peek()` Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
`E`	`poll()` Retrieves and removes the head of this queue, or `null` if this queue is empty.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this queue, if it is present.
`int`	`size()` Returns the number of elements in this collection.

Methods inherited from class java.util.AbstractQueue

`addAll, element, remove`

Different method names … add rather than insert

Your mission, should you choose to accept it …

Exercise 4 (***assessed***)

- Implement the Heap
- Use it for sorting

FIN