CS151 Intro to Data Structures

Array-based Heaps

Outline

- Array Based Trees
- Breath First Traversal
- Array Based Heaps
- More efficient way to Construct Heaps

Announcements

HW05 due next Tuesday

Will be released very shortly (just fixing checkstyle in starter code)

Midterm & Grading

Array

Physical memory is one-dimensional

- an enormous array of bytes

All data structures (are in our heads)

- differ only in the organization of data
- how each element is accessed (search/traversal) in relationship with the next
- how insert/remove/update affects the organization

Organizational Types

Arrays

- Contiguous – next element is next in memory
- Directional

Linked List

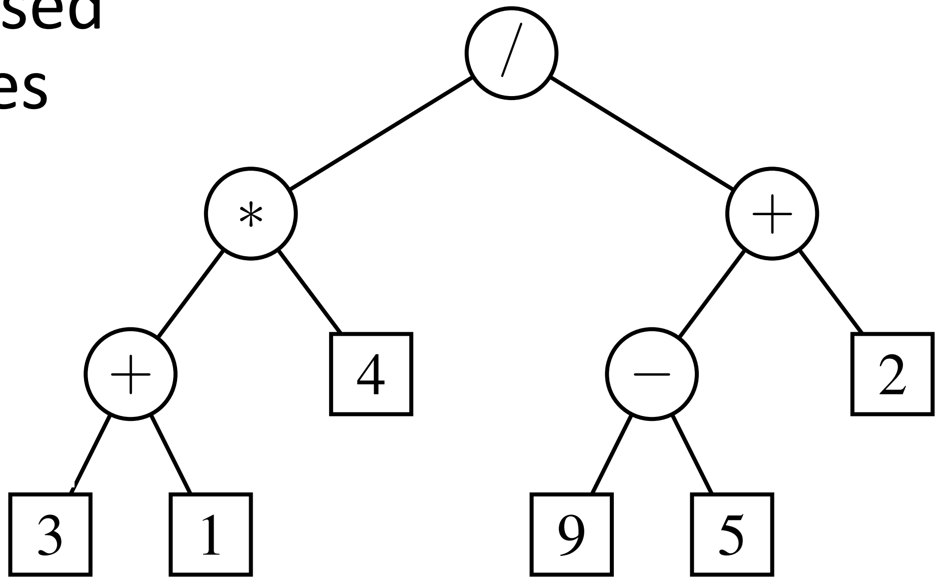
- Noncontiguous
- Directional

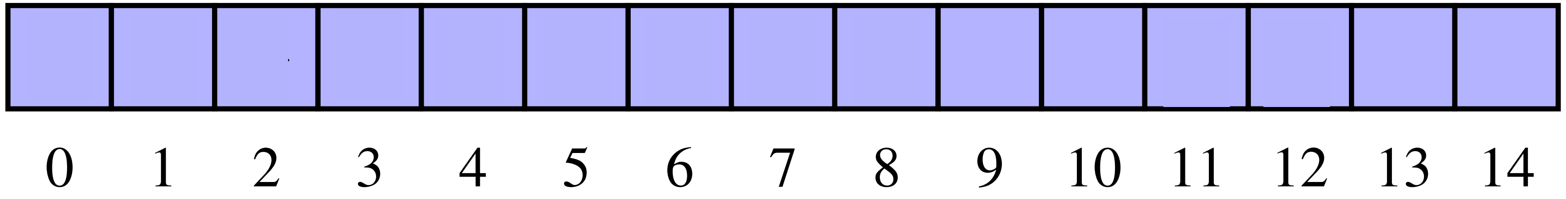
Tree

- Noncontiguous
- Multi-Directional

Array-based Binary Tree

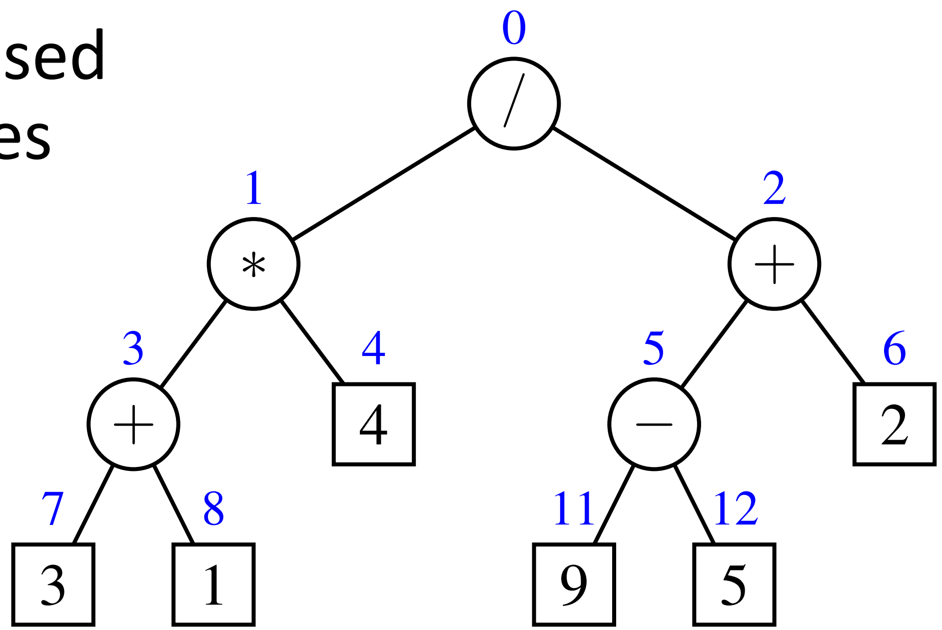
- The numbering can then be used as indices for storing the nodes directly in an array
- $f(root) = 0$
- $f(l) = 2f(p) + 1$
- $f(r) = 2f(p) + 2$





Array-based Binary Tree

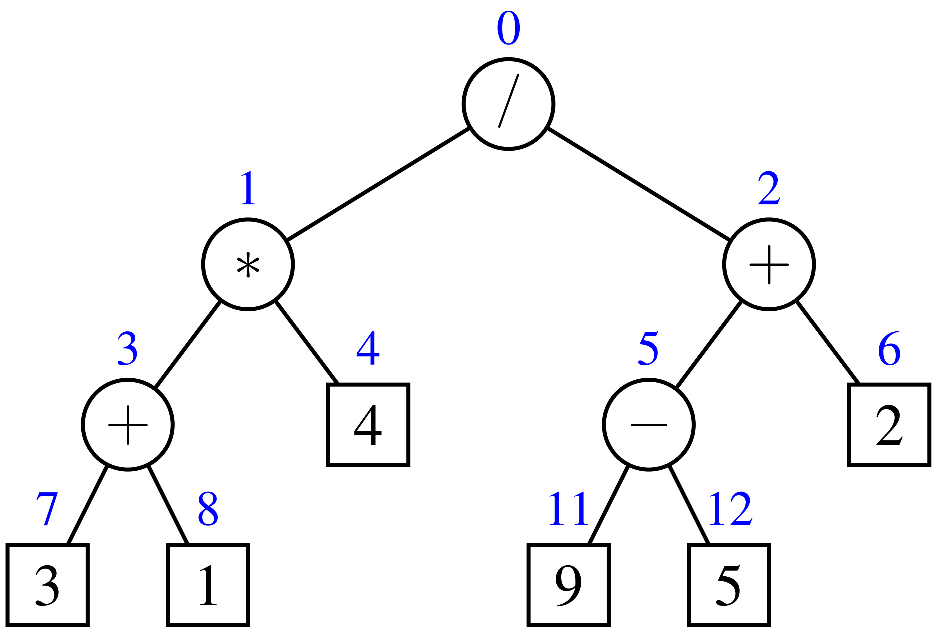
- The numbering can then be used as indices for storing the nodes directly in an array
- $f(root) = 0$
- $f(l) = 2f(p) + 1$
- $f(r) = 2f(p) + 2$

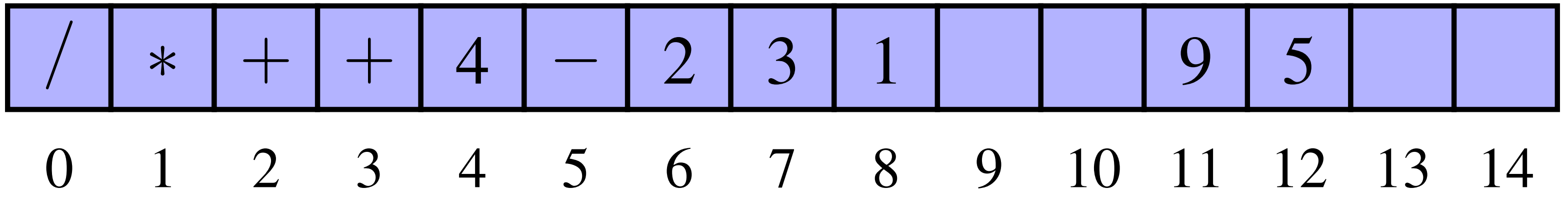


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/	*	+	+	4	−	2	3	1			9	5		

Array-based Binary Tree

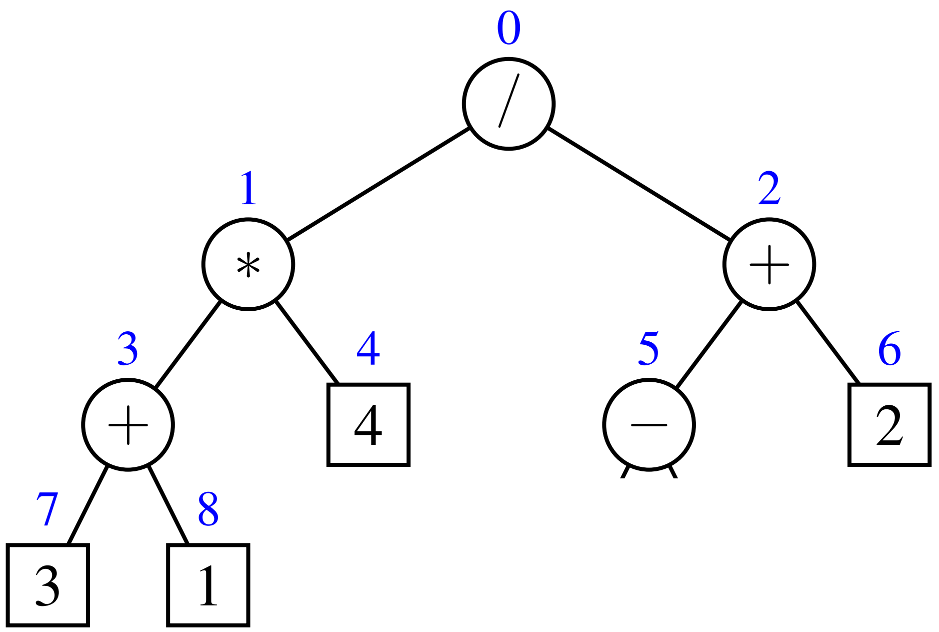
- If we don't enforce any ordering properties, then we can ensure the tree is complete
- **Complete tree** – every level is full, except for the last and all nodes in the last level are as far left as possible





Array-based Binary Tree

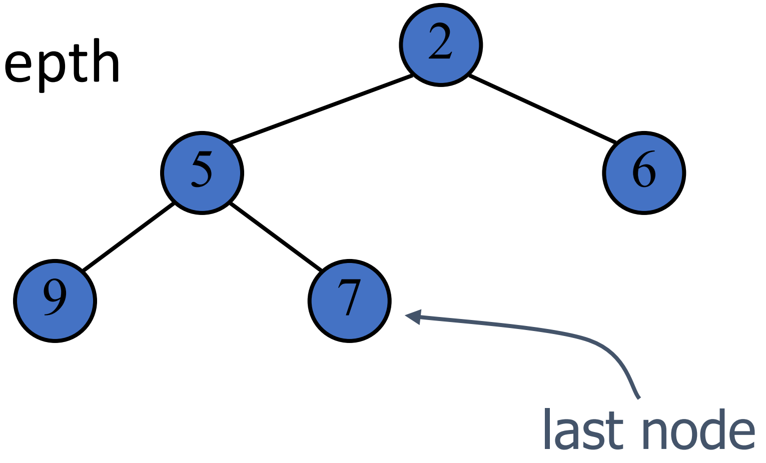
- If we don’t enforce any ordering properties, then we can ensure the tree is complete
- **Complete tree** – every level is full, except for the last and all nodes in the last level are as far left as possible



Binary Heap

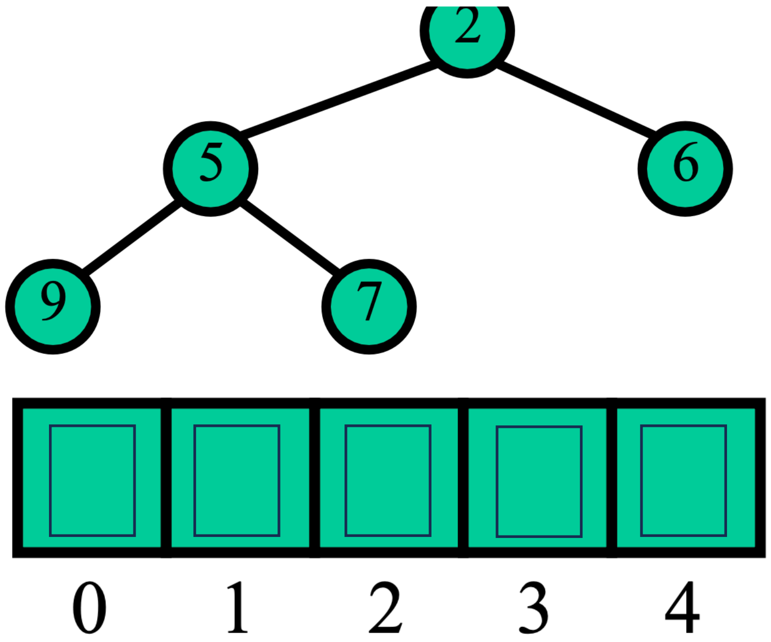
Binary tree storing keys at its nodes and satisfying:

1. heap-order: for every internal node v other than root, $key(v) \geq key(parent(v))$
2. complete binary tree: let h be the height of the heap
   - there are 2^i nodes of depth i, $0 \leq i \leq h - 1$
   - at depth h, the leaf nodes are in the leftmost positions
   - last node of a heap is the rightmost node of max depth



Array based heap

Array/ArrayList of length n
for heap with n keys

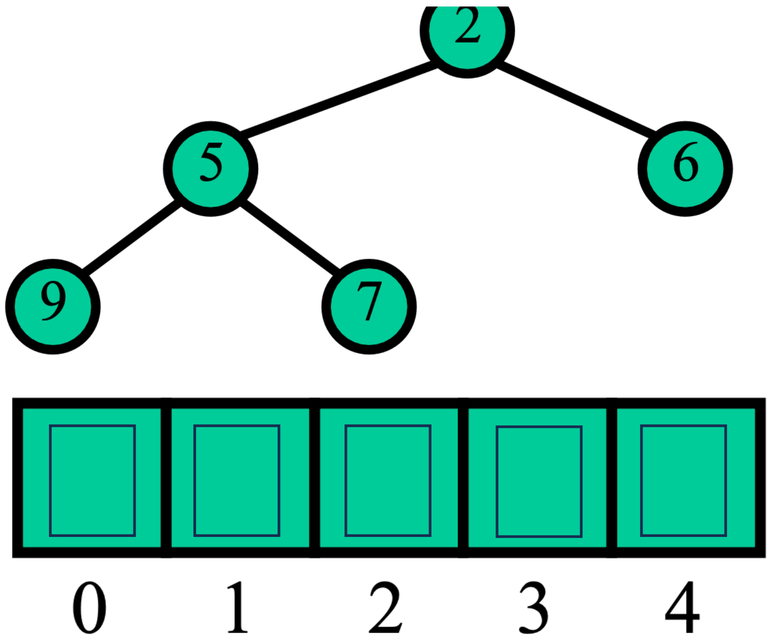


Array based heap

Array/ArrayList of length *n* for heap with *n* keys

Node at index i

- Left child index:
  - 2i + I
- Right child index:
  - 2i + 2



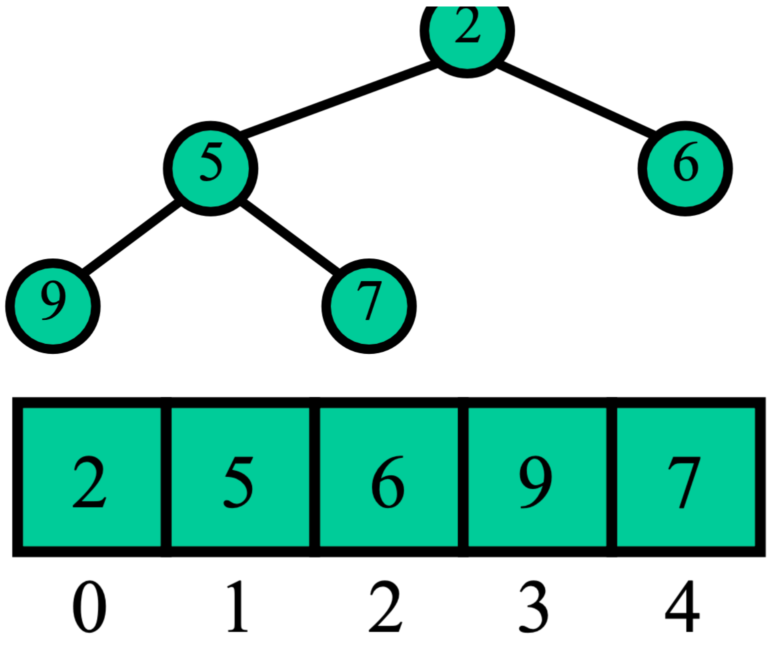
Array based heap

Array/ArrayList of length *n* for heap with *n* keys

Node at index i

- Left child index:
  - 2i + I
- Right child index:
  - 2i + 2

- Peek:
  - Get element at index 0
- Poll:
  - Remove element at index 0
- No need to store references/links



Binary Tree Interface

```
public interface BinaryTree<E extends Comparable<E>>
{
     E getRootElement();
     int size();
     boolean isEmpty();
     boolean contains(E element);
     void insert(E element);
     boolean remove(E element);
     String toStringInOrder();
     String toStringPreOrder();
     String toStringPostOrder();
}
```

Array Based Binary Tree

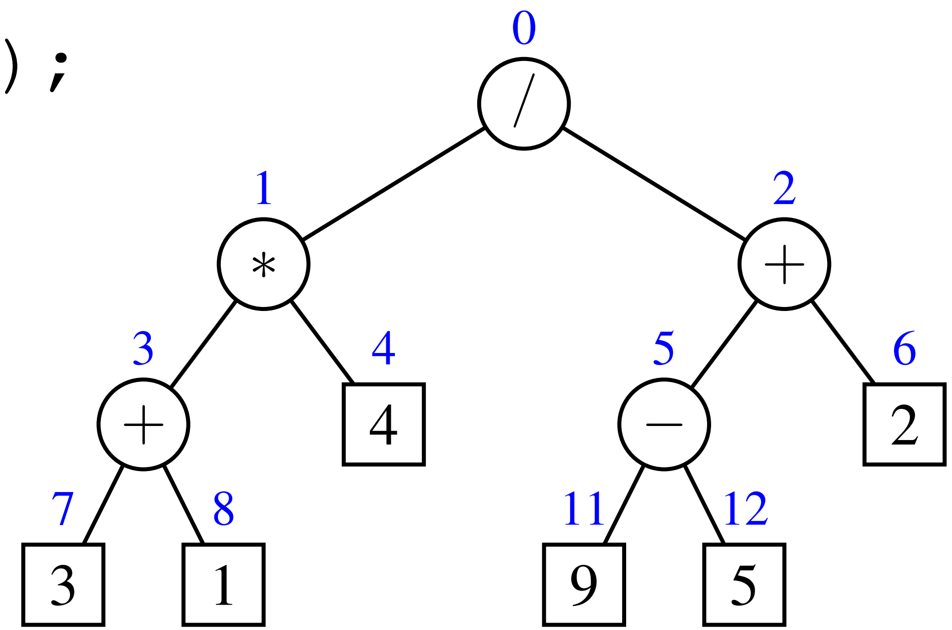
```
public class ArrayBinaryTree<E extends Comparable<E>>
implements BinaryTree<E>{
      public static final int CAPACITY=1000;
      //instance variables?
      private int size;
      private E[] data;
      public ArrayBinaryTree(){
        // what would our constructor look like?
      }
      //E getRootElement();
      //int size();
      //boolean isEmpty();
}
```

Array Based Binary Tree

```
public class ArrayBinaryTree<E extends Comparable<E>>
implements BinaryTree<E>{
      public static final int CAPACITY=1000;
      //instance variables?
      private int size;
      private E[] data;
      public ArrayBinaryTree(){
            data = (E[]) new Comparable[CAPACITY];
      }
      //E getRootElement();
      //int size();
      //boolean isEmpty();
}
```

Array Based Binary Tree Methods

- `boolean contains(E element);`
- `void insert(E element);`
- `boolean remove(E element);`



$$f(root) = 0$$
$$f(l) = 2f(p) + 1$$
$$f(r) = 2f(p) + 2$$

/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

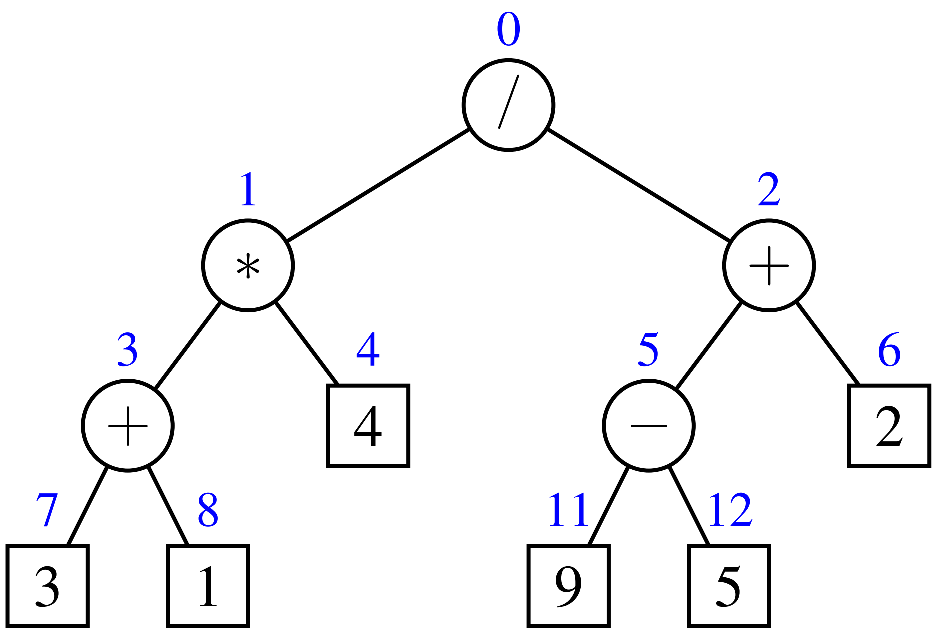
Traversals

```
String toStringInorder();

String toStringPreorder();

String toStringPostorder();

String toStringBreadthFirst();
```



/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

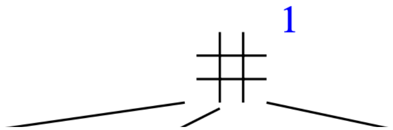
Outline

- Array Based Trees
- **Breath First Traversal**
- Array Based Heaps
- More efficient way to Construct Heaps

Breath-First Traversal

Traverse the tree level-by-level

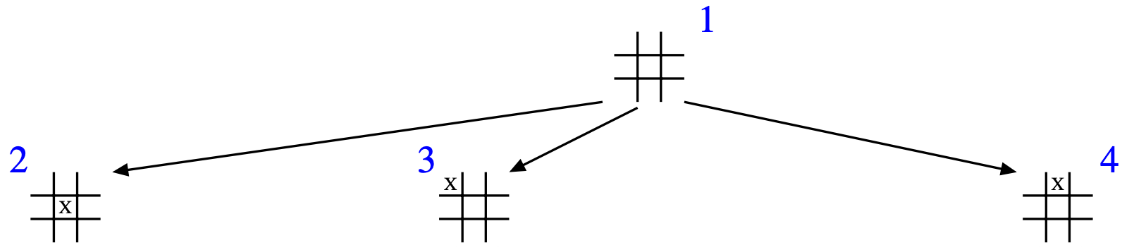
- Within a level go left-to-right



Breath-First Traversal

Traverse the tree level-by-level

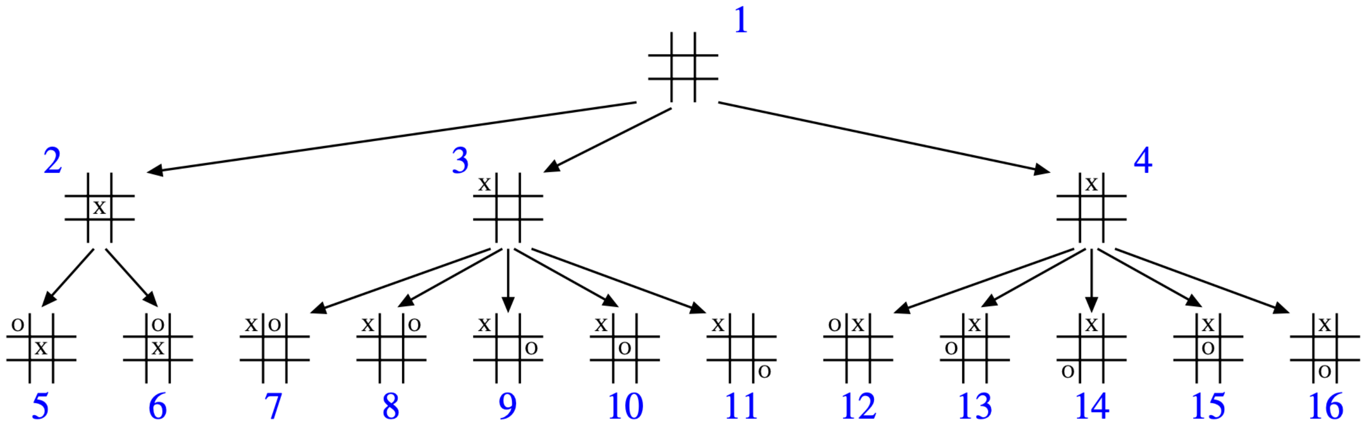
- Within a level go left-to-right



Breath-First Traversal

Traverse the tree level-by-level

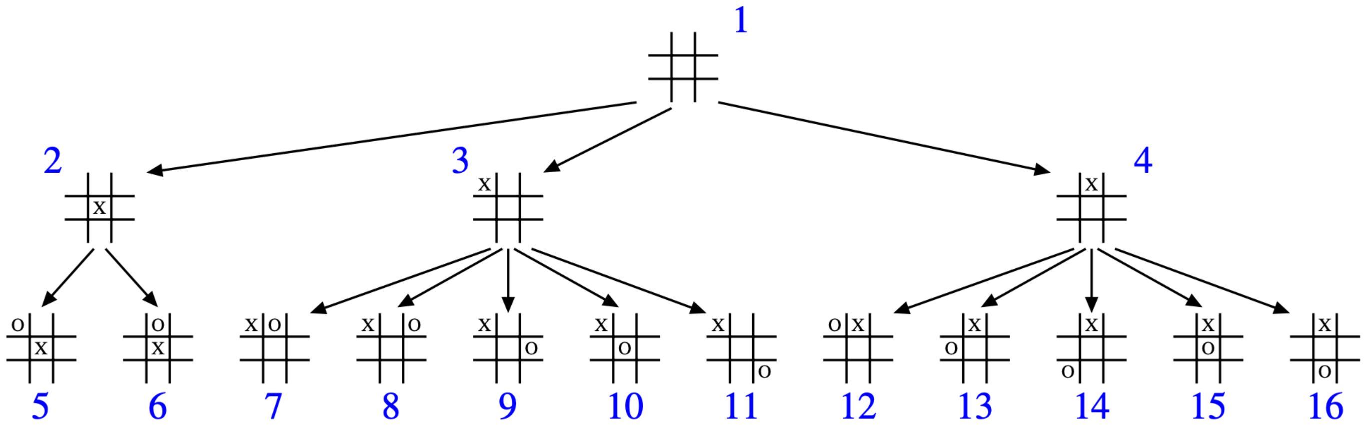
- Within a level go left-to-right



Breath-First Traversal

Traverse the tree level-by-level

- Within a level go left-to-right

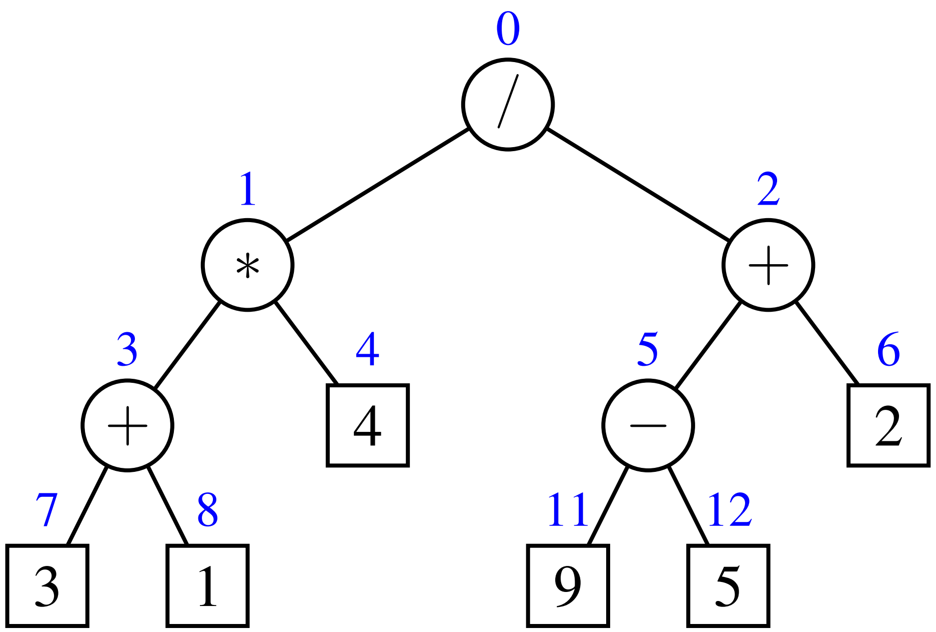


This is the array order of an array-based binary tree

Breath-First Traversal

Traverse the tree level-by-level

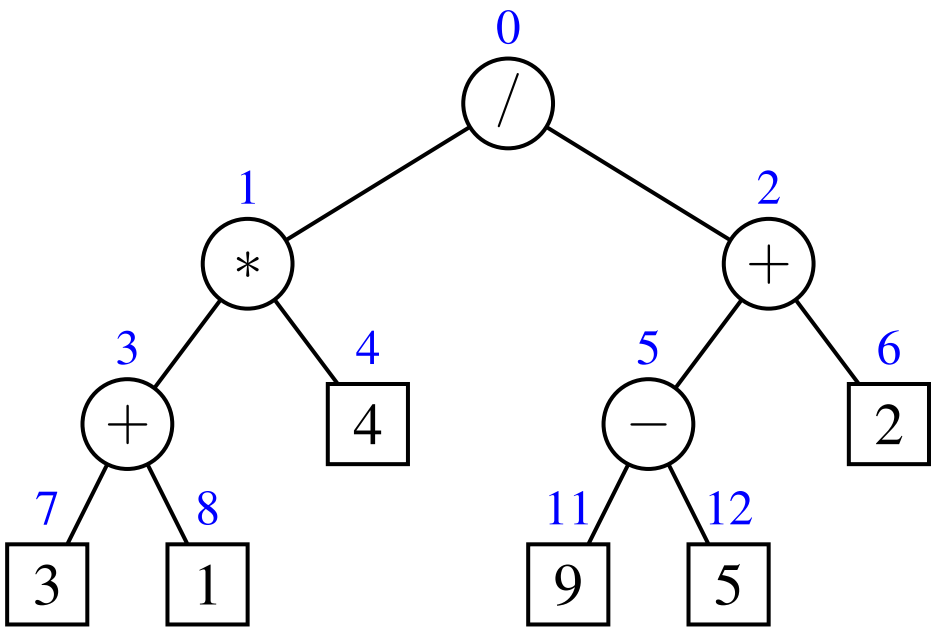
- Within a level go left-to-right



Breath-First Traversal

Traverse the tree level-by-level

- Within a level go left-to-right



/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

This is the array order of an array-based binary tree

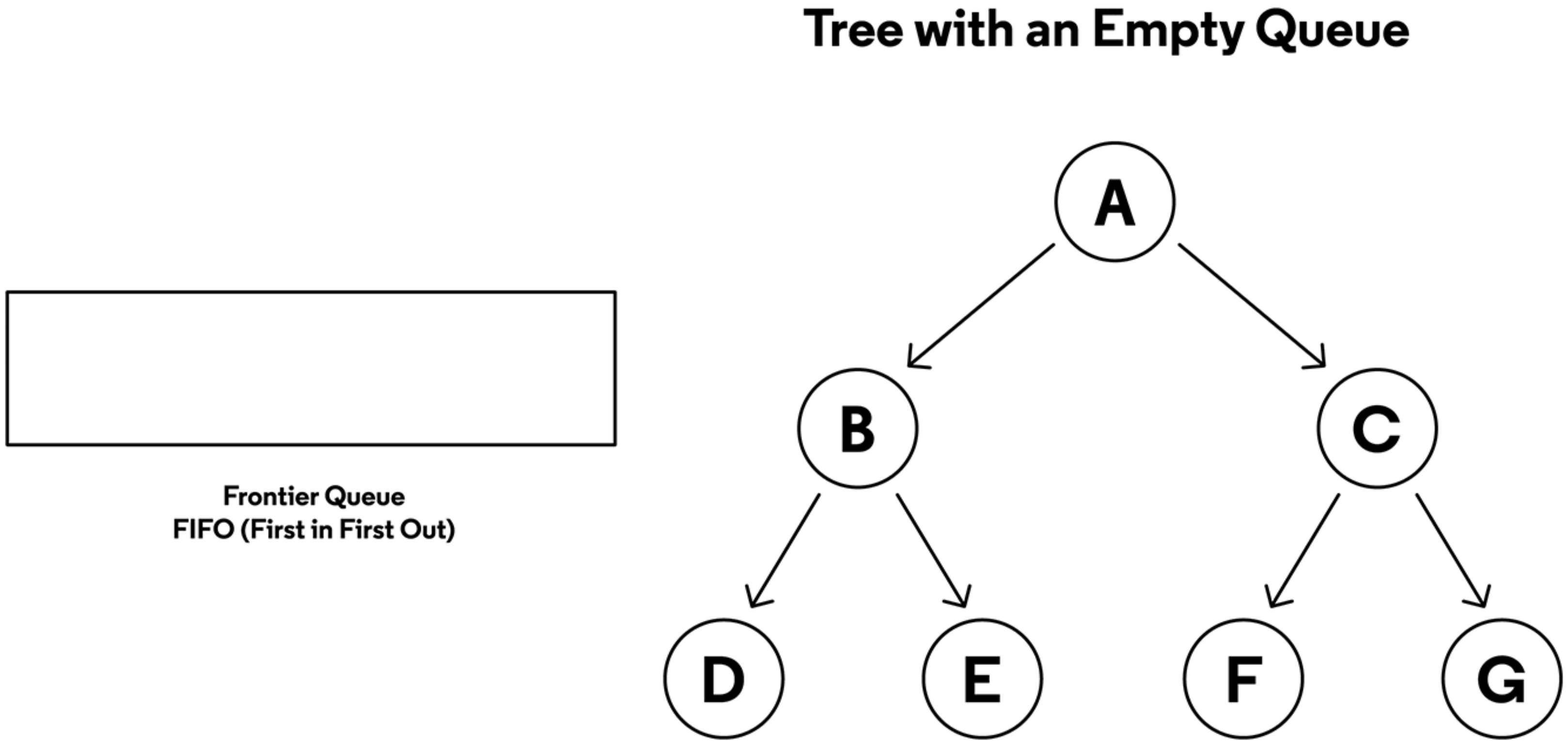
Breath First Traversal Algorithms

```
Add root to queue
while queue is not empty:
        node n = deque()
        operate on n // in our case print or concat n
        for child c of n:
                enqeue(n)
```

Breath First Search Algorithm

```
Add root to queue
while queue is not empty:
    node n = deque()
    if n is the target:
        stop
    for child c of n:
        enqeue(n)
```

Breath First Search in action



https://www.codecademy.com/article/tree-traversal

Outline

- Array Based Trees
- Breath First Traversal
- **Array Based Heaps**
- More efficient way to Construct Heaps

Helper methods for Array-based binary heap

```
int parent(int i);
int leftChild(int i);
int rightChild(int i);
void swap(int i, int j);
int containsIdx(E element);
string toStringInORderRec(int root);
```

PriorityQueue Interface

```
public interface PriorityQueue<E extends Comparable<E>>
extends BinaryTree<E> {
   E getRootElement();
   int size();
   boolean isEmpty();
   boolean contains(E element);
   void insert(E element);
   boolean remove(E element);
   String toStringInOrder();
   String toStringPreOrder();
   String toStringPostOrder();
   E peek();
   E poll();
}
```

Heap-based Priority Queue

```
public class ArrayHeap<E extends Comparable<E>>
extends ArrayBinaryTree<E> implements
PriorityQueue<E>{
  //instance variables?
  //constructor, getters, setters
  //inherited methods
  E peek();
  E poll();
}
```

Inherited Methods from ArrayBinaryTree

```
E getRootElement();
int size();
boolean isEmpty();
boolean contains(E element);
void insert(E element);
boolean remove(E element);
String toStringInOrder();
String toStringPreOrder();
String toStringPostOrder();
```

Inherited Methods from ArrayBinaryTree

```
E getRootElement();
int size();
boolean isEmpty();
boolean contains(E element);
void insert(E element);
boolean remove(E element);
String toStringInOrder();
String toStringPreOrder();
String toStringPostOrder();
```

Updating Key (Priority of an element)

What should happen when you change the key of an existing element in a heap?

What are the cases?

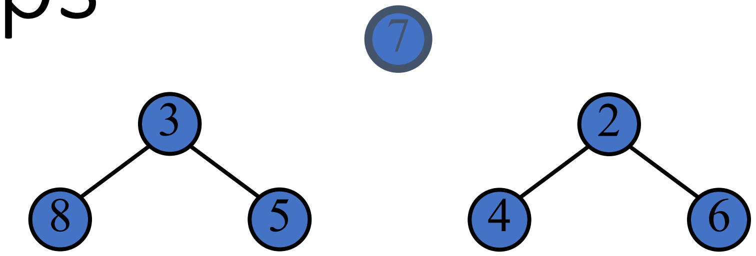
- increaseKey
- decreaseKey

Outline

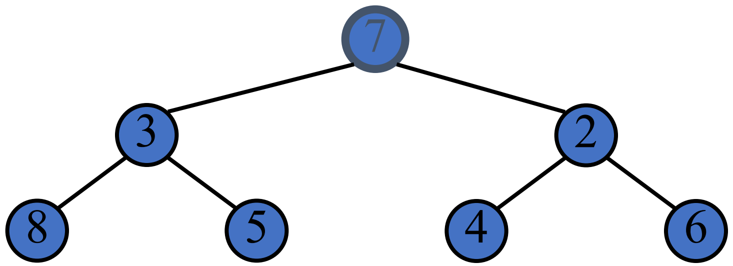
- Array Based Trees
- Breath First Traversal
- Array Based Heaps
- **More efficient way to Construct Heaps**

Merging Two Heaps

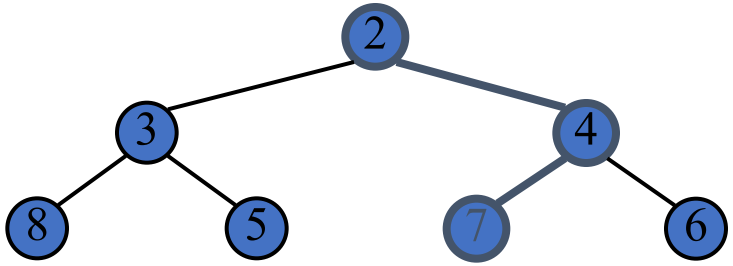
- We are given two two heaps and a key k



- We create a new heap with the root node storing k and with the two heaps as subtrees



- We perform downheap to restore the heap-order property



Inserting n elements in a Heap (Construction)

Time complexity of making a heap with *n* elements:

- Call insert n times:
  - *O(nlog(n))*

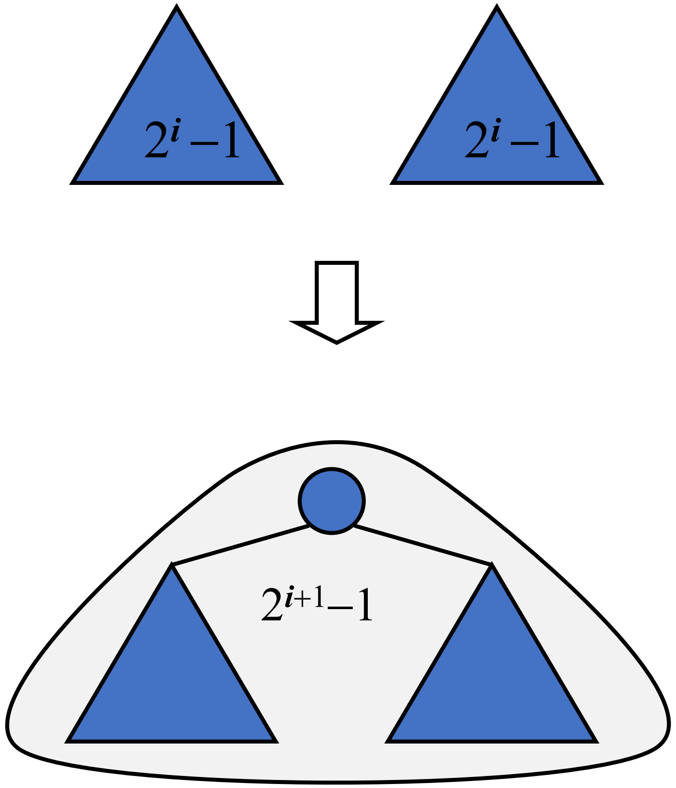
More efficient alternative:

- merge & conquer!
- Bottom-up approach
  1. construct(n +1)/2elementary heaps storing one entry each
  2. merge pairwise into (! + 1)/4 larger heaps

Bottom-up Heap Construction

- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases

- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



Example

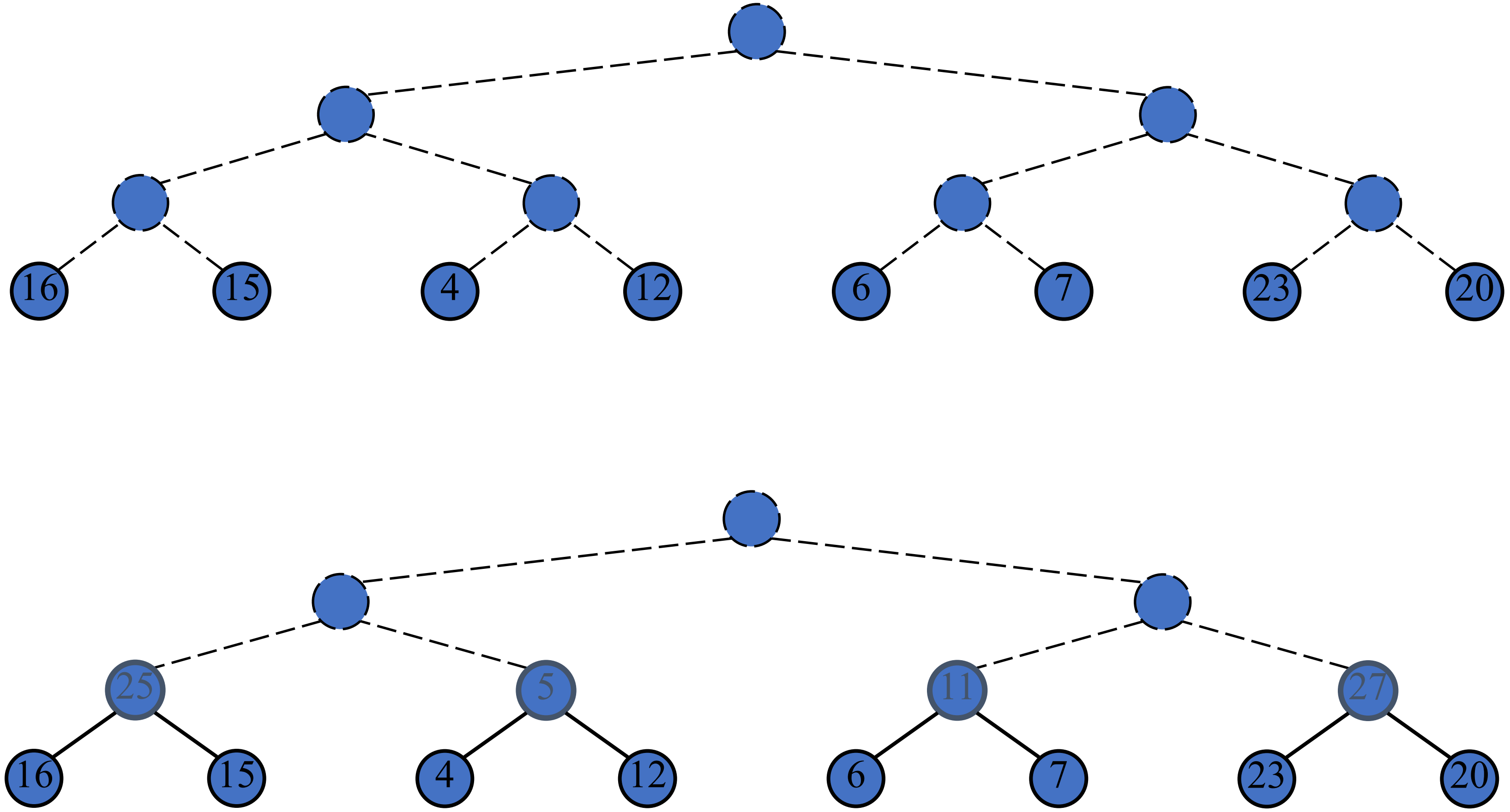
Lets insert 15 numbers: 4, 16, 25, 7, 15, 10, 12, 8, 11, 9, 6, 27, 23, 20, 5

What will the height of the tree be?

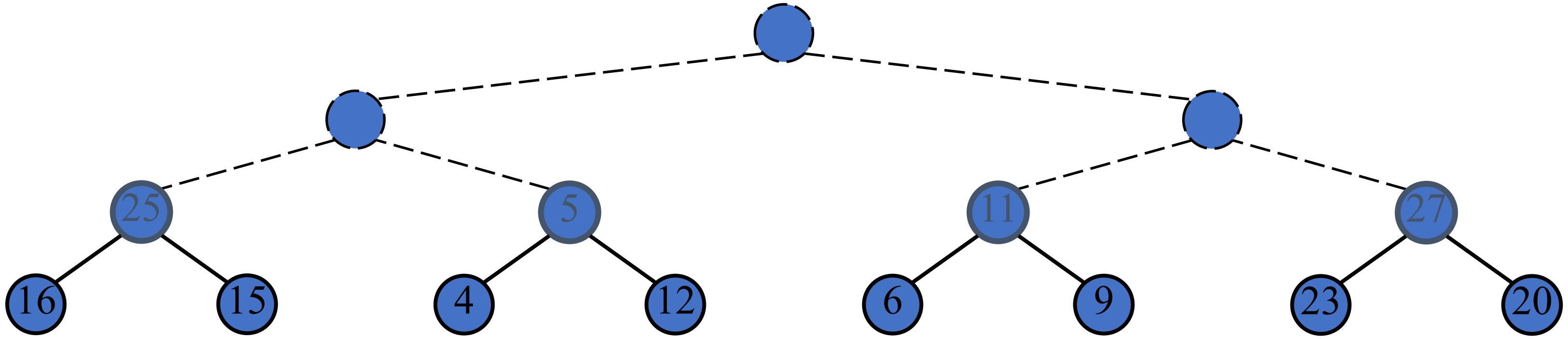
Whats the first step:

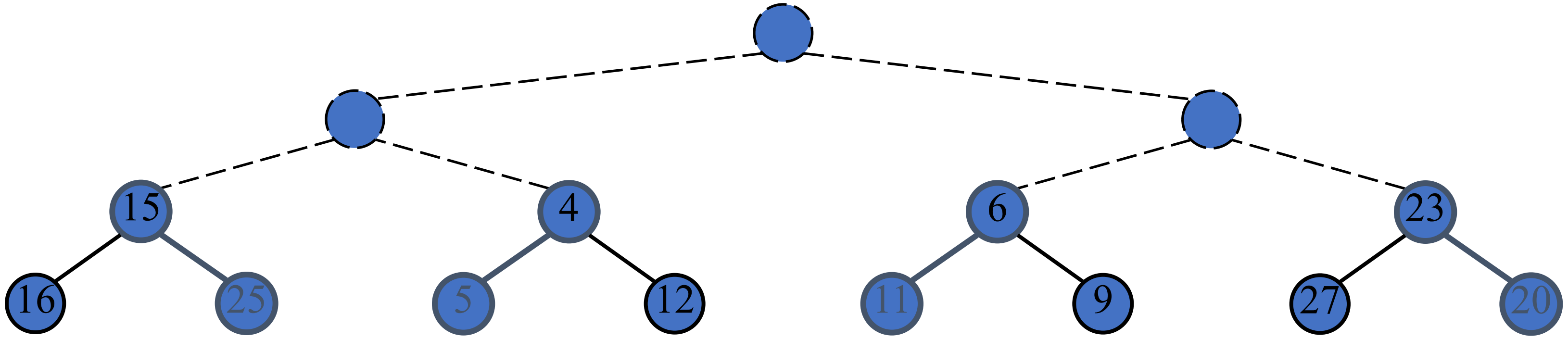
Bottom up:

Example

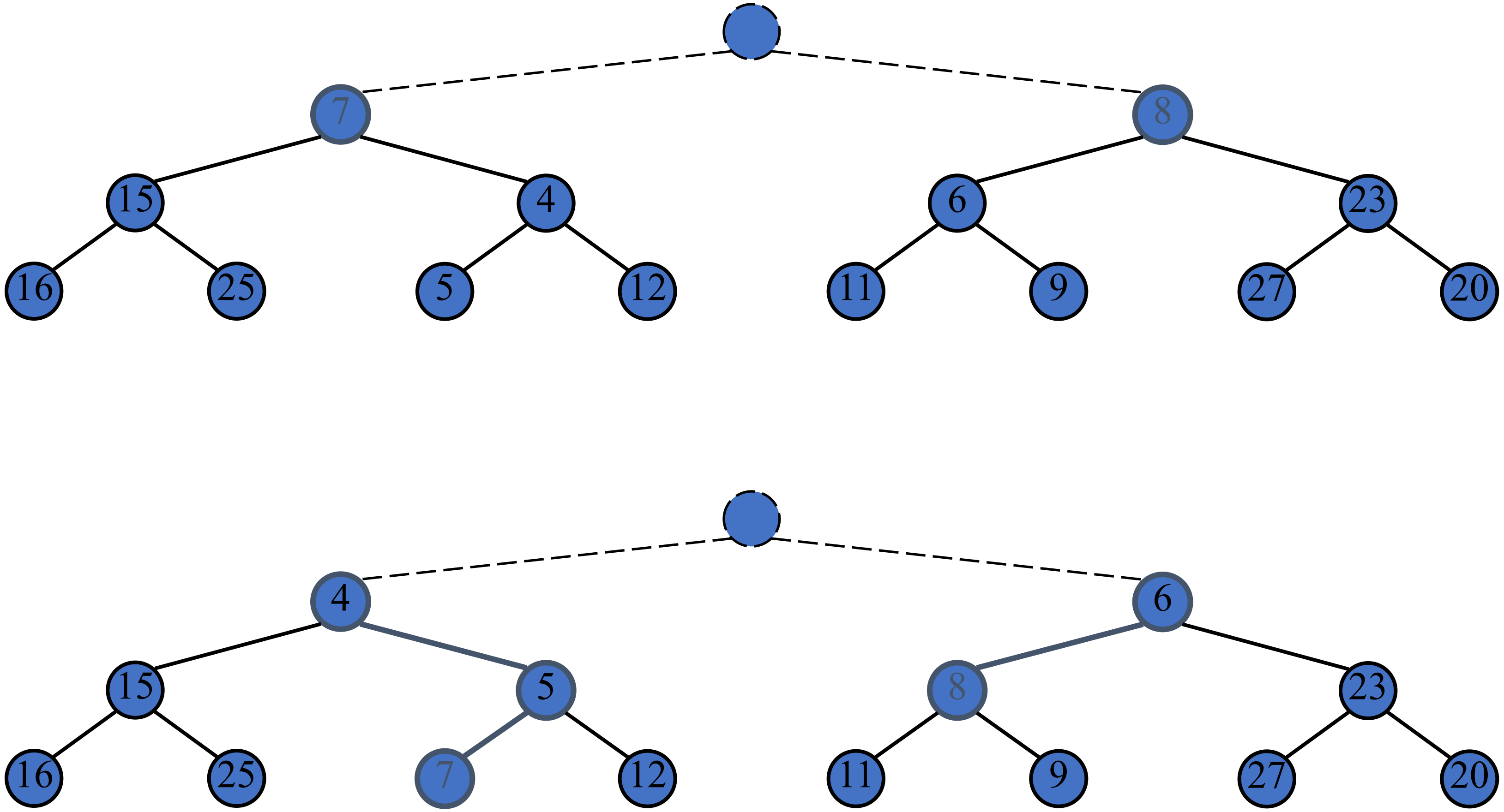


Example (contd.)

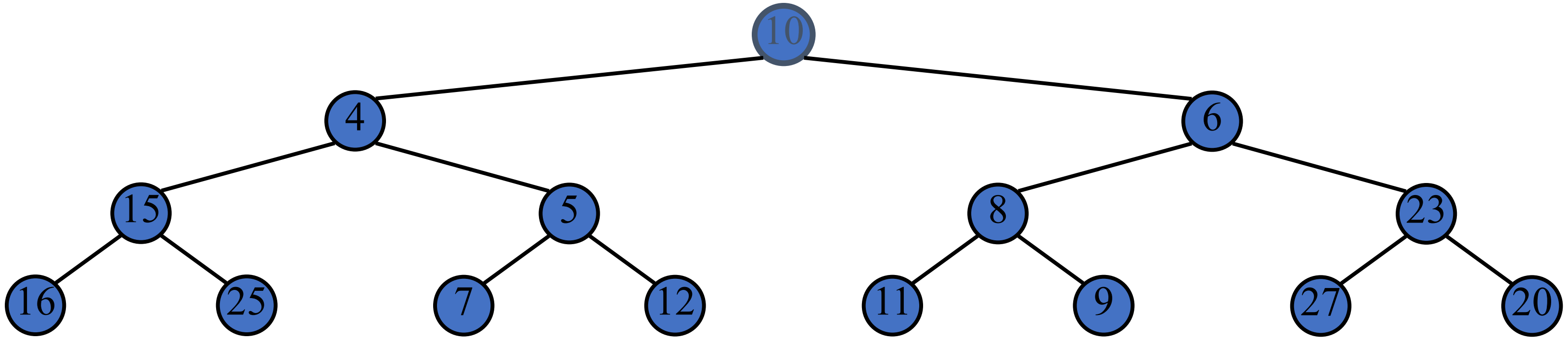


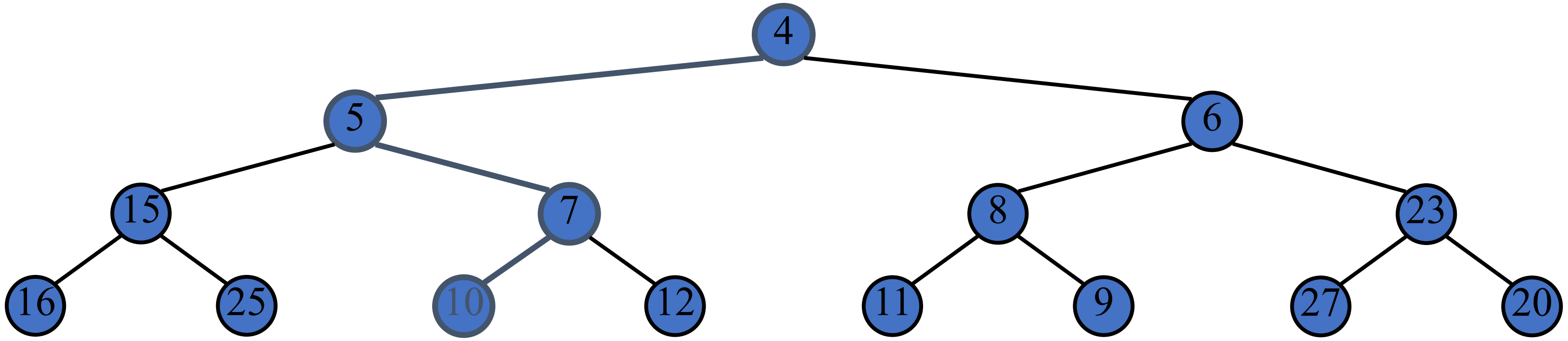


Example (contd.)



Example (end)





Runtime Analysis

n/4 + n/8 + n/16 + … +

1 = O(n) merges

Each merge is O(log n) (because we need to fix heap order when merging)

Runtime Analysis

- Complexity of merge depends on the height of the tree - generally O(log(n))
- height only reaches full O(log(n)) on the last merge. All other heaps are shorter
- In fact, half of the heaps have height 0
- Tighter analysis (with a fair bit more math) shows O(n)