Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Heap

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Admin

- Next week Mock test

Topics

- Partially ordered trees.
- Partially ordered trees: add.
- Partially ordered trees: remove.
- Heap.
- Heap: add/remove.
- Heap queue.
- Heap queue: offer and poll.
- Heap queue: push up.
- Heap queue: push down.
- Heap analysis.

Partially Ordered Trees

- **Implementing Priority Queues**
- Highest priority in the queue is at the front of the queue.
- If the highest priority is removed from the queue, the next one in the queue becomes the highest priority in the queue.
- Move the first item in the queue i.e. move the highest priority item in the queue.

- The subsequent item in the queue shift to the front i.e. move the second highest priority item to become the highest priority in the queue.

Partially Ordered Trees: Application

1. Operating System Process Scheduling
Operating systems must decide which process should run next.
A **priority queue implemented as a heap** can store processes:
The process with the highest priority is always at the root and can be selected quickly.

Priority	Process
100	Antivirus
80	Web Browser
60	Music Player

2. Hospital Emergency Triage
Patients are treated based on urgency rather than arrival time.
A max heap allows the hospital system to always retrieve the most urgent patient first.

Example:

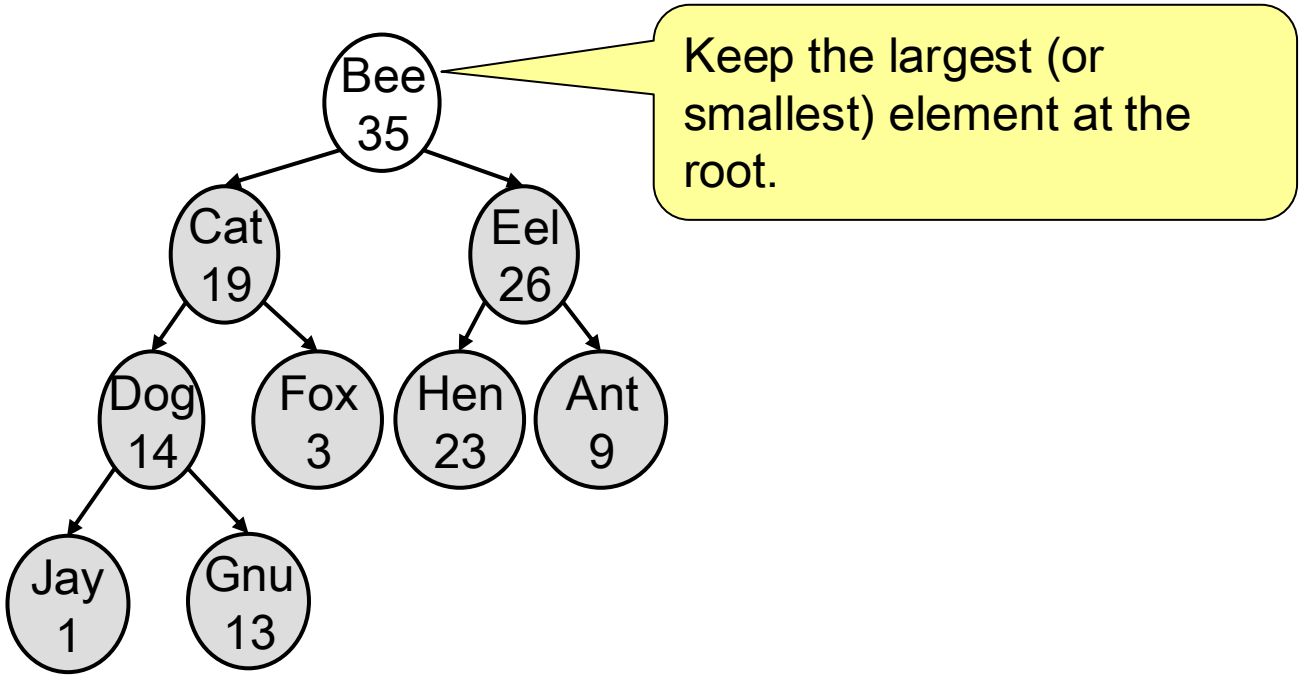
Priority	Patient
10	Heart Attack
8	Broken Leg
3	Flu

3. Path-Finding Algorithms
Algorithms such as:
•Dijkstra's Algorithm
•A* Search Algorithm
use heaps to efficiently select the next node with the smallest cost.

Applications:
•GPS navigation
•Maps

Partially Ordered Trees

- **Implementing Priority Queues efficiently with Partially Ordered Tree**
- Binary tree
- Children ≤ parent,
- Order of children is not important

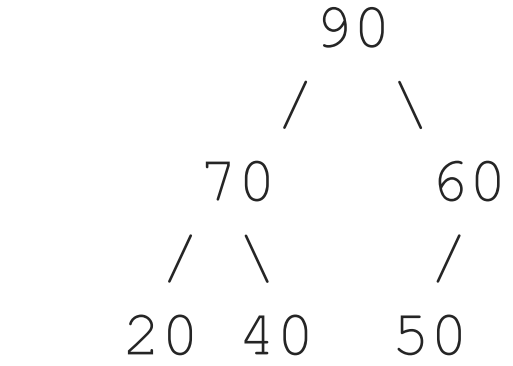


Partially Ordered Trees

- A **partially ordered tree** is a tree in which the nodes are arranged according to a **parent-child ordering rule**, but the entire tree is **not completely sorted**.
- Heaps (Max Heaps and Min Heaps) are the most common example of partially ordered trees.

Max Heap (Partially Ordered Tree)
In a Max Heap:
•Every parent node is **greater than or equal to** its children.
•There is **no ordering requirement** between siblings or nodes in different branches.
Check the ordering:
90 > 70 and 60
70 > 20 and 40
60 > 50
The heap property is satisfied.

Example:

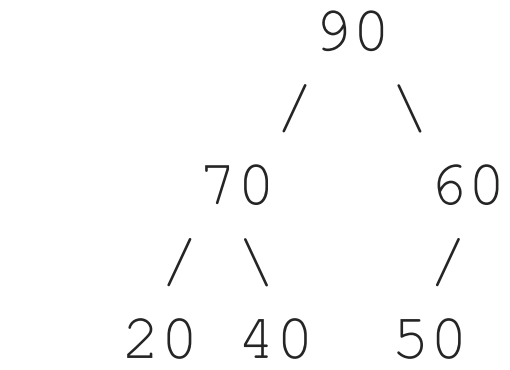


Partially Ordered Trees

Only the parent-child relationships must follow the ordering rule.
Therefore, the tree is:

- **Partially ordered** → some relationships are ordered.
- **Not fully ordered** → not every node is compared with every other node.

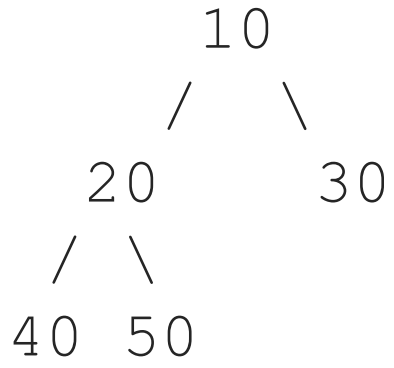
Example:



Partially Ordered Trees

Min Heap Property
A Min Heap satisfies:
Parent ≤ Left Child
Parent ≤ Right Child
Also a partially ordered tree.

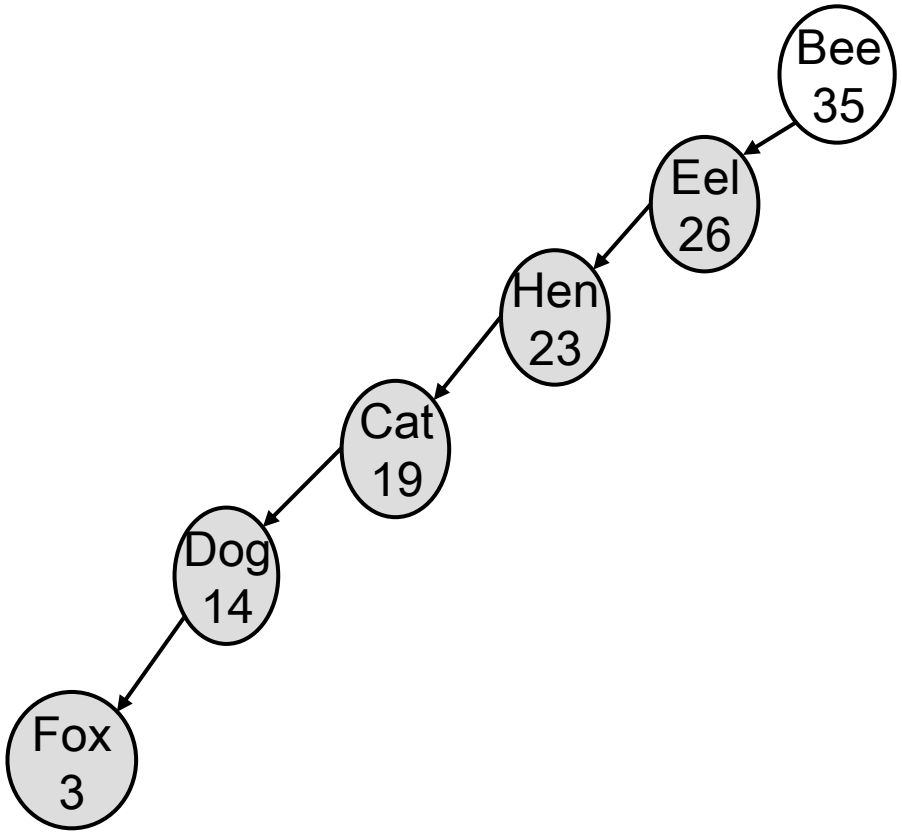
Example:

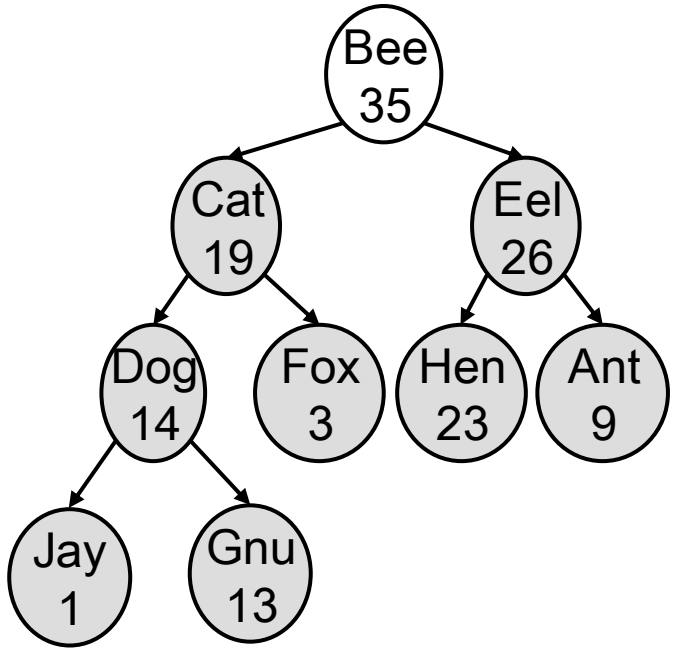


Partially Ordered Trees

Difference between Non-Ordered Tree vs. Ordered Tree:

- Complexity of non-ordered tree is O(n).
- Complexity of ordered tree is O(log(n)).





Partially Ordered Trees: Referencing nodes

- It will be necessary to find the indices of the parents and children of nodes in a heap's underlying array
- The children of a node *i*, are the array elements indexed at

$$2i+1 \text{ and } 2i+2$$

- The parent of a node *i*, is the array element indexed at floor

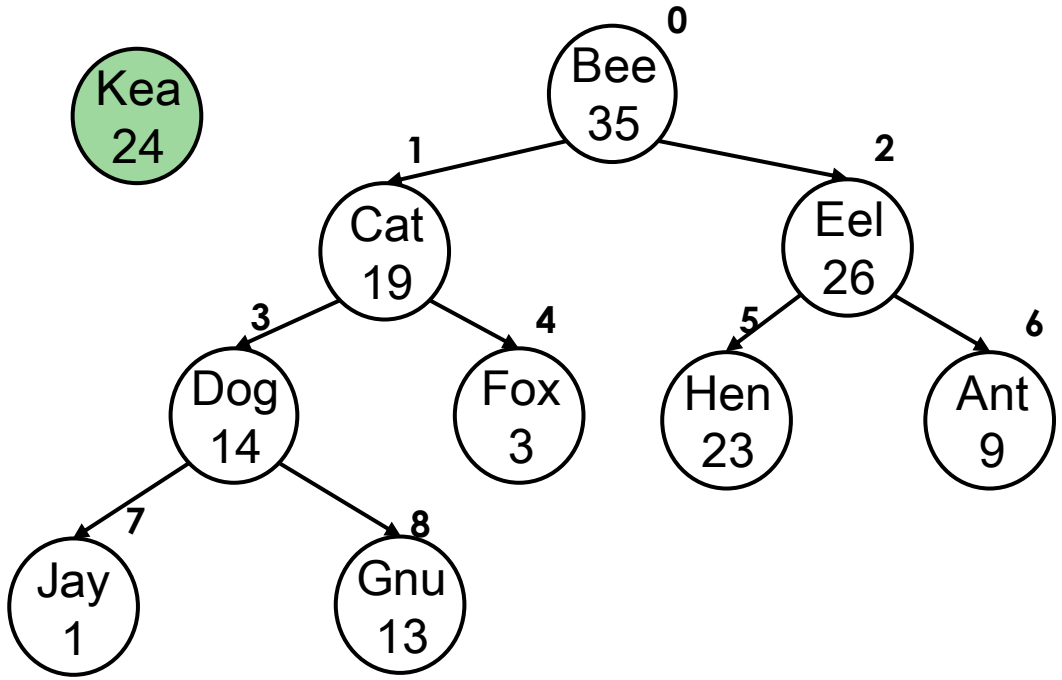
$$\lfloor (i-1)/2 \rfloor$$

Helping methods

```
private int parent(int i)
{ return (i-1)/2; }
private int left(int i)
{ return 2*i+1; }
private int right(int i)
{ return 2*i+2; }
```

Partially Ordered Tree: add

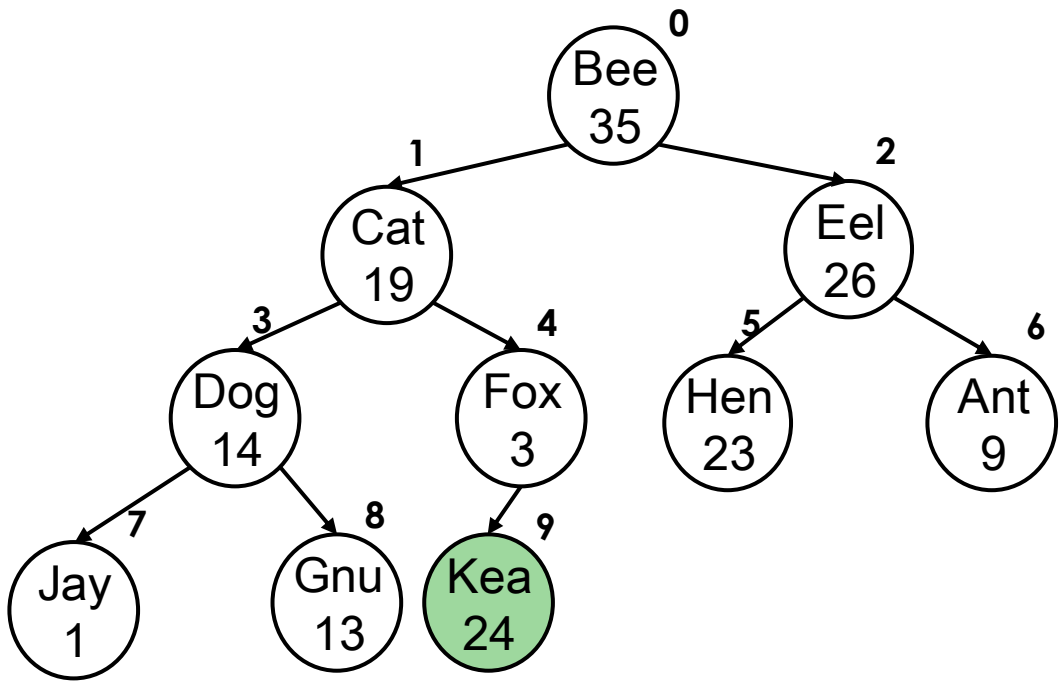
- Easy to add and remove because the order is not complete.
- Add:
  - insert at several potential ways:
    - Underneath Fox.
    - Underneath Hen.
    - Underneath Ant.



index	0	1	2	3	4	5	6	7	8
value	35	19	26	14	3	23	9	1	13

Partially Ordered Tree: add

- Easy to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - "push up" to correct position. (swapping)

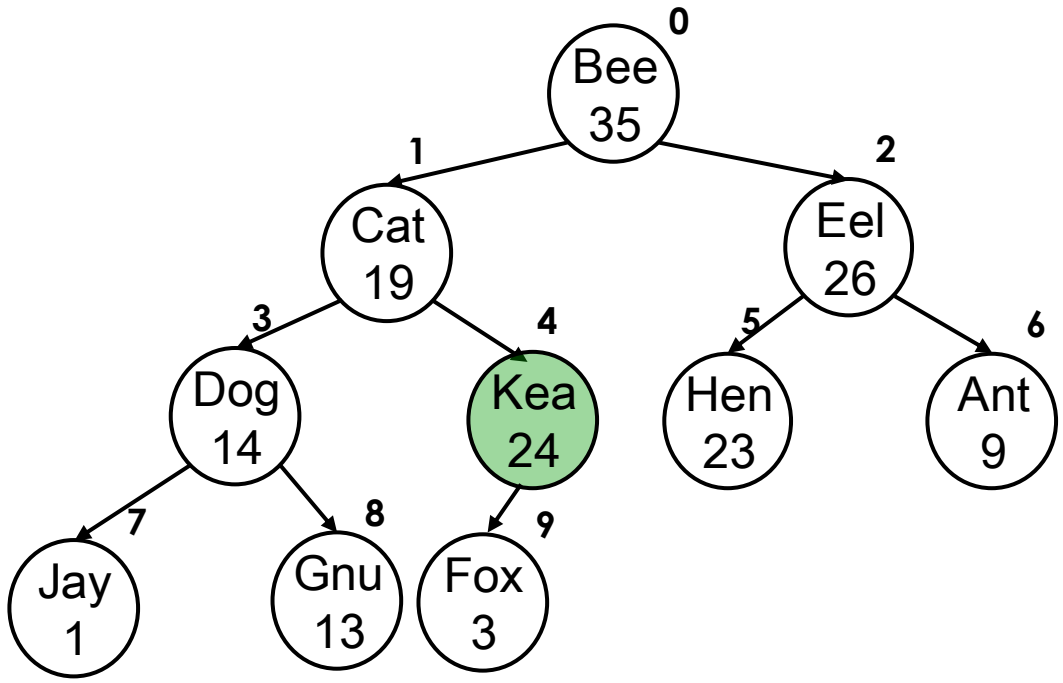



```
(9-1)/2 = 4
```

index	0	1	2	3	4	5	6	7	8	9
value	35	19	26	14	3	23	9	1	13	24

Partially Ordered Tree: add

- Easy to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - "push up" to correct position.
    (swapping)



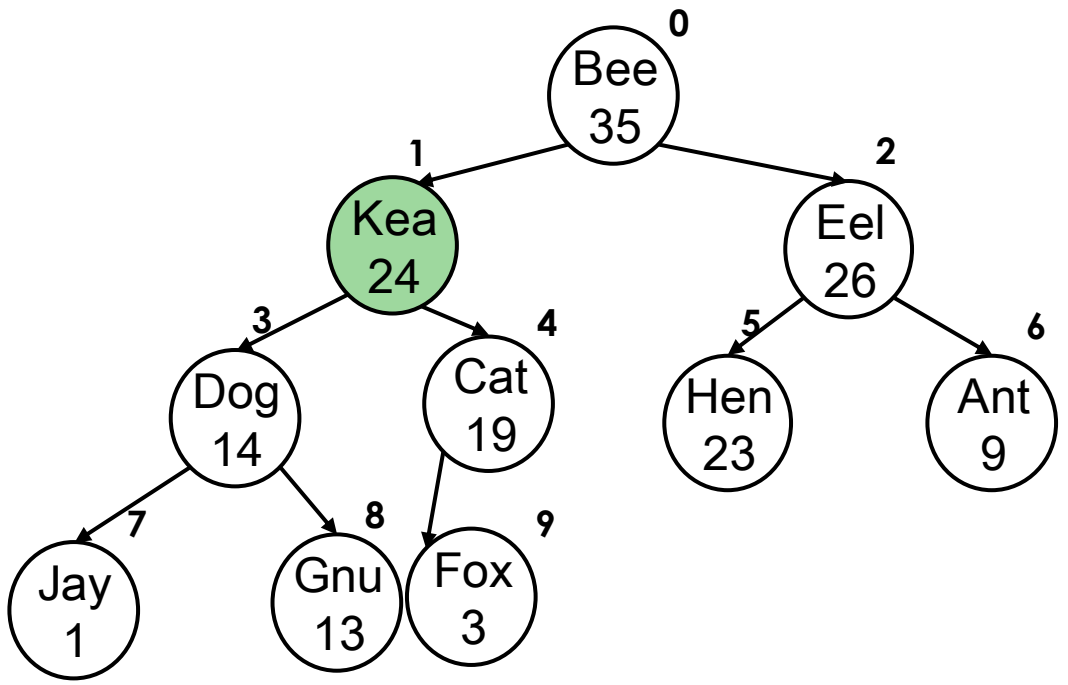
`(4-1)/2 = 1`

`(9-1)/2 = 4`

index	0	1	2	3	4	5	6	7	8	9
value	35	19	26	14	24	23	9	1	13	3

Partially Ordered Tree: add

- Easy to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - "push up" to correct position.
    (swapping)



(4-1)/2 = 1

(9-1)/2 = 4

index	0	1	2	3	4	5	6	7	8	9
value	35	24	26	14	19	23	9	1	13	3

Partially Ordered Tree: Implementation

```
public class Heap<T extends KeyedItem> {
      private int HEAPSIZE=200;
      // max. number of elements in the heap
      private T items[]; // array of heap items
      private int num_items; // number of items

      public Heap() {
            Items = new T[HEAPSIZE];
            num_items=0;
      } // end default constructor
```

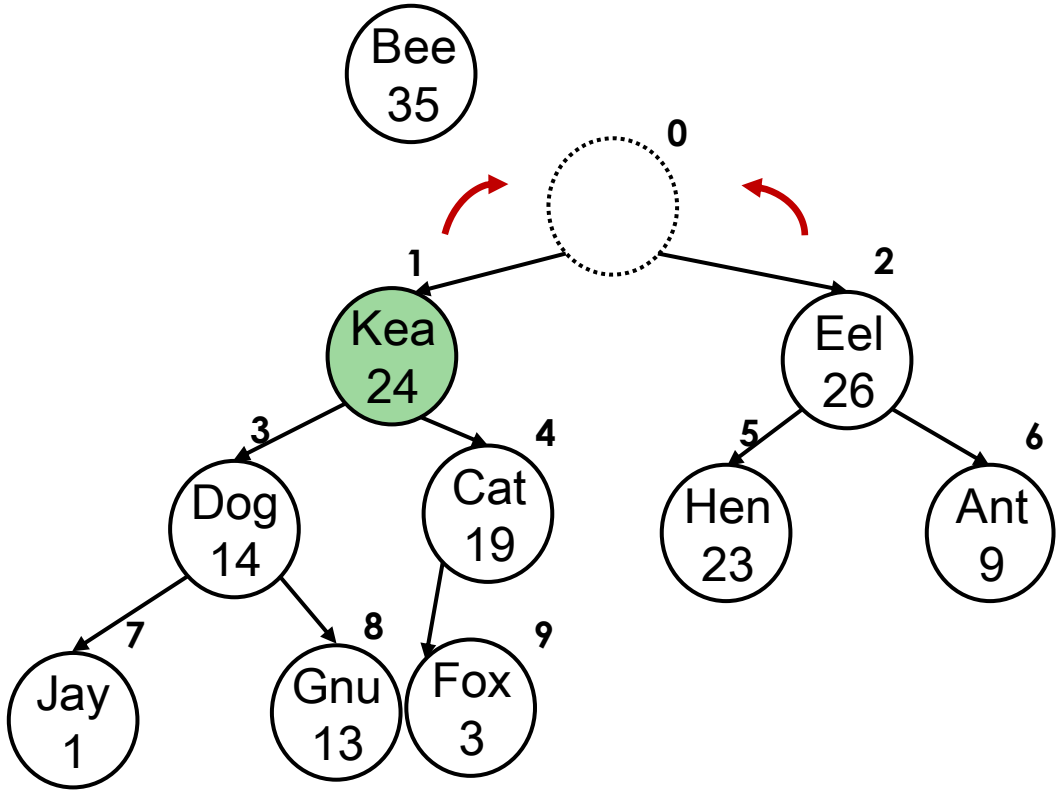
Let's assume that the priority of an element is equal to its key.
So the elements are partially sorted by their keys.
The element with the biggest key has the highest priority.

Partially Ordered Tree: add

```
public void insert(T newItem) {
	// TODO: should check for the space first
	num_items++; // increment the total number of items
	int child = num_items-1; // assigning the new item to the last order
inside the variable called child
	while (child > 0 && item[parent(child)].getKey() < newItem.getKey()){
		items[child] = items[parent(child)];
		child = parent(child);
	}
	items[child] = newItem;
}
```

Partially Ordered Tree: remove

- Remove:
  - "pull up" the largest child of the root and recurse on the subtree.
  - But: makes the tree unbalanced!



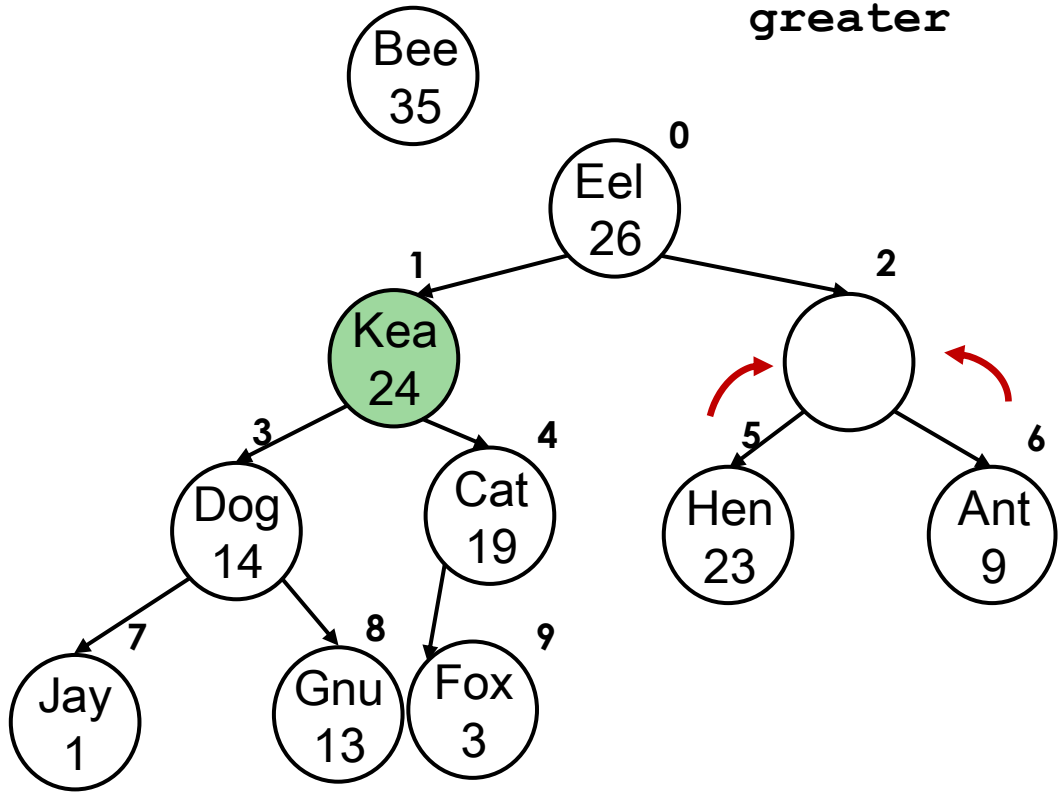

```
children of root: 2*0+1, 2*0+2 = 1, 2
```

index	0	1	2	3	4	5	6	7	8	9
value	35	24	26	14	19	23	9	1	13	3

Partially Ordered Tree: remove

- Remove:
  - "pull up" the largest child of the root and recurse on the subtree.
  - But: makes the tree unbalanced!

`right child is greater`

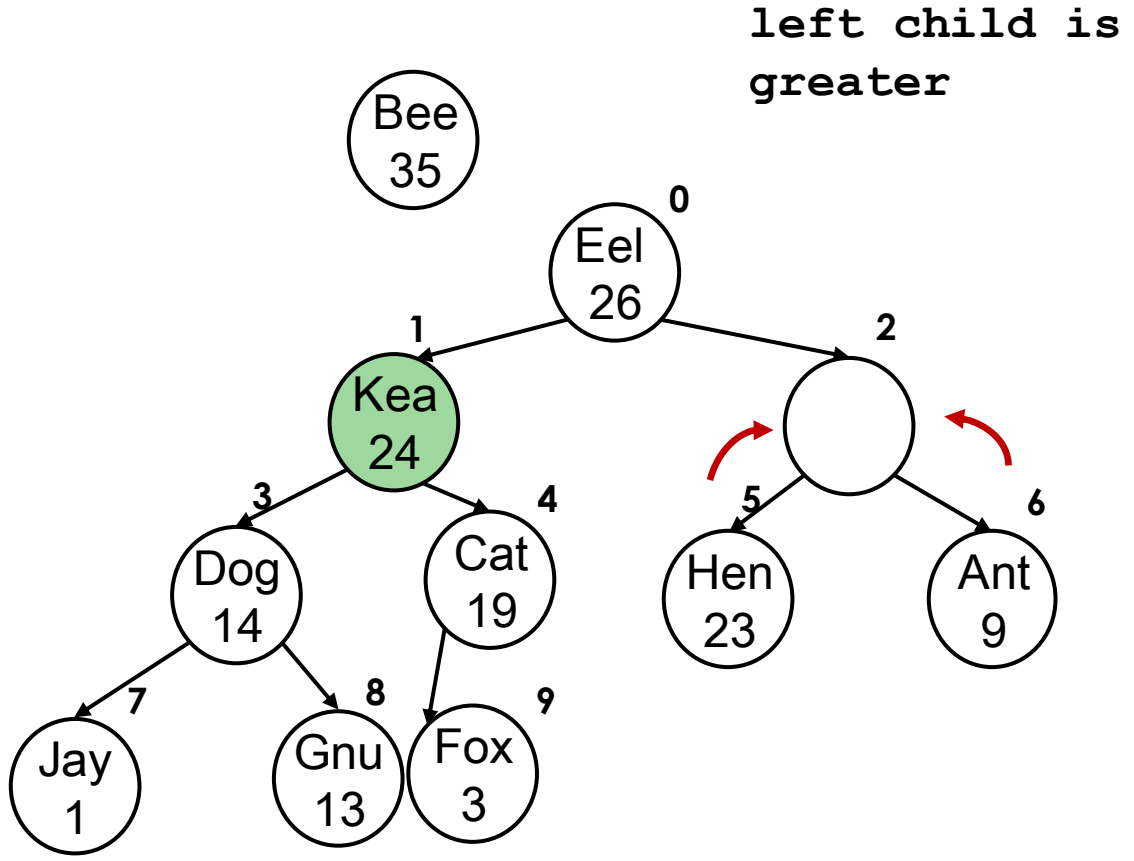


`children of root: 2*0+1, 2*0+2 = 1, 2`

index	0	1	2	3	4	5	6	7	8	9
value	35	24	26	14	19	23	9	1	13	3

Partially Ordered Tree: remove

- Remove:
  - "pull up" the largest child of the root and recurse on the subtree.
  - But: makes the tree unbalanced!



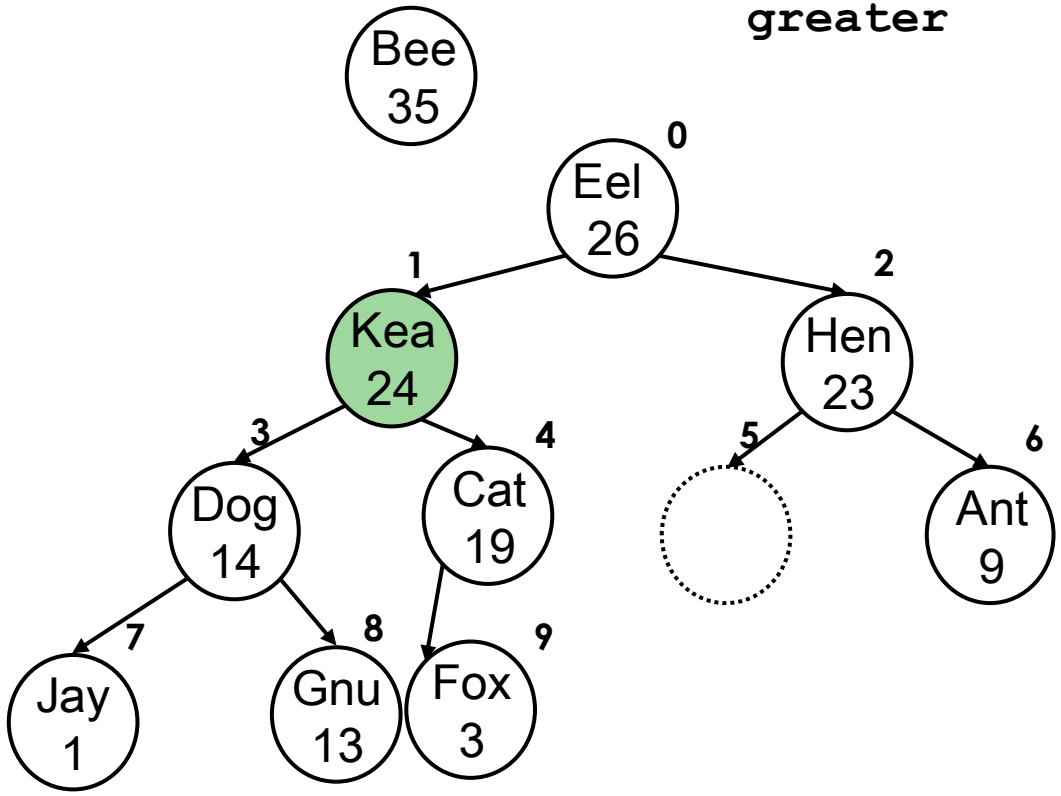

```
children of root: 2*2+1, 2*2+2 = 5, 6
```

index	0	1	2	3	4	5	6	7	8	9
value	26	24		14	19	23	9	1	13	3

Partially Ordered Tree: remove

- Remove:
  - "pull up" the largest child of the root and recurse on the subtree.
  - But: makes the tree **unbalanced!**

left child is greater



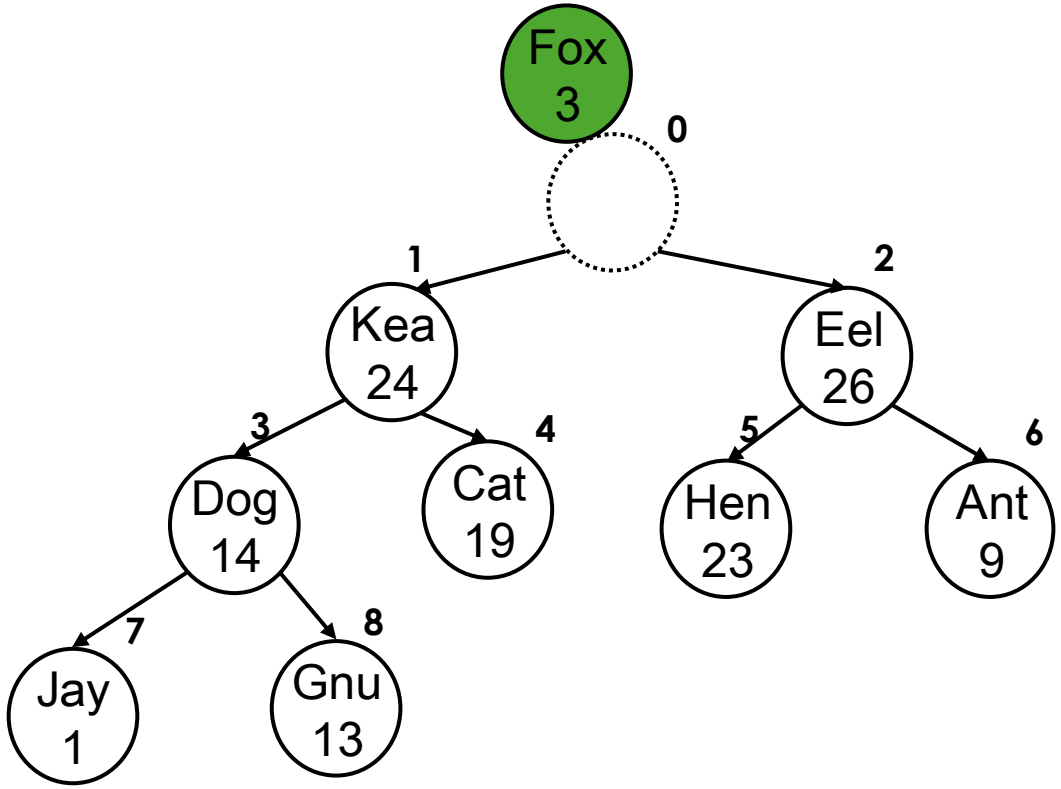
index	0	1	2	3	4	5	6	7	8	9
value	26	24	23	14	19	0	9	1	13	3

Partially Ordered Tree: remove I

- Remove:
  - "pull up" largest child of root and recurse on that subtree.
  - But: makes tree unbalanced!

Alternative:

- Replace root by the bottom rightmost node
- "push down" to correct position (swapping)
- keeps tree balanced – and complete!

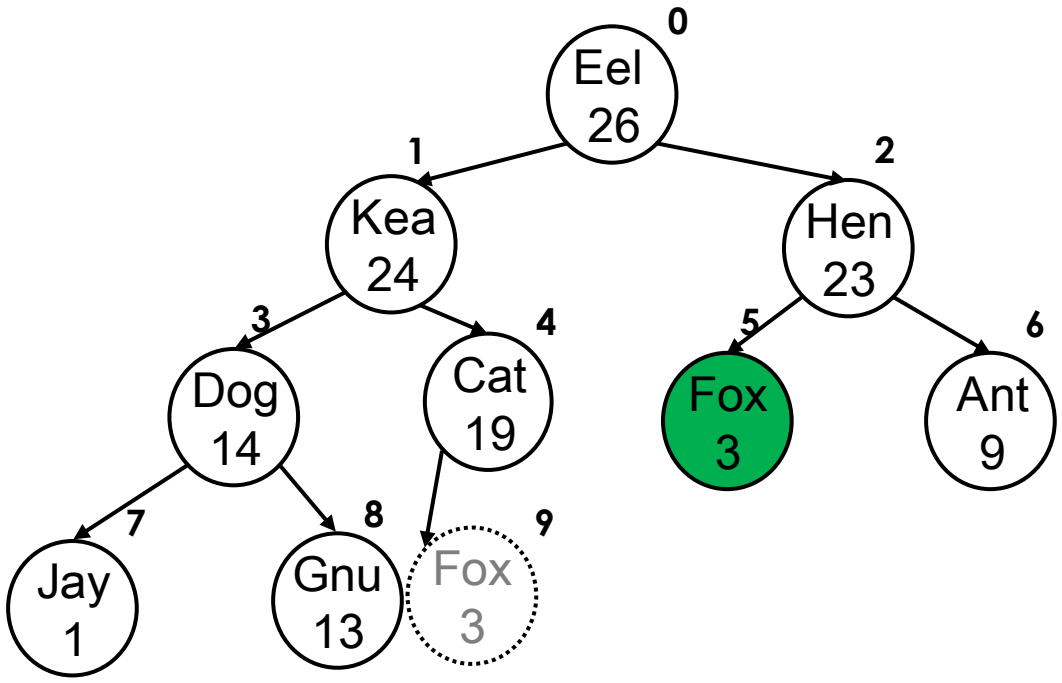


Partially Ordered Tree: remove II

- Easier to add and remove because the order is not complete.
- Add:
  - insert at bottom right
  - “push up” to correct position.
- Remove:
  - “pull up” largest child and recurse.
  - But: makes tree unbalanced!

Alternative:

- replace root by bottom rightmost node
- “push down” to correct position
- keeps tree balanced – and complete!



Partially Ordered Trees: Conclusion

For adding and removing in a partially ordered tree:

- Add: insert at bottom rightmost,
  swap with parent, …
- Remove: replace root with bottom rightmost,
  swap with the largest child, …

Partially Ordered Trees

But:

- How do you find the bottom right?
- Once you have found it, how do you find its parent to push it up?

We need a tree where you can quickly get to:

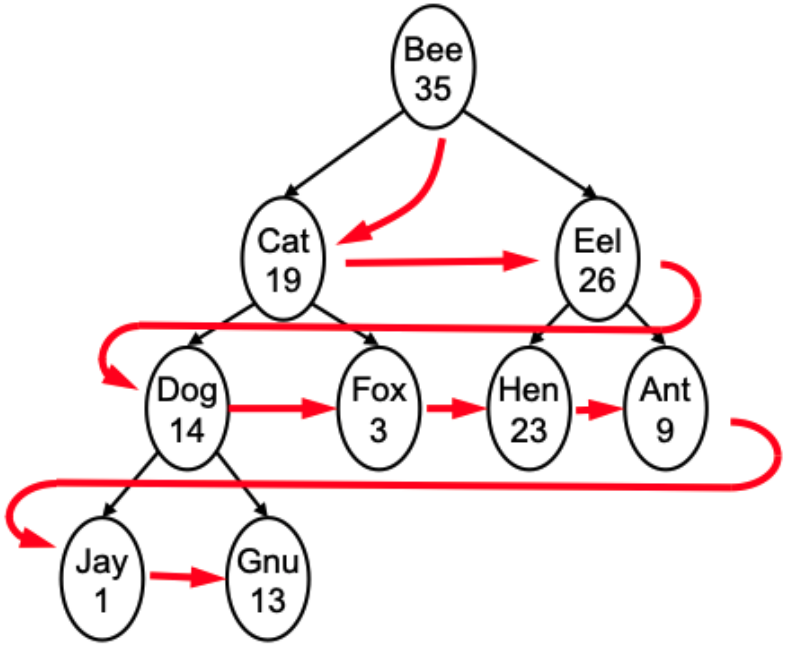
- the bottom right node,
- children from parent,
- parent from children.

Partially Ordered Trees

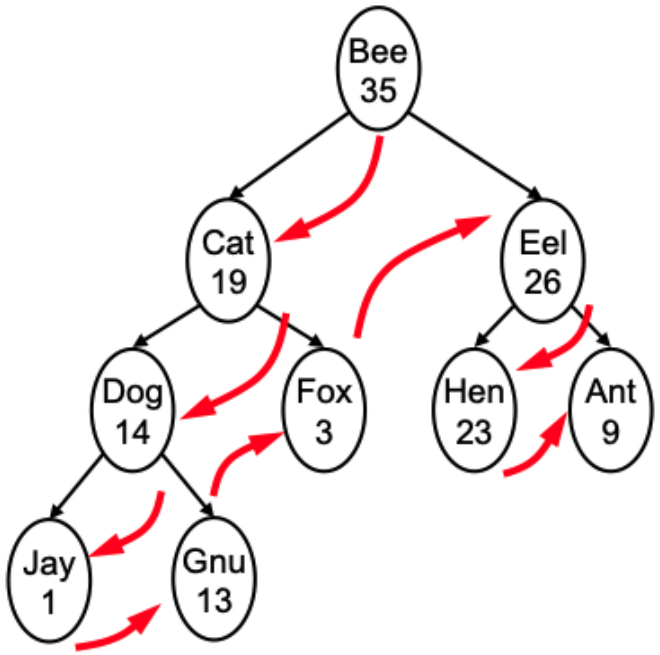
Tree traversing algorithm:

- Breadth first.
- depth first.

Option 1: breadth first.



Option 2: depth first.



Partially Ordered Trees

Abstract Data Type (ADT) to work with traversing of the tree:

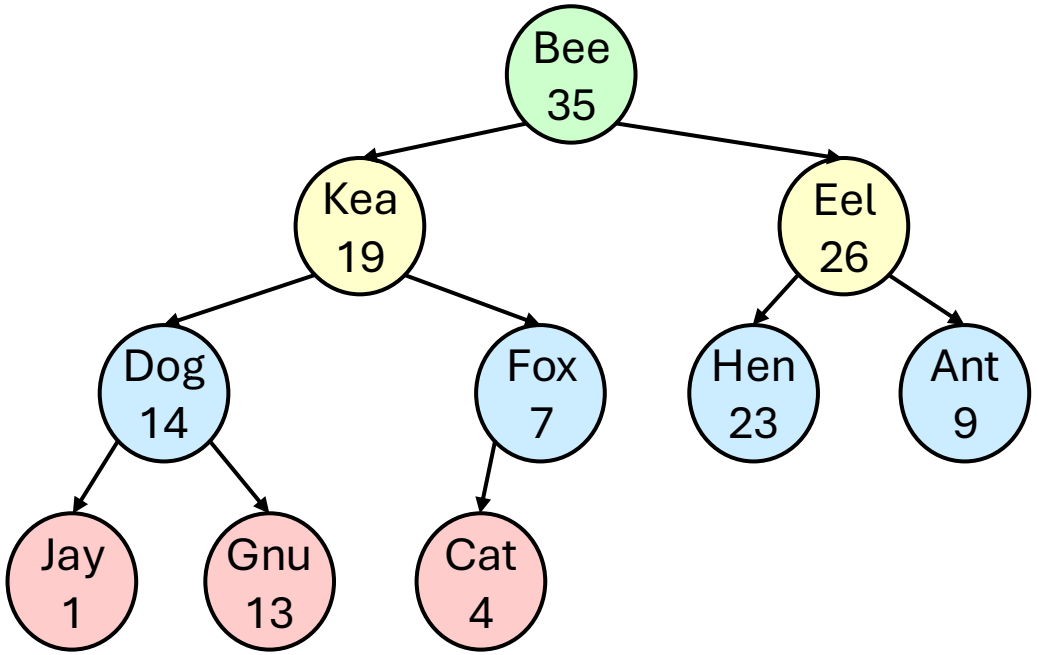
- Array.
- ArrayList.
- Queue.
- Linked list.
- Stack.
- Etc.

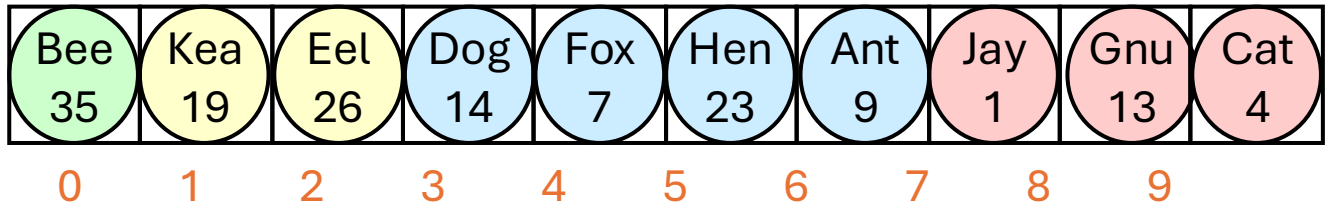
Consideration for best option:

- Simple algorithm.
- Space for storing.
- Processing time and resources.

Heap

- A complete, partially ordered, **binary tree**
  complete = every level full, except bottom, where nodes are to the left
- Implemented in an array using breadth-first order





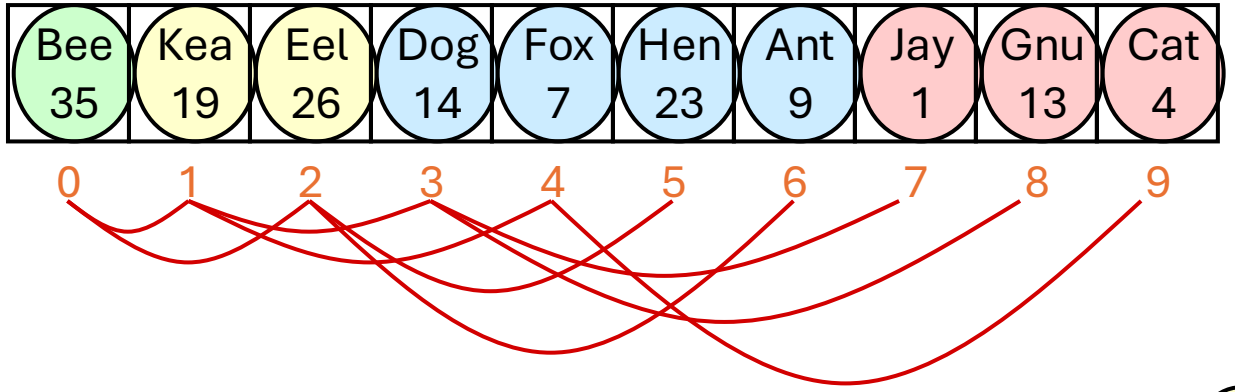
Item Order:
0 1, 2
1 3, 4
2 5, 6
… …, …
i (2i+1),(2i+2)

If we know parent (i), we can find child:
(2i+1),(2i+2)

If we know child (j), we can find parent:
(j-1)/2j

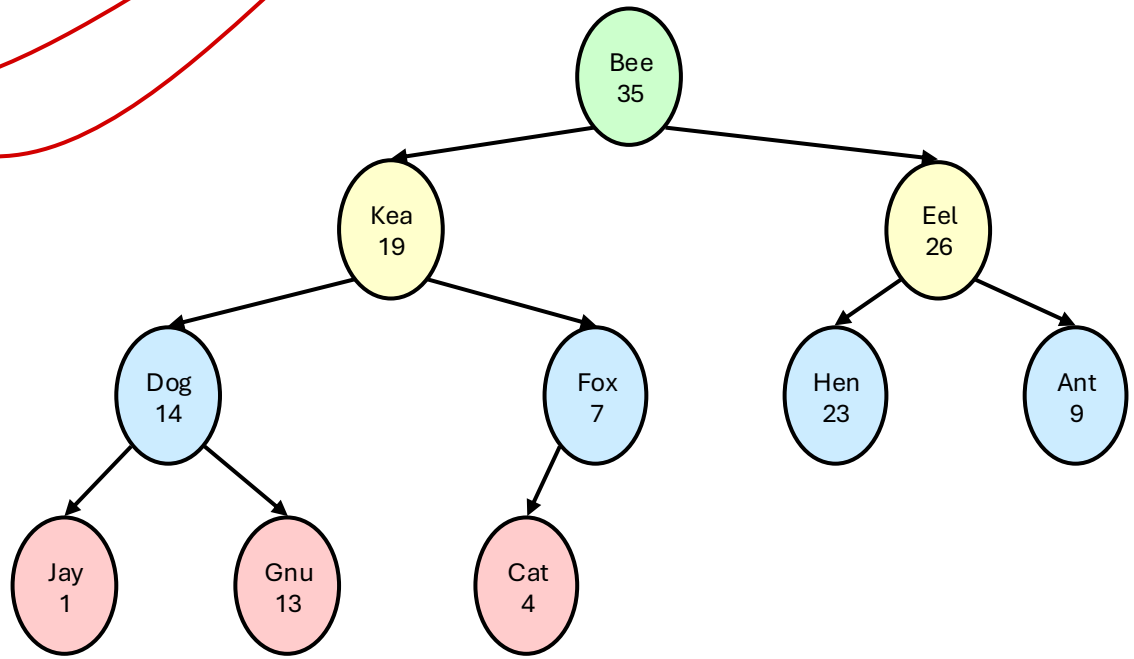
Heap

- We can **compute the index** of parent and children of a node:
  - the children of node i are at (2i+1) and (2i+2)
  - the parent of node i is at (i-1)/2

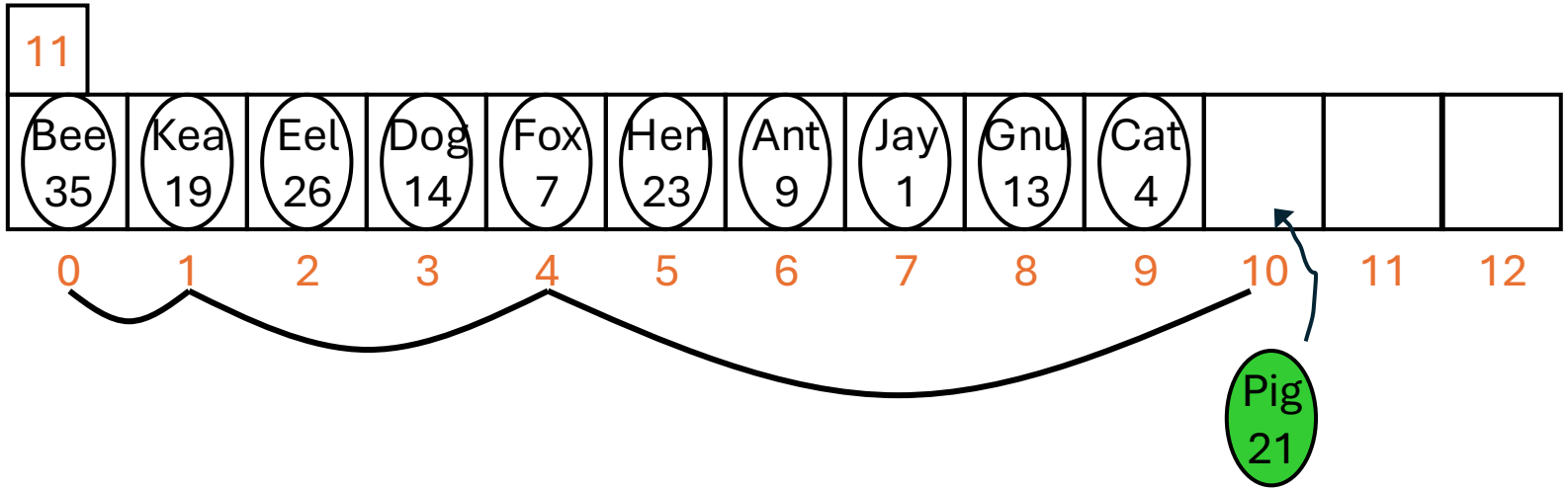


- Bottom right node is last element used.

- There are no gaps!



Heap: add

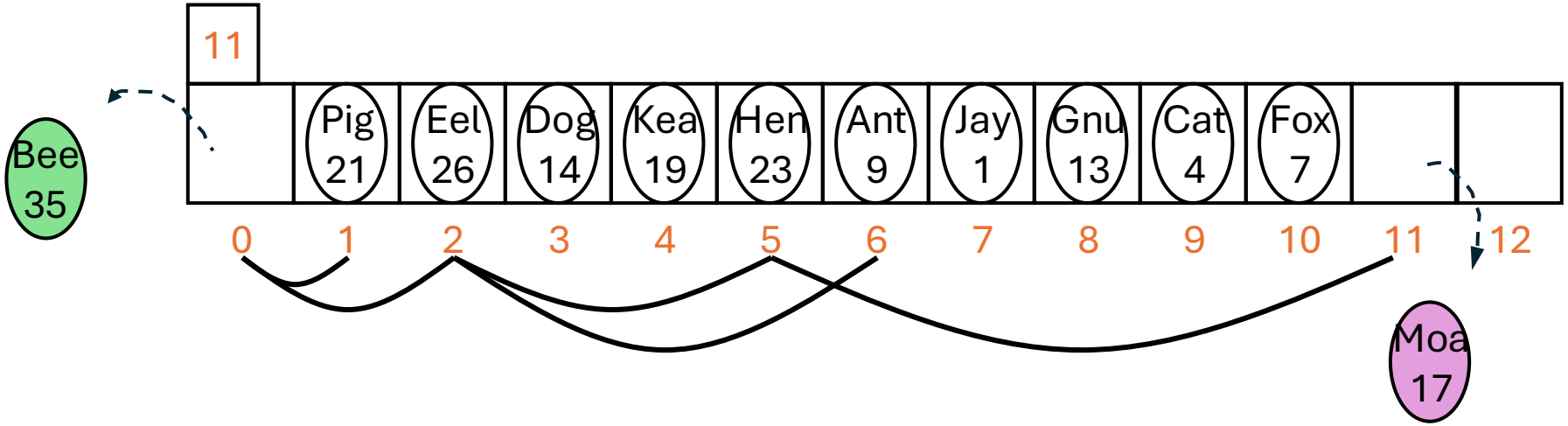


Insert at bottom of tree and push up:

- Put new item at end: 10
- Compare with parent: (10-1)/2 = 4 ⇒ Fox/7
  - If larger than parent, swap
- Compare with parent: (4-1)/2 = 1 ⇒ Kea/19
  - If larger than parent, swap
- Compare with parent: (1-1)/2 = 0 ⇒ Bee/35

(j-1)/2

Heap: remove



- Remove item at 0:
- Move the last item to 0
- Find largest child $2\times0+1 = 1,\ 2\times0+2 = \mathbf{2}$
  - If smaller than the largest child, swap
- Find largest child $2\times2+1 = 5,\ 2\times2+2 = 6$
  - If smaller than the largest child, swap
- Find largest child $2\times5+1 = 11$: No such child

HeapQueue

A class that provides a priority queue interface. It uses a min-heap (the smallest element is at the root) to store elements, allowing quick retrieval of the highest-priority element (in a max-heap, you'd use a max-heap).

Key Components

- Heap Array: The underlying data structure – an array that represents the heap. (We can also use ArrayList)
- Size: The number of elements currently in the queue.
- Methods:
  - add(item): Inserts a new item into the heap.
  - poll(): Removes and returns the item with the highest priority (the root of the heap).
  - peek(): Returns the item with the highest priority without removing it.
  - isEmpty(): Checks if the heap is empty.

HeapQueue: Max Heap

1. Purpose of the Program

HeapQueue is a **Max Heap Priority Queue** implementation.

1. Stores integers in an array.
2. The largest value is always kept at the root.
3. Supports:
   - Add an element (`add()`)
   - Remove the highest-priority element (`poll()`)
   - View the highest-priority element (`peek()`)
   - Check if the heap is empty (`isEmpty()`)

HeapQueue: Max Heap

Method	Purpose
`HeapQueue()`	Create an empty heap
`add(item)`	Insert a new element
`heapifyUp()`	Move inserted element upward
`poll()`	Remove largest element
`heapifyDown()`	Move replacement element downward
`peek()`	View largest element
`isEmpty()`	Check if heap contains elements

HeapQueue

```
import java.util.Arrays;

public class HeapQueueExample {
    private int[] heap;
    private int size;

    public HeapQueueExample() {
        heap = new int[10]; // Initial capacity
        size = 0;
    }

    // Add an item to the heap
    public void add(int item) {
        if (size == heap.length) {
            UI.println("Heap is full");
            return;
        }
        heap[size] = item;
        size++;
        heapifyUp(size - 1); // Maintain heap property
    }
```

Methods to be defined:
1. HeapifyUp

HeapQueue

```
public class HeapQueueExample {
            …
            public int poll() {
    if (isEmpty()) {
       System.out.println("Heap is empty");
       return -1; // Or throw an exception
    }
    int max = heap[0]; // Root is the max
    heap[0] = heap[size - 1]; // Move last element to root
    size--;
    heapifyDown(0); // Maintain heap property
    return max;
  }
        // Peek - Return the highest priority element without removing it
        public int peek() {
                if (isEmpty()) {
                        System.out.println("Heap is empty");
                        return -1; // Or throw an exception
                }
        return heap[0]; // Root is the max
        }
}
```

Helper Methods:

1. HeapifyUp
2. HeapifyDown

HeapQueue

```
// Helper methods
   private void heapifyUp(int index) {
      while (index > 0) {
         int parentIndex = (index - 1) / 2;
         if (heap[index] > heap[parentIndex]) {
            // Swap
            int temp = heap[index];
            heap[index] = heap[parentIndex];
            heap[parentIndex] = temp;
            index = parentIndex;
         } else {
            break; // Heap property satisfied
         }
      }
   }
```

HeapQueue

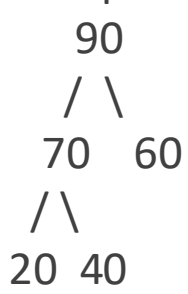
```
private void heapifyDown(int index) {
    while (true) {
        int largest = index;
        int leftChildIndex = 2 * index + 1;
        int rightChildIndex = 2 * index + 2;

        if (leftChildIndex < size && heap[leftChildIndex] > heap[largest]) {
            largest = leftChildIndex;
        }

        if (rightChildIndex < size && heap[rightChildIndex] > heap[largest]) {
            largest = rightChildIndex;
        }

        if (largest != index) {
            // Swap
            int temp = heap[index];
            heap[index] = heap[largest];
            heap[largest] = temp;
            index = largest;
        } else {
            break; // Heap property satisfied
        }
    }
}
```

HeapQueue

Heap Structure
Example heap:
[90,70,60,20,40]

Array index mapping:
Index: 0 1 2 3 4
Value: 90 70 60 20 40

Tree representation:



Relationships:
Parent = (i - 1) / 2

Left Child = 2i + 1

Right Child = 2i + 2

HeapQueue with ArrayList

HeapQueue

```
public class HeapQueue <E> extends AbstractQueue <E> {
    private List<E> data =  new ArrayList<E>();
    private Comparator<E> comp;
    public HeapQueue (Comparator <E> c) {
        comp = c;
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }
    public int size () {
        return data.size();
    }
    public E peek () {
        if  (isEmpty())
            return null;
        else
            return data.get(0);
    }
```

Use ArrayList, not array, so it handles resizing.

Comparator must be designed so that it compares the priority values (not the items)

HeapQueue

Method	Purpose	Return Value
HeapQueue(Comparator<E> c)	Creates a heap and stores comparator	Heap object
isEmpty()	Checks if heap has elements	true or false
size()	Counts elements in heap	Number of elements
peek()	Views highest-priority element without removing it	Root element or null

HeapQueue: AbstractQueue

What is AbstractQueue in Java?

`AbstractQueue` is an **abstract class** in the Java Collections Framework that provides a partial implementation of the `Queue` interface.

It is located in:

```
import java.util.AbstractQueue;
```

Why use AbstractQueue?

Without `AbstractQueue`, if you create your own queue class, you must implement **all methods** from the `Queue` interface.

`AbstractQueue` already provides implementations for some common queue operations, so you only need to implement the essential ones.

HeapQueue: AbstractQueue

```
public class HeapQueue<E> extends AbstractQueue<E>
```

This means:

- HeapQueue inherits functionality from AbstractQueue.
- HeapQueue behaves like a Java Queue.
- HeapQueue only needs to implement a few required methods.

HeapQueue: AbstractQueue

Methods Typically Required

When extending `AbstractQueue`, you usually implement:

```
public boolean offer(E e)
```

Adds an element.

```
public E poll()
```

Removes and returns the head element.

```
public E peek()
```

Returns the head element without removing it.

```
public Iterator<E> iterator()
```

Allows traversal through the queue.

```
public int size()
```

Returns the number of elements.

HeapQueue: offer and poll

```
public boolean offer(E value) {
    if  (value == null) return false;
    else  {
        data.add(value);
        pushup(data.size()-1);
        return true;
    }
}
```

add at the end of the array

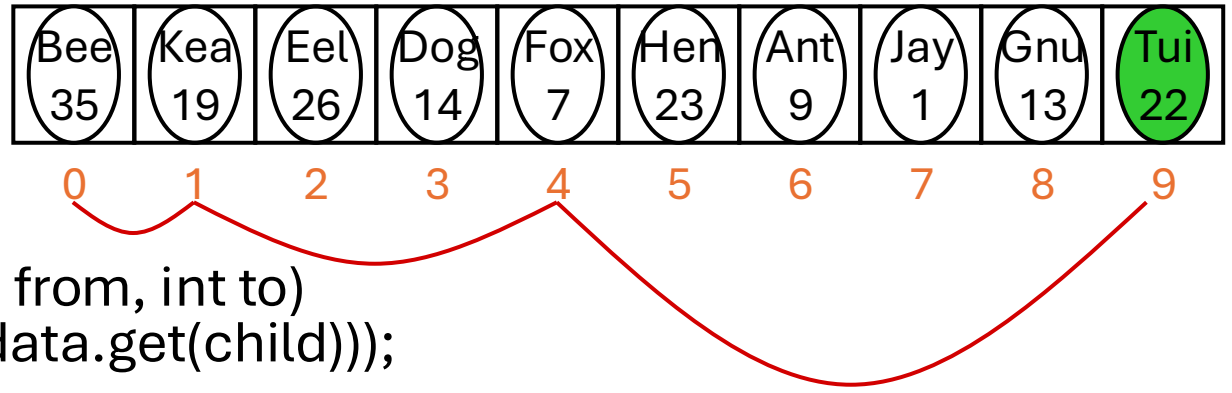
```
public E poll() {
    if  (isEmpty())              return null;
    if  (data.size() == 1)       return data.remove(0);
    else  {
        E ans = data.get(0);
        data.set(0, data.remove(data.size()-1));
        pushdown(0);
        return ans;
    }
}
```

move the last element into the root

HeapQueue: pushup

```
private void pushup(int child) {
    if  (child == 0) return;
    int parent = (child-1)/2;
    // compare with value at parent and swap if parent smaller
    if  (comp.compare(data.get(parent), data.get(child)) < 0) {
        swap(data, child, parent);
        pushup(parent);
    }
}
```

recurse up the tree...

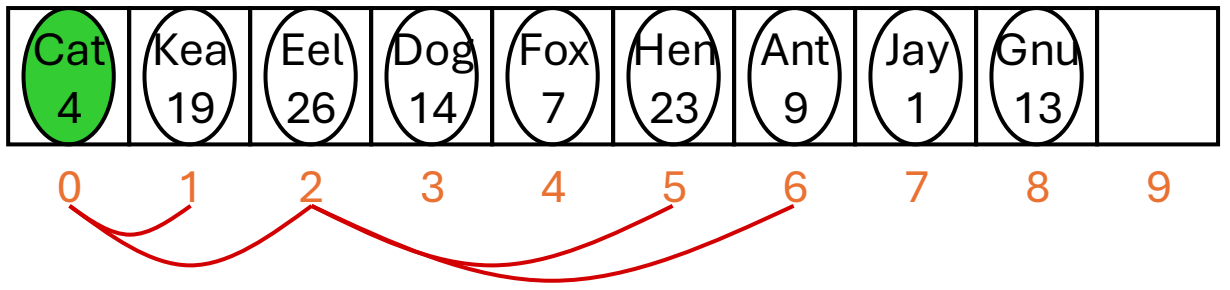



```
private void swap(List<E> data, int from, int to)
   data.set(child, data.set(parent, data.get(child)));
```

HeapQueue: pushdown

```
private void pushdown(int parent) {
    int largeCh = 2*parent+1;
    int otherCh = largeCh+1;
   // check if any children
    if  (largeCh >= data.size()) return;
   // find largest child
    if  (otherCh < data.size() &&
         comp.compare(data.get(largeCh), data.get(otherCh)) < 0 )
          largeCh = otherCh;
   // compare with largest child, and swap if smaller
    if  (comp.compare(data.get(parent), data.get(largeCh)) < 0) {
          swap(data, largeCh, parent);
          pushdown(largeCh);
    }
}
```

recurse down the tree...



Review

Question 1:
What data structure is implemented by the HeapQueue class?

Question 2: What is the purpose of the constructor below?

```
public HeapQueue() {
    heap = new int[10];
    size = 0;
}
```

Question 3: What is the purpose of the variable size?

Question 4: What happens when the following statement is executed?

```
heap[size] = item;
```

Question 5: What is the purpose of the peek() method?

Question 6: What is returned by the following code?

```
heap = [90, 70, 60, 20, 40]
size = 5
poll();
```