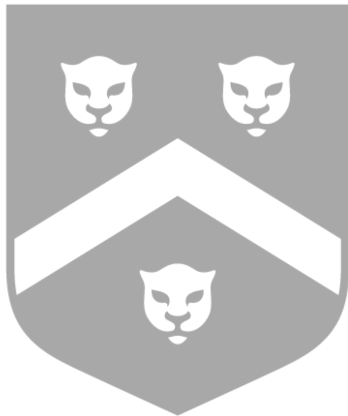
Binary Search Trees and Priority Queues



Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

October 26, 2022

Binary Search Trees

Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()

Priority Queues

Binary Search Trees

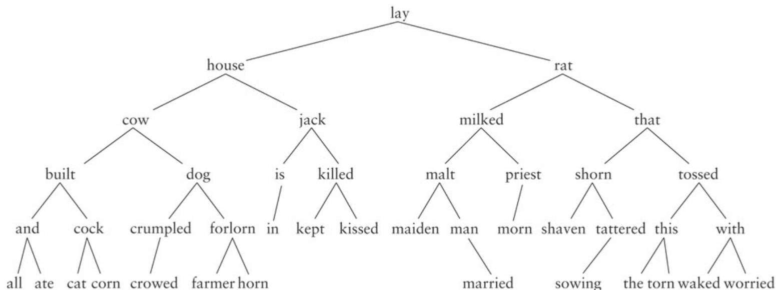
Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

Binary Search Trees

- BSTs follow the binary tree definition, with an additional property:
  - For every node, all values in the left subtree are less than that node
  - For every node, all values in the right subtree are greater than that node
- Visually, values are increasing moving from left to right – there aren't direct left-to-right connections to move in the actual structure
- Values are unique in our version (though that is not part of the definition)

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

Example BST



Binary search tree with Strings – order is decided alphabetically

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

Binary Search Algorithm

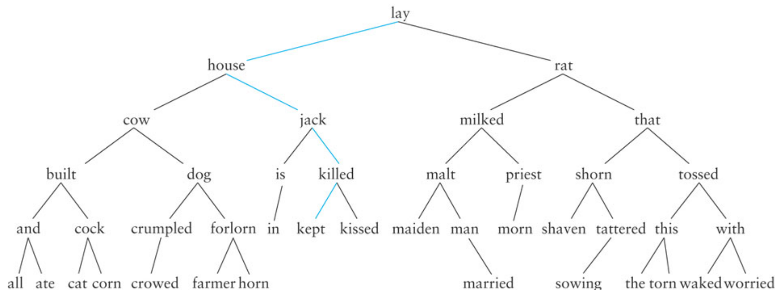
Starting at a tree's root, this algorithm recursively searches in a tree for a target:

SEARCH(tree, target)

1: **if** tree is empty **then**
2:     **return** null // target not found
3: **else if** target matches root of tree **then**
4:     **return** root node
5: **else if** target $<$ root node **then**
6:     **return** SEARCH(root's left child, target)
7: **else**
8:     **return** SEARCH(root's right child, target)

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

Example Search Path



Path to find “kept” – we only need to follow the highlighted connections

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

Interface

Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if item is inserted; **`false`** if it isn't (already in tree).
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree.
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**.
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**.
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**.

BST interface – simple and efficient operations

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

UML Diagram

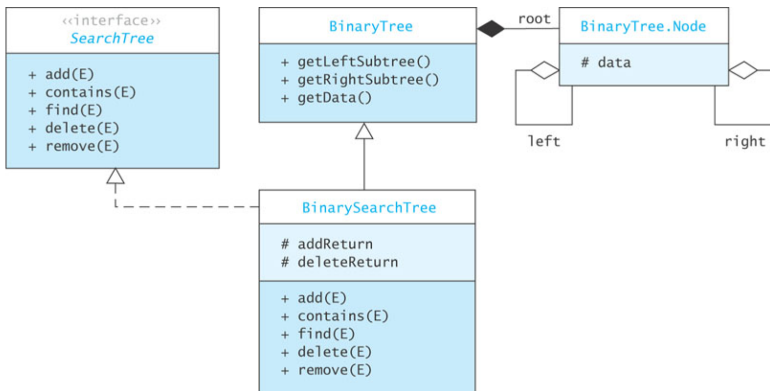


Diagram showing class models and relationships

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

find() implementation

```
public E find(E target) {
    return find(root, target);
}

private E find(Node<E> localRoot, E target) {
  if (localRoot == null)
    return null;

  // Compare the target with the data at the root.
  int compResult = target.compareTo(localRoot.data);
  if (compResult == 0)
    return localRoot.data;
  else if (compResult < 0) {
    return find(localRoot.left, target);
  else
    return find(localRoot.right, target);
}
```

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

add Algorithm

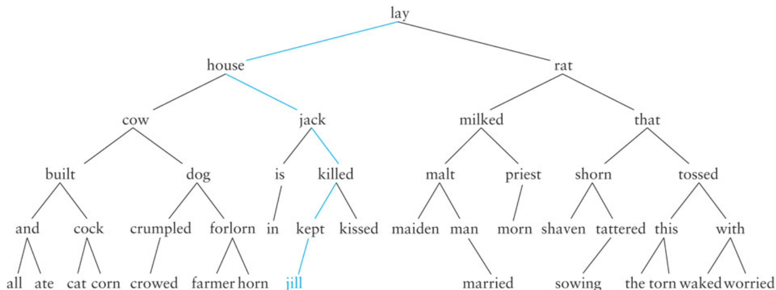
ADD(tree, item)

1: **if** tree is empty **then**
2: replace null with item node
3: **return** true
4: **else if** item matches root of tree **then**
5: **return** false
6: **else if** item $<$ root node **then**
7: **return** ADD(root's left child, item)
8: **else**
9: **return** ADD(root's right child, item)

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

add path

Deciding where to add a new value follows a similar path to finding an existing value



adding “jill” in the appropriate place

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

add() implementation

```
/** Starter method add.
    pre: The object to insert must implement the
         Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
      if the object already exists in the tree
*/
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}
```

Binary Search Trees
Binary Search Trees
Example
Searching
Search Example
Interface
UML Diagram
find()
add()
add()
Priority Queues

`add()` implementation

```
private Node<E> add(Node<E> localRoot, E item) {
  if (localRoot == null) {
    // item is not in the tree - insert it.
    addReturn = true;
    return new Node<E>(item);
  } else if (item.compareTo(localRoot.data) == 0) {
    addReturn = false;
    return localRoot;
  } else if (item.compareTo(localRoot.data) < 0) {
    localRoot.left = add(localRoot.left, item);
    return localRoot;
  } else {
    localRoot.right = add(localRoot.right, item);
    return localRoot;
  }
}
```

Binary Search Trees

Priority Queues

Definition

Heap

offer()

poll()

Java Class

Priority Queues

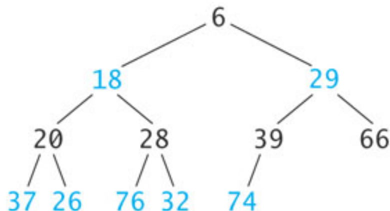
Binary Search Trees
Priority Queues
Definition
Heap
offer()
poll()
Java Class

Definition

- A priority queue works like a queue with an extra feature:
  - Each item in the queue can be marked with a priority
  - Smallest priority item is always at the front
- Works like a queue when all items have the same priority
- Otherwise, items in the queue might need to be rearranged when a new item is added
- Same operations as queues: offer/poll/peek

Heap

- A **heap** is a complete binary tree with an additional property:
  - Each node in the heap is smaller than all of its descendants
- We can use a heap to efficiently implement a priority queue
- For each priority queue method that modifies the queue, we need to make sure that our heap stays consistent



offer Algorithm

OFFER(heap, item)

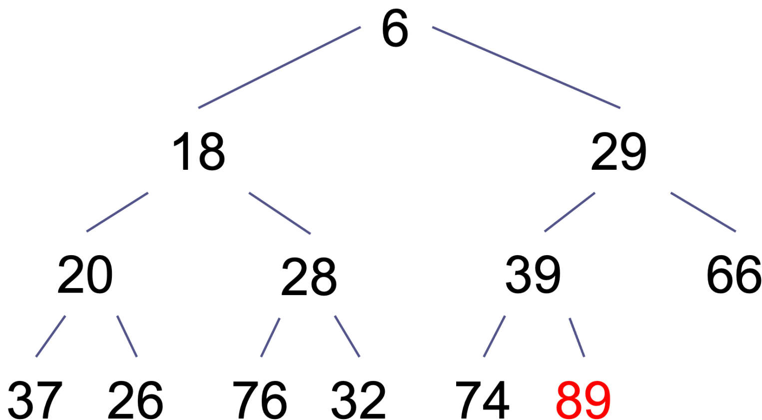
1: Insert new item at next position in bottom of heap
2: **while** new item < parent **do**
3: Swap new item and parent

Binary Search Trees

Priority Queues

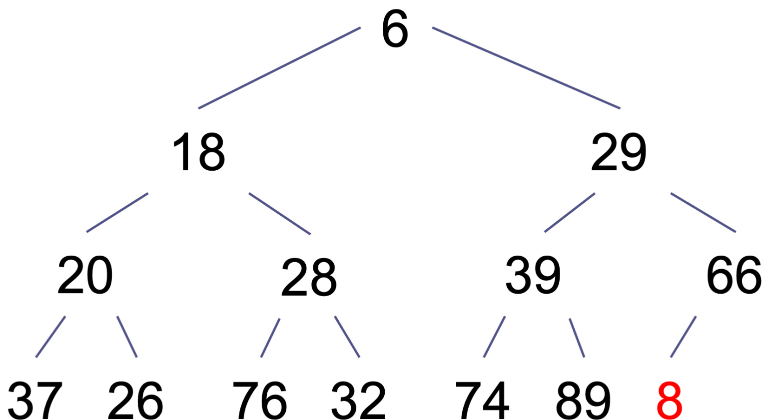
- Definition
- Heap
- offer()
- poll()
- Java Class

offer example



offer(89): insert at bottom, compare with 39 as parent

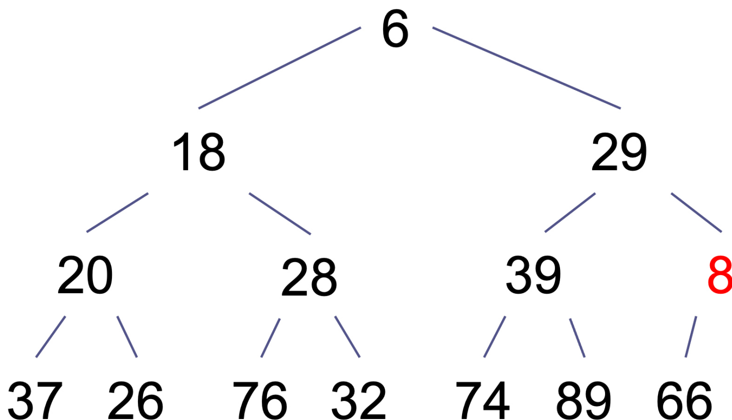
offer example



offer(8): insert at bottom, compare with 66 as parent

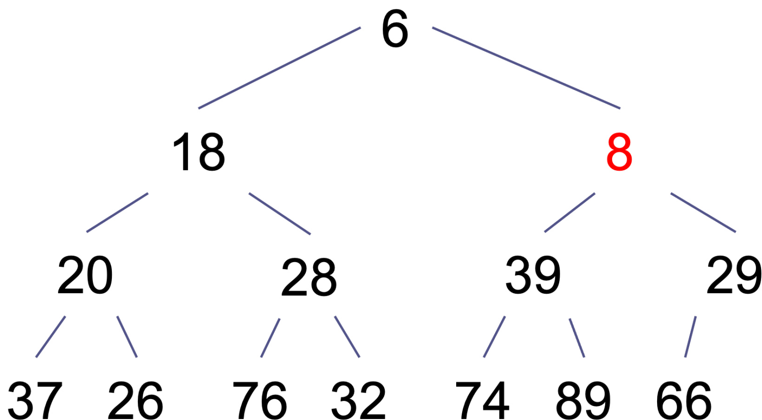
Binary Search Trees
Priority Queues
Definition
Heap
offer()
poll()
Java Class

offer example



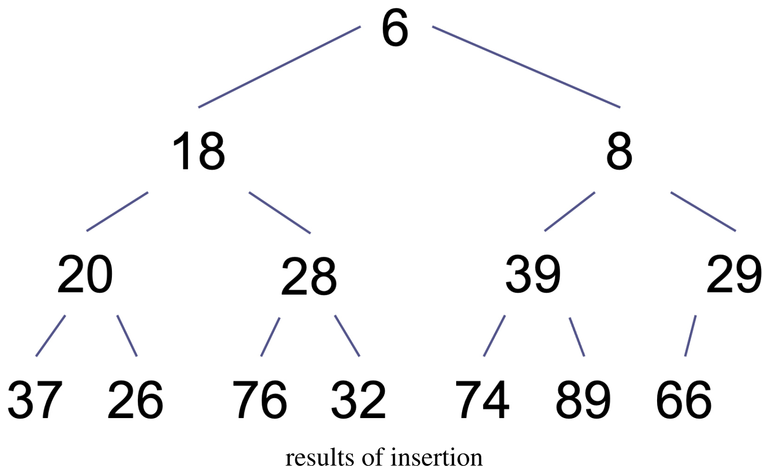
swap with parent, compare with 29 as new parent

offer example



swap with parent, compare with 6 as new parent

offer example



results of insertion

poll Algorithm

POLL(heap)

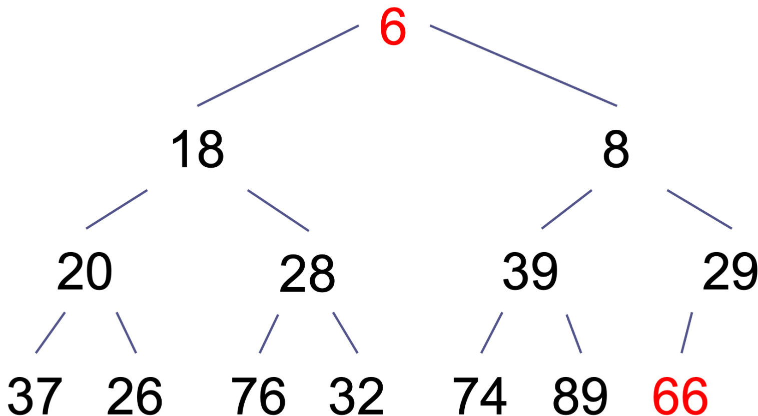
1: Remove root and replace it with last item in heap
2: **while** moved item > valid children **do**
3: Swap moved item and smallest child
4: Return old root

Binary Search Trees

Priority Queues

- Definition
- Heap
- offer()
- poll()
- Java Class

poll example



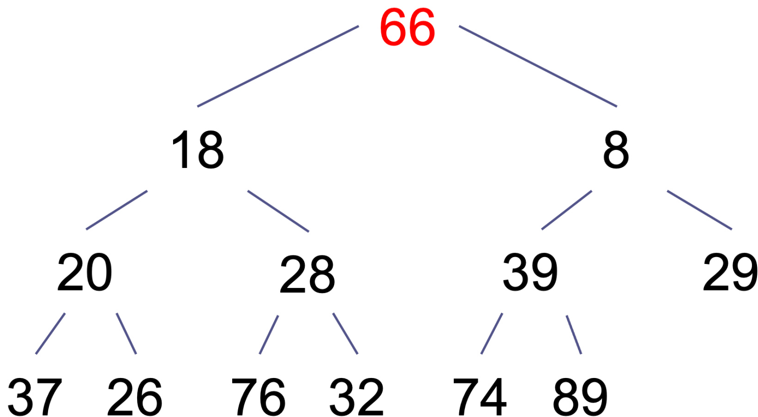
poll(): remove 6, move 66 to root location

Binary Search Trees

Priority Queues

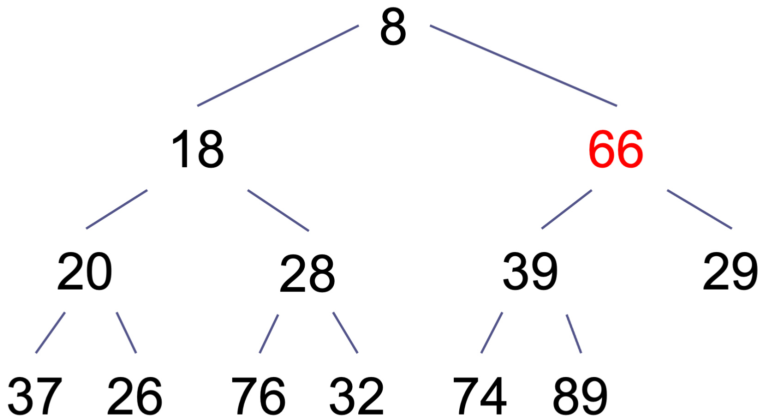
- Definition
- Heap
- offer()
- poll()
- Java Class

poll example



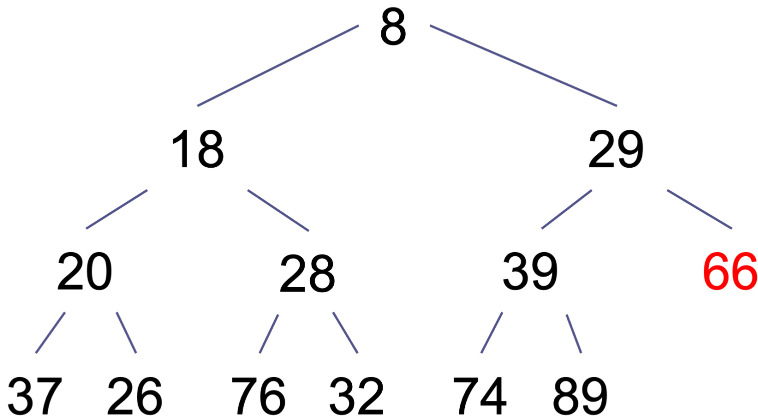
compare with children, swap with smaller child (right child)

poll example



compare with children, swap with smaller child (right child)

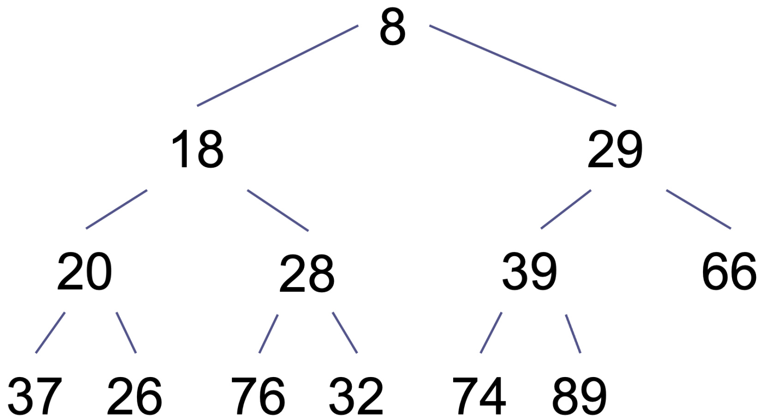
poll example



no children to compare to, return 6 as polled value

Binary Search Trees
Priority Queues
Definition
Heap
offer()
poll()
Java Class

poll example



no children to compare to, return 6 as polled value

Binary Search Trees

Priority Queues

- Definition
- Heap
- offer()
- poll()
- Java Class

PriorityQueue Java Class

Data Field	Attribute
`ArrayList<E> theData`	An `ArrayList` to hold the data.
`Comparator<E> comparator`	An optional object that implements the `Comparator<E>` interface by providing a `compare` method.
Method	**Behavior**
`KWPriorityQueue()`	Constructs a heap-based priority queue that uses the elements' natural ordering.
`KWPriorityQueue (Comparator<E> comp)`	Constructs a heap-based priority queue that uses the `compare` method of `Comparator comp` to determine the ordering of the elements.
`private int compare(E left, E right)`	Compares two objects and returns a negative number if object `left` is less than object `right`, zero if they are equal, and a positive number if object `left` is greater than object `right`.
`private void swap(int i, int j)`	Exchanges the object references in `theData` at indexes `i` and `j`.