Priority Queues

- The PQueue ADT
- A linked implementation
- Heaps
- A heap implementation of a priority queue
- Heapsort

Reading: Weiss Section 6.9 and Chapter 21

Priority Queues

A priority queue is an extension of the queue ADT. In a priority queue however, when an item is added to the queue, it is assigned a *priority* (an integer) to indicate the relative importance of the item.

Instead of storing items in chronological order, a priority queue archives items with the highest priority before all others.

Items are no longer removed on a first-in-first-out basis — items are removed depending on their priority, with items of equal priority processed in chronological order.

Example uses include:

- scheduling services — eg distributing CPU time among several threads
- optimization algorithms — such as Prim's algorithms, Dijkstra's Algorithm, A*
- sorting
- backbone routers of internet

PQueue Specification

☐ Constructors

1. *PQueue()*: initialises an empty priority queue.

☐ Checkers

2. *isEmpty()*: returns *true* if the priority queue is empty.

☐ Manipulators

3. *enqueue(e, k)*: places e in the priority queue with key (or priority) k, or throws an exception if k is negative. The item is placed in front of all elements with lesser priority, but behind all others.

4. *examine()*: returns the element at the front of the queue, or throws an exception if the queue is empty.

5. *dequeue()*: removes the element at the front of the queue and returns it, or throws an exception if the queue is empty.

Example

```
PQueue p = new PQueue();    []

p.enqueue('a', 1);          [<a,1>]

p.enqueue('b', 3);          [<b,3>, <a,1>]

p.enqueue('c', 3);          [<b,3>, <c,3>, <a,1>]

p.enqueue('d', 2);          [<b,3>, <c,3>, <d,2>, <a,1>]

p.dequeue();                [<c,3>, <d,2>, <a,1>]
                            // b is returned.
```

Linked Implementation

```
/**
 * A Linked Priority Queue for the generic type E
 **/
public class PQueueLinked<E> {

  // Only one link is required!
  private Link<E> front;

  // Constructor
  public PQueueLinked() {
    front = null;
  }
```

A Link Inner Class

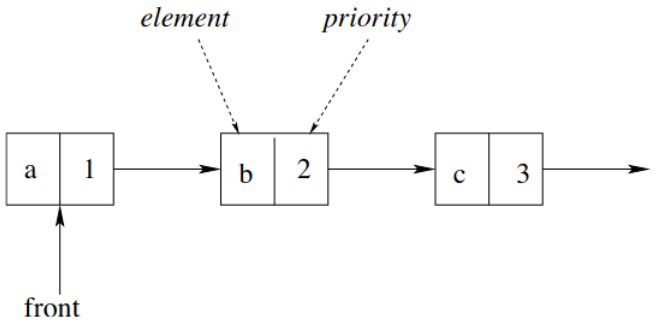
```
/**
 * An inner class to hold the element, the successor,
 * and the priority
 **/
private class Link<E> {
  E element;
  int priority;
  Link<E> next;

  public Link(E e, int p, Link<E> n) {
    this.element = e;
    this.priority = p;
    this.next = n;
  }

}
```

isEmpty, examine, and dequeue

The process of checking to see if the priority queue is empty, examining the front element, or dequeuing the front element can all be done in the same way as for the queue ADT.



```
public boolean isEmpty() {return front == null;}

public E examine() throws Exception {
  if (!isEmpty()) {
    return (E) front.element;
  } else throw new Exception("Empty Queue");
}

public E dequeue() throws Exception {
  if (!isEmpty()) {
    E temp = (E) front.element;
    front = front.next;
    return temp;
  } else throw new Exception("Empty Queue");
}
```

Enqueuing

To enqueue, we start at the front of the queue and keep moving back until we find some element of lesser priority, or reach the end of the queue.

We then insert the new element in front of the lesser element.

```
public void enqueue(E e, int p) {
  if (isEmpty() || p > front.priority) {
    front = new Link<E>(e, p, front);
  } else {
    Link<E> l = front;
    while (l.next != null && l.next.priority >= p) {
      l = l.next;
    }
    l.next = new Link<E>(e, p, l.next);
  }
}
```

Performance

- *enqueue*: This operation is performed by iterating through the queue from the front to the back until the correct location to insert the new element is found. In the worst case, the entire queue will be examined $\rightarrow O(n)$ where n is the size of the queue.
- *examine*: This operation simply returns the element at the front of the queue $\rightarrow O(1)$.
- *dequeue*: This operation returns the element at the front of the queue and updates the value of front $\rightarrow O(1)$.

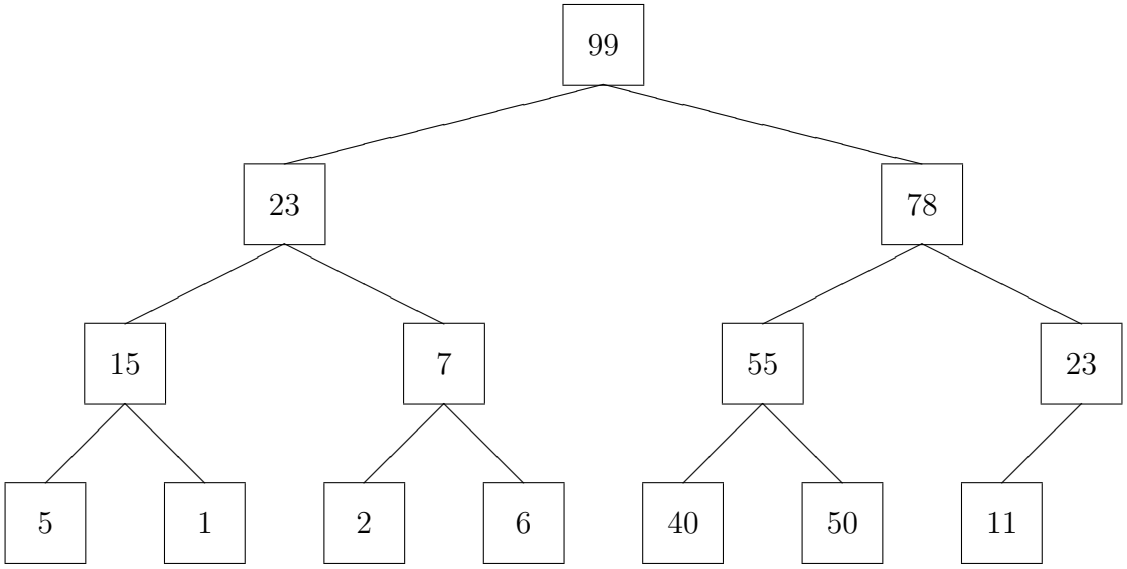
Heap Implementation

The **Heap** data structure is based on a binary tree, where each node of the tree contains an *element* and a *key* (an integer) — effectively the priority of the element.

A heap has the added property that the key associated with any node is greater than or equal to the key associated with either of its children.

$\rightarrow$ the root of the binary tree has the largest key.

Note also that there is no requirement to order the left and right children of a node.



Just like the infinite binary tree, we *store* the heap as a linear array:

99	23	78	15	7	55	23	5	1	2	6	40	50	11

The bottom level of the binary tree, if not complete, is filled from the left.

Parents and Children

Suppose the array A is indexed from 1.

Then $A[1]$ holds the root of the binary tree, which by the heap property is, the element with the largest key value.

The two children of the root are stored in $A[2]$ and $A[3]$.

In general, the left child of node i is $2i$ and the right child is $2i + 1$.

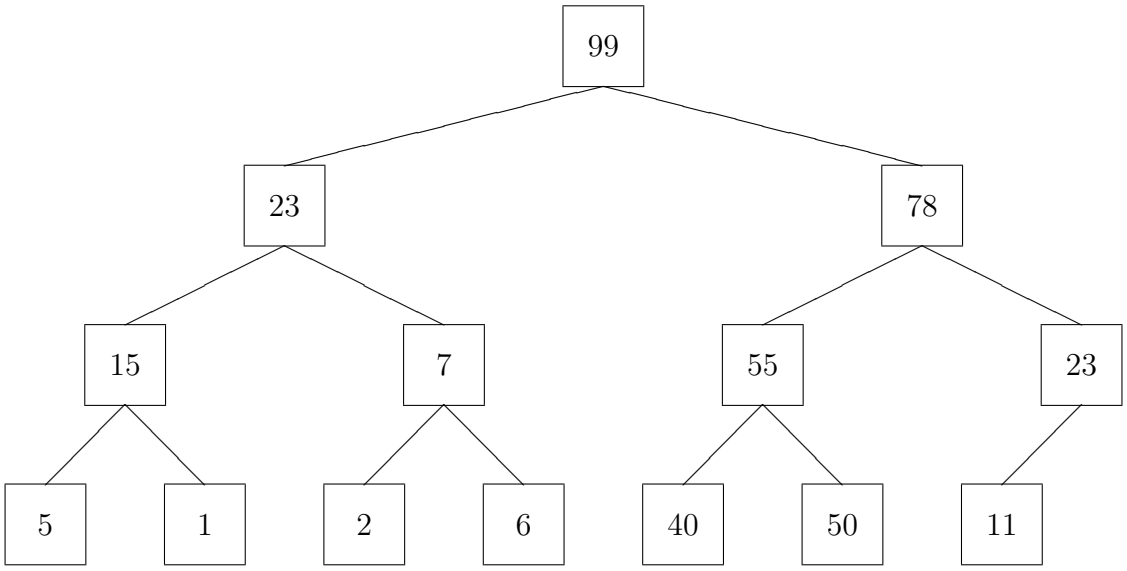
Conversely, the parent of node i is $\lfloor i/2 \rfloor$.

$\rightarrow$ operations to determine both the parent and children of a node are $O(1)$.

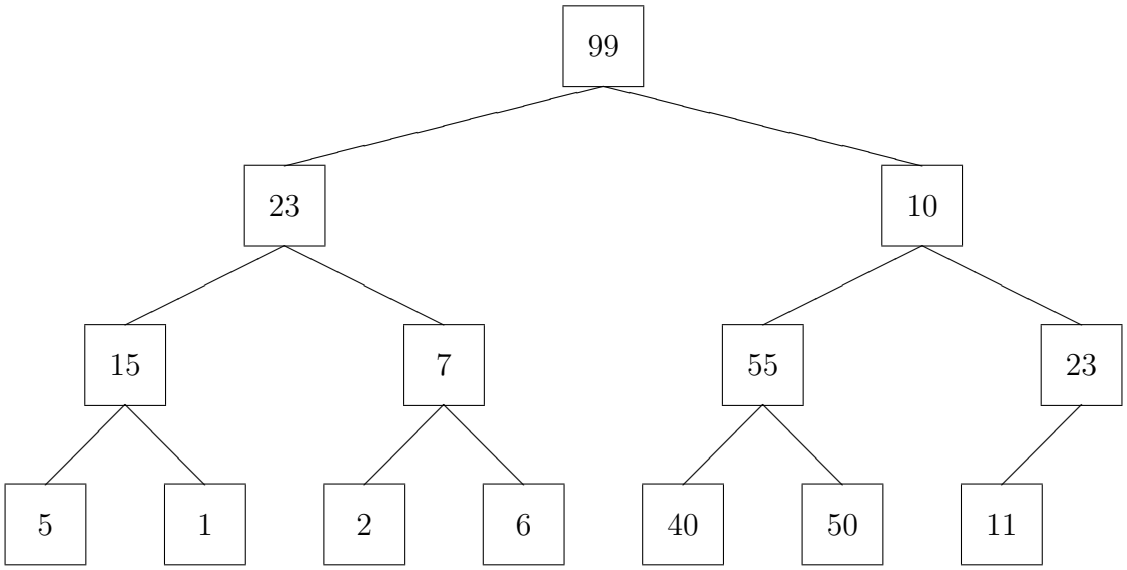
Heapify

Most of the operations we wish to perform on the heap will alter the data structure. Often these operations will destroy the heap property and it will have to be restored.

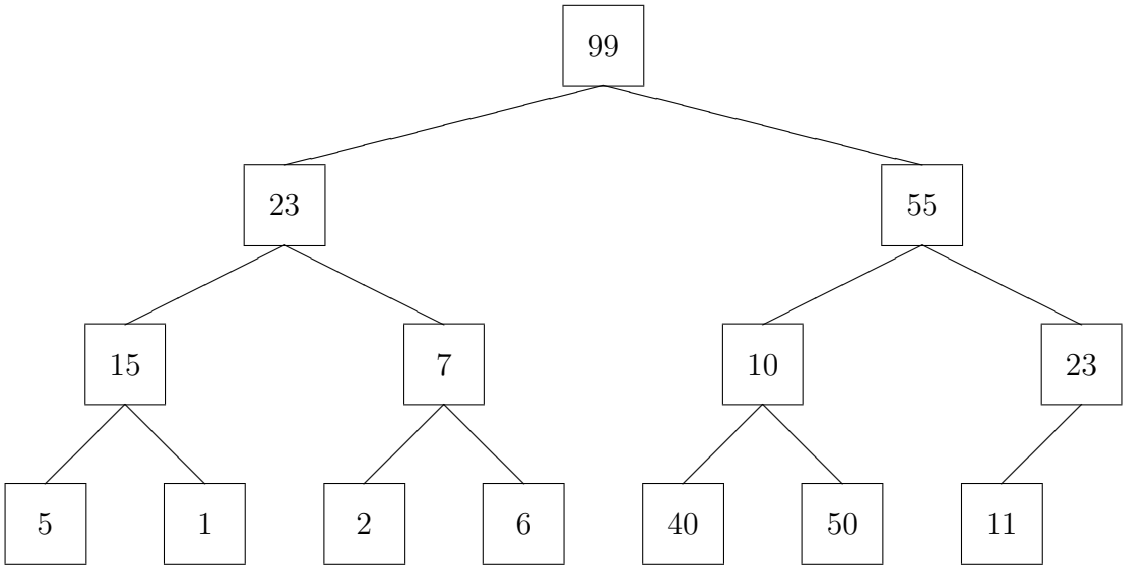
For example, suppose we decide to alter the value associated with one of the nodes — eg we wish to set $A[3] = 10$.



Making the change is trivial, but the resulting structure is no longer a heap — the entry $A[3]$ is no longer larger than both of its children.



We can rectify this by swapping $A[3]$ with the *larger* of its two children.



This means that the problem at $A[3]$ has been fixed — however, we *may* have introduced a problem at $A[6]$.

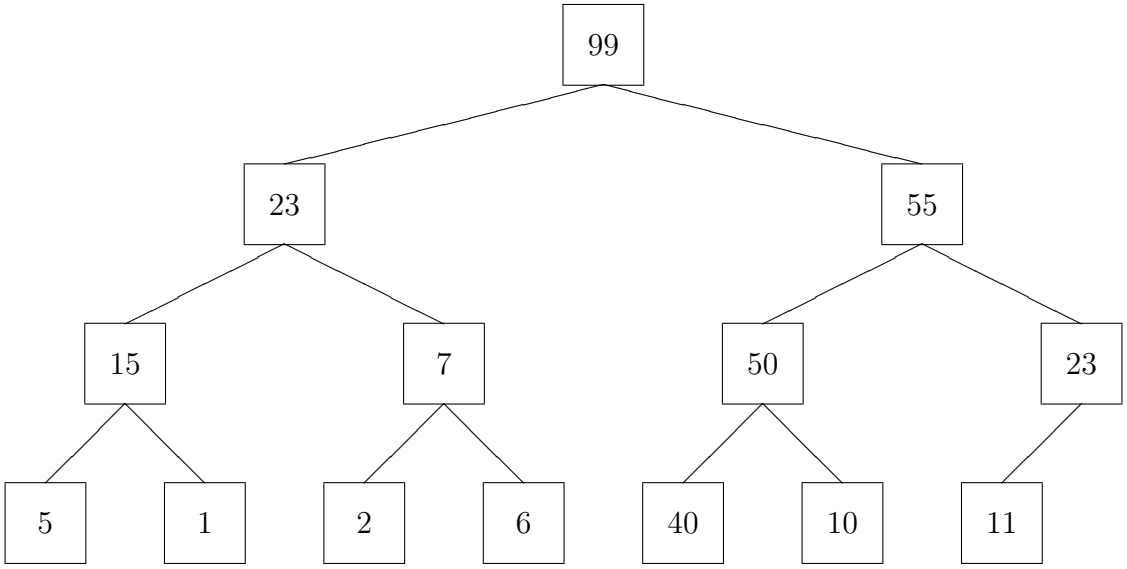
So we now examine $A[6]$ and if it is smaller than its children we perform another exchange.

We continue recursively checking the swapped child until we restore the heap property or reach the end of the tree.

The procedure outlined above, whereby a small element "percolates" down the tree is called *heapify*.

Heapify takes a position i in the tree as an argument and iterates down the tree from i, swapping entries if necessary until the heap property is restored.

Heapify assumes that the two children of i are proper heaps, but that the key value $A[i]$ may not be larger than the key values of its children.


99
23
55
15
7
50
23
5
1
2
6
40
10
11

Complexity of heapify

A balanced binary tree with n elements in it has a height of $\log n$ and hence *heapify* performs at most $\log n$ exchanges.

$\rightarrow$ *heapify* is $O(\log n)$.

Consider now how all the operations necessary for a priority queue can be accomplished by using a heap together with *heapify*...

Performance

- *enqueue*: The key is entered at the end of the array (ie, in the last position in the tree). The resulting structure may not be a heap, because the value of the new key may be greater than its parent. If this is the case, exchange the two keys and proceed to examine the parent. In the worst case, $\log n$ exchanges will have to be done $\rightarrow O(\log n)$.

- *examine*: The root of the binary tree is the entry with the largest key value. Hence, merely return the root of the tree $\rightarrow O(1)$.

- *dequeue*: In this case, we must also delete the root node from the tree and then restore the heap property. This can be achieved by moving the final entry in the tree to the newly-vacated root position and then calling `heapify(1)` to restore the heap property. This involves a few constant time operations, together with one call to *heapify* $\rightarrow O(\log n)$.

Heapsort

The heap data structure allows us to implement a relatively efficient sorting procedure. Suppose we are given an unordered array containing a number of integers (or objects that can be completely ordered) that we wish to arrange from highest to lowest. Suppose there are n elements in the array.

Imagine enqueuing each of the elements into a priority queue with a priority equal to their value. This involves n operations at $O(\log n)$ each $\rightarrow$ $O(n \log n)$.

We can then dequeue each element into an array; the elements being dequeued in sorted order. This involves n operations at $O(\log n)$ each $\rightarrow$ $O(n \log n)$.

$\rightarrow$ overall complexity is $O(n \log n)$, which is optimal for sorting using comparisons.

Pseudo-code for Heapsort

```
int[] heapSort(int[] arr) {
  // Create a priority queue
  PQueue p = new PQueue();
  // Add in the elements with themselves as the key
  for (int i : arr) {
    p.enqueue(i, i);
  }

  // Create a new array to store the result
  int[] ans = new int[arr.length];
  // Dequeue the elements from the priority queue into ans
  for (int i = 0; i < ans.length; i++) {
    ans[i] = p.dequeue();
  }
  return ans;
}
```

Performance Comparison

Operation	Linked	Heap
enqueue	n	$\log n$
examine	1	1
dequeue	1	$\log n$

Summary

- Priority queues behave like queues except elements are enqueued with a priority and are returned in order of their priority
- Linked representation
  - based on maintaining the elements in sorted priority order
  - constant time examination and dequeuing
  - linear time enqueuing
- Heap representation
  - based on maintaining elements in a heap: elements are stored in a binary tree with the added property that all nodes have a greater than or equal value to both of its children.
  - constant time examination
  - logarithmic time enqueuing and dequeuing