Cosc 241
Programming and Problem Solving
Lecture 20 (9/5/2019)
Heaps

Lech Szymanski
lech.szymanski@cs.otago.ac.nz

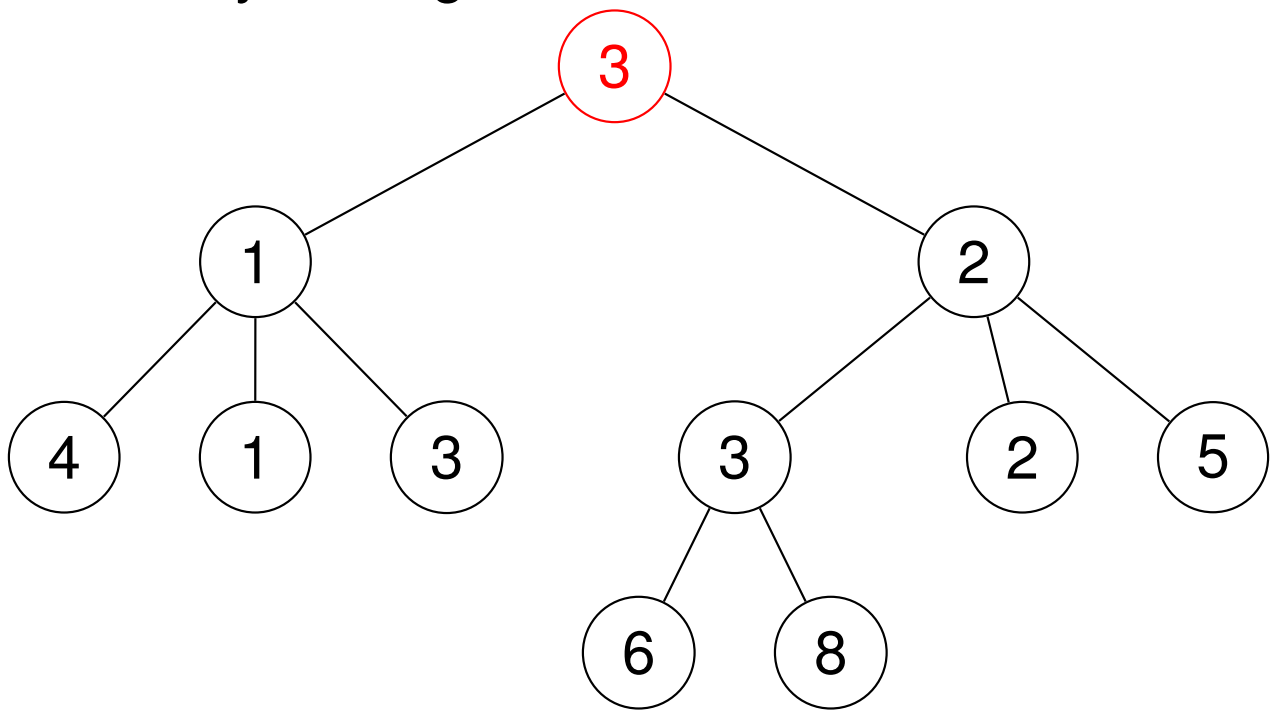
Keywords: heap

Prelude: trees

- A tree is a certain type of linked data structure.
- A tree consists of nodes and each node may contain links to a set of children.
- There is exactly one node that is not the child of any other node and it is called the root.
- There are no cycles (that is, a node cannot be the child of more than one parent).
- Generally each node stores data of some kind.

1K words

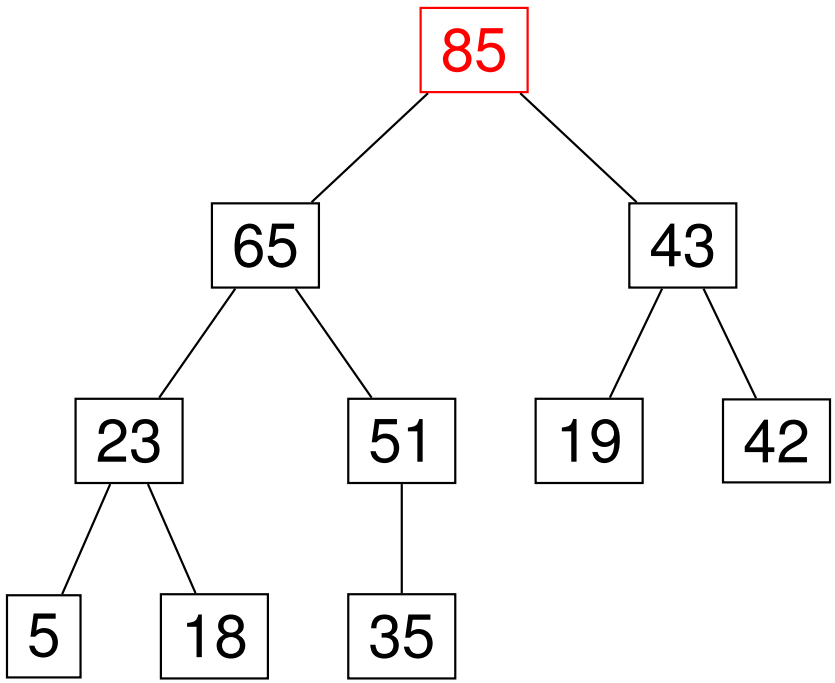
The red node is the root, values stored at the nodes are shown, and a node's children are the ones immediately below it connected to it by an edge.



Heaps

- A heap is a tree where the values stored are comparable which satisfies certain additional restrictions.
- The value stored at a node is always greater than or equal to the value stored at any of its children.
- Each node has at most two children.
- In fact, in a given level either:
  - Every node has two children, or
  - From left to right we see a sequence of nodes with two children, then possibly one node with one child, then nodes with no children, and no nodes in the next level have children.

Another 1K words



Heap operations

The fundamental operations that a heap should support are:

- Adding an item,
- Returning the maximum value in the heap,
- Removing the maximum value from the heap.

These correspond to

- (push, peek, pop) for stacks, and
- (add, peek, remove) for queues.

Usefulness

There are two main uses for heaps:

- In *priority queues* a data structure in which the item with highest priority is always the next one to be processed.
- As part of the *heap sort* algorithm, which sorts data by adding it item by item to a heap and then simply removing from the heap until nothing is left.

Heap algorithms

Adding an item:

- Place the item to be added in the first vacant leaf position.
- Let it float up the branch towards the root so long as it is larger than its parent.

Returning/Removing the maximum

- The maximum is always the root of the tree, so finding and returning it is not an issue.
- The issue is, if we want to remove it, how to reconstruct the tree, *maintaining the heap property* after the root element is removed.
- The key idea is in some sense the reverse of addition.
- Replace the root value with the value of the last leaf (removing the last leaf).
- Then, to re-establish the heap property exchange the root and the larger of its children (and do this recursively downwards).

Heap ADT

```
public interface Heap<T extends Comparable<T>> {

    public void add(T element);
    public T get();
    public T remove();

}
```

Heap implementations

What data structure shall we use to represent a heap?

- A *linked binary tree* is certainly possible.
  - To facilitate both upwards and downwards navigation, we would probably want to maintain links to parent nodes as well as to children.
  - To determine where the next item is to be added and where the last item is (for removal) we probably want to have a data field for the last item.
- How about an array?
  - We can store the root at position 0 and its (potential) children at indices 1 and 2.
  - The children of node 1 go in indices 3 and 4, and those of node 2 in indices 5 and 6.
  - In general, the children of node i go at indices $2i + 1$ and $2i + 2$.
  - If we keep track of the current size, then the position of the last item (and the next insertion point) is also known.
  - The only drawback is the need to resize if the heap gets too large.

Arrays it is

What extra internals will we need? (But we should really discover these as we go)

- A method to expand the capacity if/when needed.
- A method to swap the values at two positions.
- A method to find the index of the larger child (or tell us there isn't one).

Fields, constructors

```
public class ArrayHeap<T extends Comparable<T>>
    implements Heap<T>{

private static final int DEFAULT_CAPACITY = 31;
private static final int NO_LARGER_CHILD = -1;

private T[] heap;
private int size = 0;

public ArrayHeap() {
    this(DEFAULT_CAPACITY);
}

public ArrayHeap(int capacity){
    heap = (T[]) new Comparable[capacity];
}

  ...
```

Get, expand, swap

```
public T get() {
    return heap[0];
}

private void expandCapacity() {
    heap = Arrays.copyOf(heap, 2*heap.length+1);
}

private void swap(int i, int j) {
    T temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}
```

Add

```
public void add(T element) {
    if (size == heap.length) expandCapacity();
    heap[size] = element;
    size++;
    int childIndex = size-1;
    int parentIndex = (childIndex-1)/2;
    while (parentIndex >= 0 &&
            heap[parentIndex].compareTo(heap[childIndex]) < 0) {
        swap(childIndex, parentIndex);
        childIndex = parentIndex;
        parentIndex = (childIndex-1)/2;
    }
}
```

Larger child

```
private int getLargerChildIndex(int i) {
    int l = 2*i+1;
    int r = 2*i+2;
    if (r >= size || heap[r].compareTo(heap[l]) < 0) {
        if (l < size && heap[i].compareTo(heap[l]) < 0) {
            return l;
        }
    } else { // Right child exists and is larger than left
        if (heap[i].compareTo(heap[r]) < 0) {
            return r;
        }
    }
    return NO_LARGER_CHILD;
}
```

Remove

```
public T remove() {

    T result = heap[0];
    heap[0] = heap[size-1];
    size--;
    int parentIndex = 0;
    do{
        int largerChildIndex = getLargerChildIndex(parentIndex);
        if (largerChildIndex == NO_LARGER_CHILD) break;
        swap(parentIndex, largerChildIndex);
        parentIndex = largerChildIndex;
    } while (true);
    return result;
}
```