

# CSC 427: Data Structures and Algorithm Analysis

Fall 2006

## Applications:

- scheduling applications
  - priority queue, `java.util.PriorityQueue`
  - min-heaps & max-heaps, heap sort
- data compression
  - image/sound/video formats
  - text compression, Huffman codes
- graph algorithms

1

## Scheduling applications

many real-world applications involve optimal scheduling

- choosing the next in line at the deli
- prioritizing a list of chores
- balancing transmission of multiple signals over limited bandwidth
- selecting a job from a printer queue
- selecting the next disk sector to access from among a backlog
- multiprogramming/multitasking

what all of these applications have in common is:

- a collections of actions/options, each with a priority
- must be able to:
  - ✓ add a new action/option with a given priority to the collection
  - ✓ at a given time, find the highest priority option
  - ✓ remove that highest priority option from the collection

2

## Priority Queue

*priority queue* is the ADT that encapsulates these 3 operations:

- ✓ *add item (with a given priority)*
- ✓ *find highest priority item*
- ✓ *remove highest priority item*

e.g., assume printer jobs are given a priority 1-5, with 1 being the most urgent

a priority queue can be implemented in a variety of ways

- unsorted list

job1	job 2	job 3	job 4	job 5
3	4	1	4	2

efficiency of add? efficiency of find? efficiency of remove?

- sorted list (sorted by priority)

job4	job 2	job 1	job 5	job 3
4	4	3	2	1

efficiency of add? efficiency of find? efficiency of remove?

- others?

3

## java.util.PriorityQueue

Java provides a `PriorityQueue` class

```
public class PriorityQueue<E extends Comparable<? super E>> {
    /** Constructs an empty priority queue
     */
    public PriorityQueue<E>() { ... }

    /** Adds an item to the priority queue (ordered based on compareTo)
     * @param newItem the item to be added
     * @return true if the items was added successfully
     */
    public boolean add(E newItem) { ... }

    /** Accesses the smallest item from the priority queue (based on compareTo)
     * @return the smallest item
     */
    public E peek() { ... }

    /** Accesses and removes the smallest item (based on compareTo)
     * @return the smallest item
     */
    public E remove() { ... }

    public int size() { ... }
    public void clear() { ... }
    ...
}
```

the underlying data structure is  
a special kind of binary tree  
called a *heap*

4

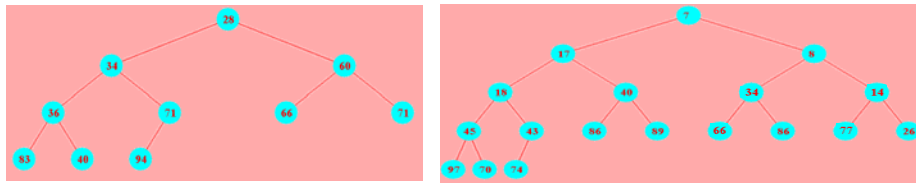
## Heaps

a *complete tree* is a tree in which

- all leaves are on the same level or else on 2 adjacent levels
- all leaves at the lowest level are as far left as possible

a *heap* is complete binary tree in which

- for every node, the value stored is  $\leq$  the values stored in both subtrees  
(technically, this is a *min-heap* -- can also define a *max-heap* where the value is  $\geq$ )



since complete, a heap has minimal height =  $\lfloor \log_2 N \rfloor + 1$

- can insert in  $O(\text{height}) = O(\log N)$ , but searching is  $O(N)$
- not good for general storage, but perfect for implementing priority queues  
can access min value in  $O(1)$ , remove min value in  $O(\text{height}) = O(\log N)$

5

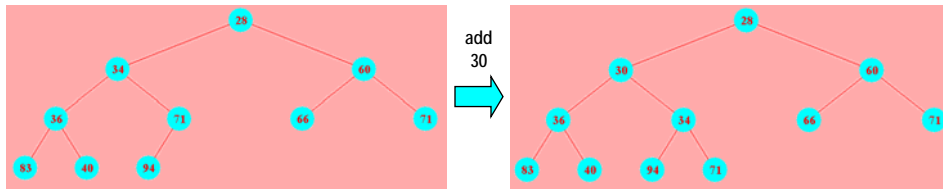
6

## Inserting into a heap

### to insert into a heap

- place new item in next open leaf position
- if new value is smaller than parent, then swap nodes
- continue up toward the root, swapping with parent, until smaller parent found

see <http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MinHeapAppl.html>



### note: insertion maintains completeness and the heap property

- worst case, if add smallest value, will have to swap all the way up to the root
- but only nodes on the path are swapped  $\rightarrow O(\text{height}) = O(\log N)$  swaps

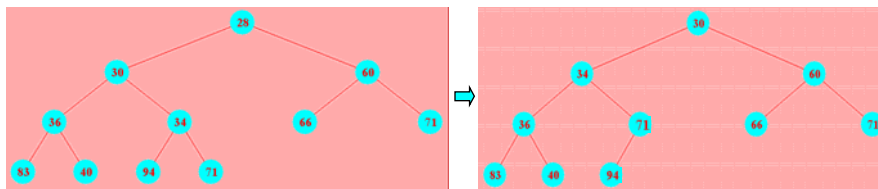
7

## Removing from a heap

### to remove the min value (root) of a heap

- replace root with last node on bottom level
- if new root value is greater than either child, swap with smaller child
- continue down toward the leaves, swapping with smaller child, until smallest

see <http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MinHeapAppl.html>



### note: removing root maintains completeness and the heap property

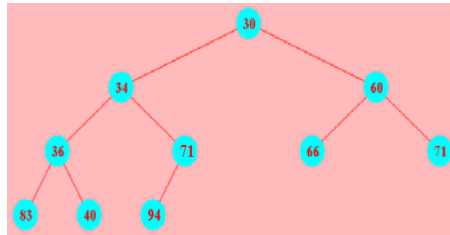
- worst case, if last value is largest, will have to swap all the way down to leaf
- but only nodes on the path are swapped  $\rightarrow O(\text{height}) = O(\log N)$  swaps

8

## Implementing a heap

a heap provides for  $O(1)$  find min,  $O(\log N)$  insertion and min removal

- also has a simple, List-based implementation
- since there are no holes in a heap, can store nodes in an ArrayList, level-by-level



30	34	60	36	71	66	71	83	40	94
----	----	----	----	----	----	----	----	----	----

- root is at index 0
- last leaf is at index `size()-1`
- for a node at index  $i$ , children are at  $2*i+1$  and  $2*i+2$
- to add at next available leaf, simply add at end

9

## SimpleMinHeap class

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class SimpleMinHeap<E extends Comparable<? super E>> {
    private ArrayList<E> values;

    public SimpleMinHeap() {
        this.values = new ArrayList<E>();
    }

    public E minValue() {
        if (this.values.size() == 0) {
            throw new NoSuchElementException();
        }
        return this.values.get(0);
    }

    public void add(E newValue) {
        values.add(newValue);
        int pos = this.values.size()-1;

        while (pos > 0) {
            if (newValue.compareTo(this.values.get((pos-1)/2)) < 0) {
                values.set(pos, this.values.get((pos-1)/2));
                pos = (pos-1)/2;
            }
            else {
                break;
            }
        }
        this.values.set(pos, newValue);
    }
    . . .
}
```

we can define our own simple min-heap implementation

- `minValue` returns the value at index 0
- `add` places the new value at the next available leaf (i.e., end of list), then moves upward until in position

10

## SimpleMinHeap class (cont.)

```
...  
public void remove() {  
    E newValue = this.values.remove(this.values.size()-1);  
    int pos = 0;  
  
    if (this.values.size() > 0) {  
        while (2*pos+1 < this.values.size()) {  
            int minChild = 2*pos+1;  
            if (2*pos+2 < this.values.size() &&  
                this.values.get(2*pos+2).compareTo(this.values.get(2*pos+1)) < 0) {  
                minChild = 2*pos+2;  
            }  
  
            if (newValue.compareTo(this.values.get(minChild)) > 0) {  
                this.values.set(pos, this.values.get(minChild));  
                pos = minChild;  
            }  
            else {  
                break;  
            }  
        }  
        this.values.set(pos, newValue);  
    }  
}
```

- remove removes the last leaf (i.e., last index), copies its value to the root, and then moves downward until in position

11

## Heap sort

the priority queue nature of heaps suggests an efficient sorting algorithm

- start with the ArrayList to be sorted
- construct a heap out of the elements
- repeatedly, remove min element and put back into the ArrayList

```
public static <E extends Comparable<? super E>>  
void heapSort(ArrayList<E> items) {  
    SimpleMinHeap<E> itemHeap = new SimpleMinHeap<E>();  
  
    for (int i = 0; i < items.size(); i++) {  
        itemHeap.add(items.get(i));  
    }  
  
    for (int i = 0; i < items.size(); i++) {  
        items.set(i, itemHeap.minValue());  
        itemHeap.remove();  
    }  
}
```

- N items in list, each insertion can require  $O(\log N)$  swaps to reheapify  
→ construct heap in  $O(N \log N)$
- N items in heap, each removal can require  $O(\log N)$  swap to reheapify  
→ copy back in  $O(N \log N)$

thus, overall efficiency is  $O(N \log N)$ , which is as good as it gets!

- can also implement so that the sorting is done in place, requires no extra storage

12

## Another application: data compression

in a multimedia world, document sizes continue to increase

- a high-quality 3.3 megapixel digital picture is ~2 MB
- an MP3 song is ~3-6 MB
- a full-length movie (now downloadable from iTunes) is ~1.2 GB

storing multimedia files can take up a lot of disk space

- perhaps more importantly, downloading multimedia requires significant bandwidth

it could be a lot worse!

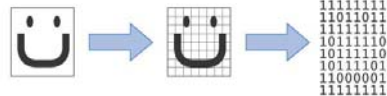
- image/sound/video formats rely heavily on data compression to limit file size  
e.g., if no compression, 3.3 megapixels \* 3 bytes/pixel = ~10 MB
- the JPEG format provides 10:1 to 20:1 compression without visible loss

13

## Image compression

the natural digital representation of an image is a *bitmap*

- for black & white images, can use a single bit for each pixel:  
0 = black, 1 = white
- for color images, store intensity for each RGB component, typically from 0-255  
e.g., purple is (128,0,128)



note: color requires 24x space as B&W

a variety of image formats exist

- some (GIF, PNG, TIFF, ...) are *lossless*, meaning that no information is lost during the compression – typically used for diagrams & drawings  
use techniques such as *run-length encoding*, where similar pixels are grouped  
e.g., above bitmap: 10-1-2-1-11-1-5-1-1-1-5-1-1-1-4-1-3-5-9
- others (JPEG, XPM, ...) are *lossy*, meaning that some information is lost – typically used for photographs where some loss of precision is not noticeable  
use techniques such as rounding values, combining close-but-not-identical pixels

14

## Audio, video, & text compression

audio & video compression algorithms similarly rely on domain-specific tricks

- e.g., in most videos, very little changes from one frame to the next  
need only store the initial frame and changes in each successive frame
- e.g., MP3 files remove sound out of human hearing range, overlapping noises

what about text files?

- in the absence of domain-specific knowledge, can't do better than a fixed-width code  
e.g., ASCII code uses 8-bits for each character

```
'0': 00110000  'A': 01000001  'a': 01100001
'1': 00110001  'B': 01000010  'b': 01100010
'2': 00110010  'C': 01000011  'c': 01100011
.
.
.
```

15

## Fixed- vs. variable-width codes

suppose we had a document that contained only the letters a-f

- with a fixed-width code, would need 3 bits for each character

```
a 000      d 011
b 001      e 100
c 010      f 101
```

- if the document contained 100 characters,  $100 * 3 = 300$  bits required

however, suppose we knew the distribution of letters in the document

a:45, b:13, c:12, d:16, e:9, f:5

- can customize a variable-width code, optimized for that specific file

```
a 0        d 111
b 101      e 1101
c 100      f 1100
```

- requires only  $45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224$  bits

16

## Huffman codes

Huffman compression is a technique for constructing an optimal\* variable-length code for text

- optimal in that it represents a specific file using the fewest bits (among all symbol-for-symbol codes)

Huffman codes are also known as prefix codes

- no individual code is a prefix of any other code

a	0	d	111
b	101	e	1101
c	100	f	1100

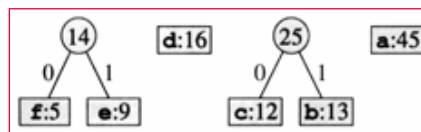
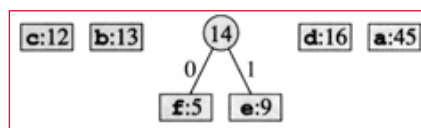
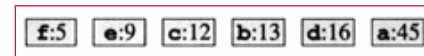
- this makes decompression unambiguous: 1010111110001001101
- *note:* since the code is specific to a particular file, it must be stored along with the compressed file in order to allow for eventual decompression

17

## Huffman trees

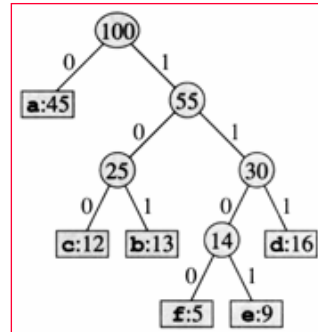
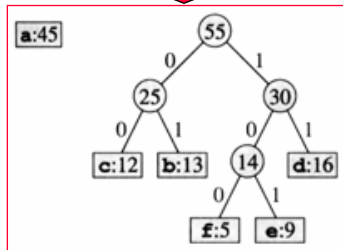
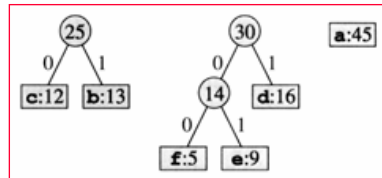
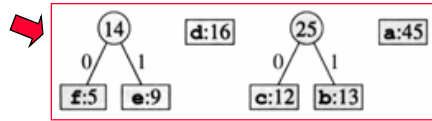
to construct a Huffman code for a specific file, utilize a greedy algorithm to construct a *Huffman tree*:

1. process the file and count the frequency for each letter in the file
2. create a single-node tree for each letter, labeled with its frequency
3. repeatedly,
  - a. pick the two trees with smallest root values
  - b. combine these two trees into a single tree whose root is labeled with the sum of the two subtree frequencies
4. when only one tree remains, can extract the codes from the Huffman tree by following edges from root to each leaf (left edge = 0, right edge = 1)



18

## Huffman tree construction (cont.)



the code corresponding to each letter can be read by following the edges from the root: left edge = 0, right edge = 1

a: 0	d: 111
b: 101	e: 1101
c: 100	f: 1100

19

## Huffman code compression

note that at each step, need to pick the two trees with smallest root values

- perfect application for a priority queue (i.e., a min-heap)
- store each single-node tree in a priority queue
- repeatedly,
  - remove the two min-value trees from the priority queue
  - combine into a new tree with sum at root and insert back into priority queue

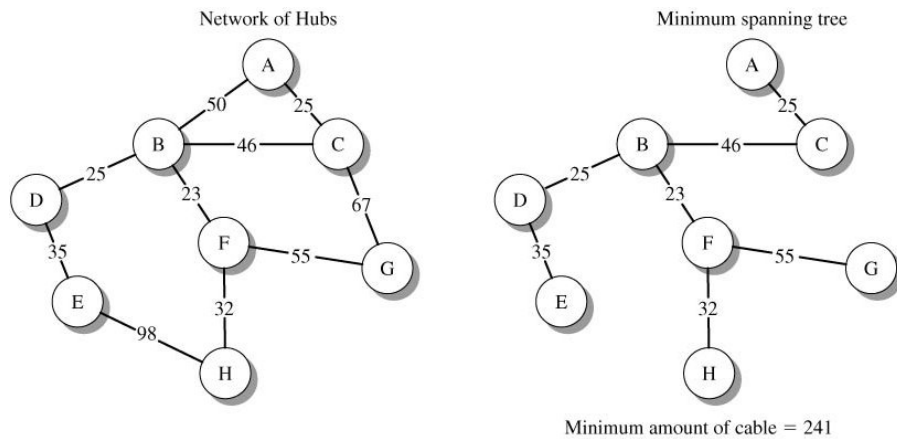
while designed for compressing text, it is interesting to note that Huffman codes are used in a variety of applications

- the last step in the JPEG algorithm, after image-specific techniques are applied, is to compress the resulting file using a Huffman code

20

## Minimum spanning tree

suppose you have a network with redundant connections, and want to find minimal wiring that connects the entire network



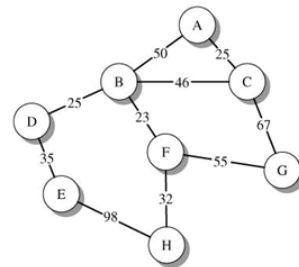
21

## Prim's algorithm

*Prim's algorithm* is a greedy algorithm for finding a minimum spanning tree

- i.e., a collection of edges that connects all the nodes, with minimum length/cost

1. start with a list of all nodes and an empty tree
2. as long as the node list is nonempty, repeatedly
  - a. select the node from the list with the shortest edge connecting it to a node in the tree
  - b. add that node and edge to the tree, and remove the node from the list



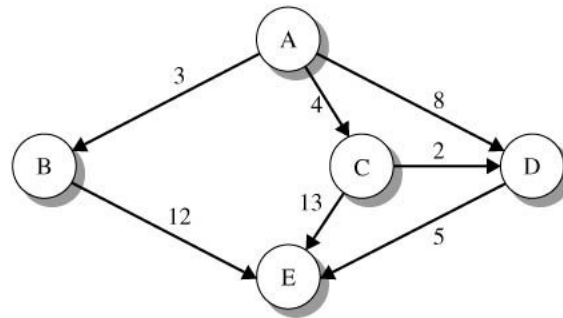
once again, could use a priority queue (i.e., a min-heap) to store the edges and select the shortest edge

see [students.ceid.upatras.gr/~papage1/project/prim.htm](http://students.ceid.upatras.gr/~papage1/project/prim.htm) for an animation

22

## Shortest path

suppose you were given a network and wanted to find the optimal path that a message should travel between two specific computers



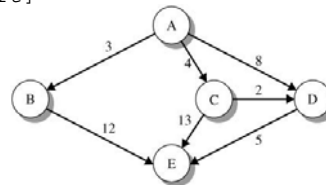
23

## Shortest path algorithms

a variety of algorithms can find the shortest path from start to end

backtracking-style solution:

1. start with a list (priority queue?) containing path: [start]
2. repeatedly
  - a. select the shortest path in the list:  
[start, node<sub>1</sub>, ..., node<sub>i</sub>]
  - b. if node<sub>i</sub> = end, then done
  - c. if not, expand that path with each neighbor of node<sub>i</sub> and add the resulting paths to the list



this solution is inefficient due to redundancy

- may end up storing the same subpaths multiple times – potentially would have to generate and test every possible path (whose length  $\leq$  optimal)
- *Dijkstra's algorithm* does this in a more intelligent way
  - combines greedy with dynamic programming to achieve  $O(n^2)$ ,  $n$  is # of nodes
  - maintains a table of shortest distances so far from start to each node
  - as it expands, it updates the distances so that no sub-optimal subpath is used

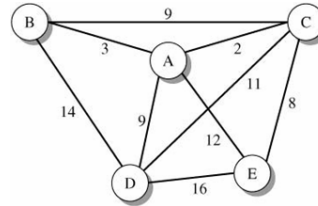
24

## Intractability

despite the speed of modern computers, there are still problems whose solutions are intractable

e.g., Traveling Salesperson Problem (TSP)

- given a map of cities and roads, what is the shortest path that visits each city exactly once and then returns to the starting city?



TSP has been classified as an *NP-hard problem*

- if given a solution, can verify it in polynomial time, i.e.,  $O(n^2)$
- but, there is no known polynomial-time algorithm for finding an optimal solution
- significant research has gone into finding efficient solutions to NP-hard problems like TSP (without success)
  
- since there are  $(n-1)!$  possible tours, generate-and-test will take a long time on any non-trivial graph
  - 5! = 120      10! = 3,628,800      20! = 2,432,902,008,176,640,000
- dynamic programming can avoid some redundancy, but still  $O(n!)$

25