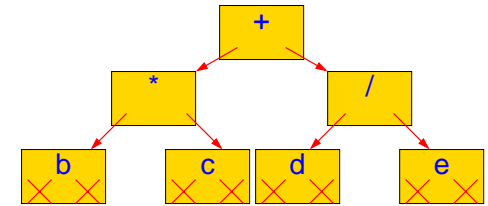


COMP2100 Lecture 27

- Warm-up: trees
 - preorder, inorder, postorder traversals
 - implementation using an explicit stack
 - breadth-first search using queues
- Huffman coding and Huffman trees

Jeffrey H. Kingston, *Algorithms and Data Structures: Design, Correctness, Analysis*, second edition, Addison-Wesley, 1997.

Basic traversals using recursion (material from COMP1110 lectures 19–21)



Preorder + * b c / d e
Inorder b * c + d / e
Postorder b c * d e / +

COMP2100

1

COMP2100

2

COMP2100

3

```
public class Tree<T> {
    public T item;
    public Tree<T> left;
    public Tree<T> right;

    public Tree(T item, Tree<T> left, Tree<T> right) {
        this.item = item;
        this.left = left;
        this.right = right;
    }
}
```

```
public void preorderDisplay() {
    preorder(this);
    System.out.println();
}

public void preorder(Tree<T> p) {
    if (p != null) {
        System.out.print(p.item + " ");
        preorder(p.left);
        preorder(p.right);
    }
}
```

```
public void inorderDisplay() {
    inorder(this);
    System.out.println();
}

public void inorder(Tree<T> p) {
    if (p != null) {
        inorder(p.left);
        System.out.print(p.item + " ");
        inorder(p.right);
    }
}
```

COMP2100

4

COMP2100

5

COMP2100

6

Preorder using an explicit stack

```
public void postorderDisplay() {
    postorder(this);
    System.out.println();
}

public void postorder(Tree<T> p) {
    if (p != null) {
        postorder(p.left);
        postorder(p.right);
        System.out.print(p.item + " ");
    }
}
}
```

COMP2100

7

COMP2100

8

COMP2100

9

```
public void
preorderStackDisplay() {
    preorderStack(this);
    System.out.println();
}

public void
preorderStack(Tree<T> p) {
    Tree<T> n;
    Stack<Tree<T>> s;
    n = p;
    s = new Stack<Tree<T>>();

    while (n != null) {
        System.out.print(n.item + " ");
        if (n.right != null)
            s.push(n.right);
        n = n.left;
        if (n == null && !s.empty()) {
            n = s.top();
            s.pop();
        }
    }
}
```

Exercise for the reader: how about inorder and postorder?

Queues

```
public class Queue<T> {
    public Queue() {}
    public boolean empty() {}
    public T head() {}
    public void enqueue(T end) {}
    public void dequeue() {}
}
```

COMP2100

10

COMP2100

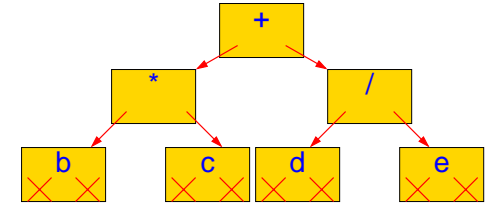
11

COMP2100

12

Exercise for the reader:
implement the **QUEUE** class

Breadth-first search



Preorder	+ * b c / d e
Inorder	b * c + d / e
Postorder	b c * d e / +
Breadth-first search	+ * / b c d e

Implementation of breadth-first search using a queue

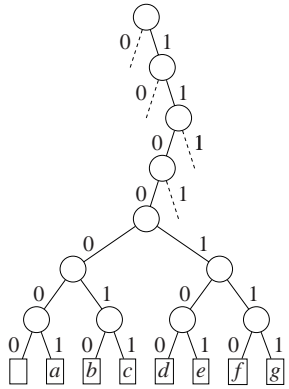
```

public void
breadthFirstSearchDisplay() {
    breadthFirstSearch(this);
    System.out.println();
}

public void
breadthFirstSearch(Tree<T> p)
{
    Tree<T> n;
    Queue<Tree<T>> q;
    q = new Queue<Tree<T>>();

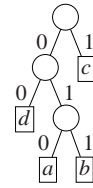
    if (p != null)
        q.enqueue(p);
    while (!q.empty()) {
        n = q.head();
        q.dequeue();
        System.out.print(n.item + " ");
        if (n.left != null)
            q.enqueue(n.left);
        if (n.right != null)
            q.enqueue(n.right);
    }
}
    
```

Huffman codes and Huffman trees



$a = 1100001$
 $b = 1100010$
 $c = 1100011$
 $d = 1100100$
 $e = 1100101$
 $f = 1100110$
 $g = 1100111$

add = 110000111001001100100



$d = 00$
 $a = 010$
 $b = 011$
 $c = 1$

add = 0100000

To decode: start at the top of the tree, follow the branches until you reach a leaf, then return to the root and repeat.

It works because it's a prefix encoding: no code is a prefix (initial subsequence) of any other code.

$d = 00, a = 010, b = 011, c = 1$

cdabcdcc

d occurs 3 times and has code length 2,
a occurs 1 time and has code length 3, ...

Total length = $3 \times 2 + 1 \times 3 + 1 \times 3 + 5 \times 1 = 17$

This is like traversing the encoding tree and counting the total number of edges followed.

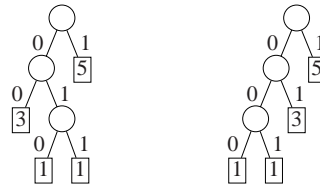
The optimal prefix encoding problem: given a fixed sequence of characters (the string to be encoded) find an encoding tree that minimizes the length of the encoding of the sequence.

Reformulate as a problem with *weights*:

Weights are integers stored in leaf nodes (here corresponding to the number of occurrences of the character). Find a tree with the minimum *weighted leaf path length*.


Huffman trees (D. A. Huffman, 1952)


Example: given weights $\{1, 1, 3, 5\}$, these are two trees with minimum weighted leaf path length:

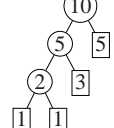


In both cases, the weighted leaf path length = 17.

$F = \{ \boxed{3}, \boxed{1}, \boxed{1}, \boxed{5} \}$

$F = \{ \boxed{3}, \textcircled{2}, \boxed{5} \}$


$F = \{ \textcircled{5}, \boxed{5} \}$


$F = \{ \textcircled{10} \}$


Priority queues

Priority queues

- Each queue entry now has an associated priority (lower number = higher priority).
- The head of the queue is always an element with highest priority.
- Used in many applications, especially simulations.

```
public class PriorityQueueHeapEntry<K extends Comparable, V> {  
  
    public PriorityQueueHeapEntry(K k, V v) {}  
  
    public K key() {}  
  
    public V value() {}  
}
```

```
public class PriorityQueueHeap<K extends Comparable, V> {  
  
    public PriorityQueueHeap() {}  
  
    public static <K extends Comparable, V>  
        PriorityQueueHeapEntry<K, V> newEntry(K k, V v) {}  
  
    public boolean empty() {}  
  
    public PriorityQueueHeapEntry<K, V> head() {}  
  
    public void enqueue(PriorityQueueHeapEntry<K, V> end) {}  
  
    public void dequeue() {}  
}
```

Huffman implementation using a priority queue

```
public static Tree<Integer> huffman(List<Integer> weights) {  
    List<Integer> e;  
    PriorityQueueHeap<Integer, Tree<Integer>> q;  
    PriorityQueueHeapEntry<Integer, Tree<Integer>> e1, e2;  
    Tree<Integer> t;  
    q = new PriorityQueueHeap<Integer, Tree<Integer>>();  
    e = weights.first();  
    while (e != null) {  
        t = new Tree<Integer>(e.value(), null, null);  
        q.enqueue(newEntry(e.value(), t));  
        e = e.next();  
    }  
    e1 = q.head();  
}
```

```
q.dequeue();
while (!q.empty()) {
    e2 = q.head();
    q.dequeue();
    t = new Tree<Integer>(e1.key()+e2.key(),e1.value(),e2.value());
    q.enqueue(newEntry(e1.key()+e2.key(),t));
    e1 = q.head();
    q.dequeue();
}
return e1.value();
}
```