Applications of Trees

- First application: coding and data compression
- We will define optimal variable-length binary codes and code trees
- We will study Huffman's algorithm which constructs them
- Huffman's algorithm is an example of a Greedy Algorithms, an important class of simple optimization algorithms

Text, Codes, and Compression

- Computer systems represent data as bit strings
- Encoding: transformation of data into bit strings
- Decoding: transformationof bit strings into data
- The code defines the transformation

Text, Codes, and Compression

- For example: ASCII, the international coding standard, uses a 7-bit code
- HEX Code - Character
- 20 - <space>
- 41 - A
- 42 - B
- 61 - a

Text, Codes, and Compression

- Such encodings are called
  - fixed-length or
  - block codes
- They are attractive because the encoding and decoding is extremely simple
  - For coding, we can use a block of integers or codewords indexed by characters
  - For decoding, we can use a block of characters indexed by codewords

Text, Codes, and Compression

- For example: the sentence
  The cat sat on the mat

  is encoded in ASCII as

  1010100 110100 011001 0101 .....
- Note that the spaces are there simply to improve readability ... they don't appear in the encoded version.

Text, Codes, and Compression

- The following bit string is an ASCII encoded message:

  1000100110010111000111101111110
  0100110100111011101100111010000
  0110100111100110100000110010111
  0000111100111111001

Text, Codes, and Compression

- And we can decode it by chopping it into smaller strings eachs of 7 bits in length and by replacing the bit strings with their corresponding characters:

  1000100(D)1100101(e)1100011(c)1101111(o)1100100(d)1101001(i)1101110(n)1100111(g)0100000()1101001(i)1110011(s)0100000()1100101(e)1100001(a)1110011(s)1111001(y)

Text, Codes, and Compression

- Every code can be thought of in terns of
- a finite alphabet of source symbols
- a finite alphabet of code symbols
- Each code maps every finite sequence or string of source symbols into a string of code symbols

Text, Codes, and Compression

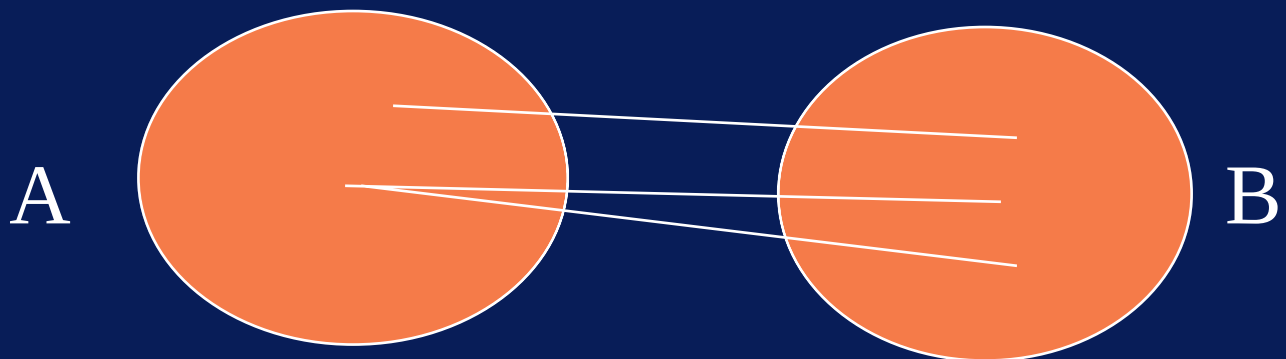
- Let A be the source alphabet
- Let B be the code alphabet
- A code f is an injective map

$$f: S_A \rightarrow S_B$$

- where S_A is the set of all strings of symbols from A
- where S_B is the set of all strings of symbols from B

Text, Codes, and Compression

- Injectivity ensures that each encoded string can be decoded uniquely (we do not want two source strings that are encoded as the same string)



Injective Mapping: each element in the range is related to at most one element in the domain

Text, Codes, and Compression

- We are primarily interested in the code alphabet {0, 1} since we want to code source symbols strings as bit strings

Text, Codes, and Compression

- There is a problem with block codes: n symbols produce nb bits with a block code of length b
- For example,
  - if n = 100,000 (the number of characters in a typical 200-page book)
  - b = 7 (e.g. 7-bit ASCII code)
  - then the characters are encoded as 700,000 bits

Text, Codes, and Compression

- While we cannot encode the ASCII characters with fewer than 7 bits
- We can encode the characters with a different number of bits, depending on their frequency of occurence.
- Use fewer bits for the more frequent characters
- Use more bits for the less frequent characters
- Such a code is called a variable-length

Text, Codes, and Compression

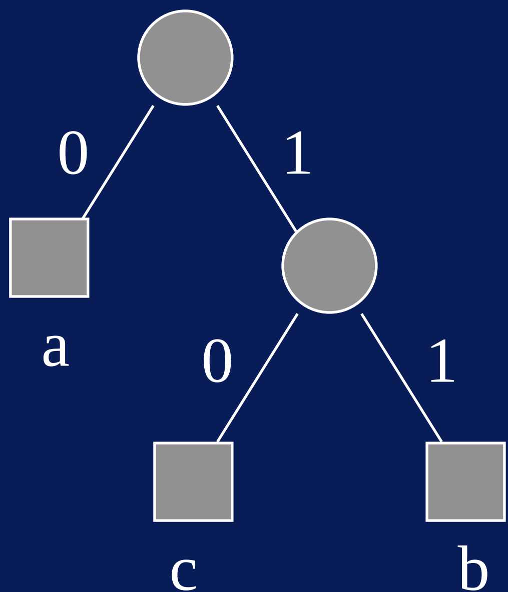
- First problem with variable length codes:
  - when scanning an encoded text from left to right (decoding it)
  - How do we know when one codeword finishes and another starts?
- We require each codeword not be a prefix of any other codeword
- So, for the binary code alphabet, we should base the codes on binary code

Text, Codes, and Compression

- Binary code trees:
- binary tree whose external nodes are labelled uniquely with the source alphabet symbols
- Left branches are labelled 0
- Right branches are labelled 1

Text, Codes, and Compression

A binary code tree and its prefix code



a 0
b 11
c 10

Text, Codes, and Compression

- The codeword corresponding to a symbol is the bit string given by the path from the root to the external node labeled with the symbol
- Note that, as required, no codeword is a prefix for any other codeword
  - This follows directly from the fact that source symbols are only on external nodes
  - and there is only one (unique) path to that symbol

Text, Codes, and Compression

- Codes that satisfy the prefix property are called prefix codes
- Prefix codes are important because
  - we can uniquely decode an encoded text with a left-to-right scan of the encoded text
  - by consideringly only the current bit in the encoded text
  - decoder uses the code tree for this purpose

Text, Codes, and Compression

- Read the encoded message bit by bit
- Start at the root
- if the bit is a 0, move left
- if the bit is a 1, move right
- if the node is external, output the corresponding symbol and begin again at the root

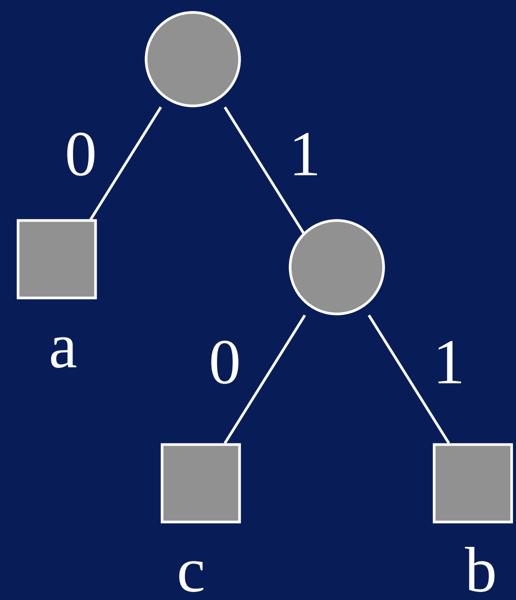
Text, Codes, and Compression

- Encoded message:

  0 0 1 1 1 0 0

- Decoded message:

  A A B C A



Optimal Variable-Length Codes

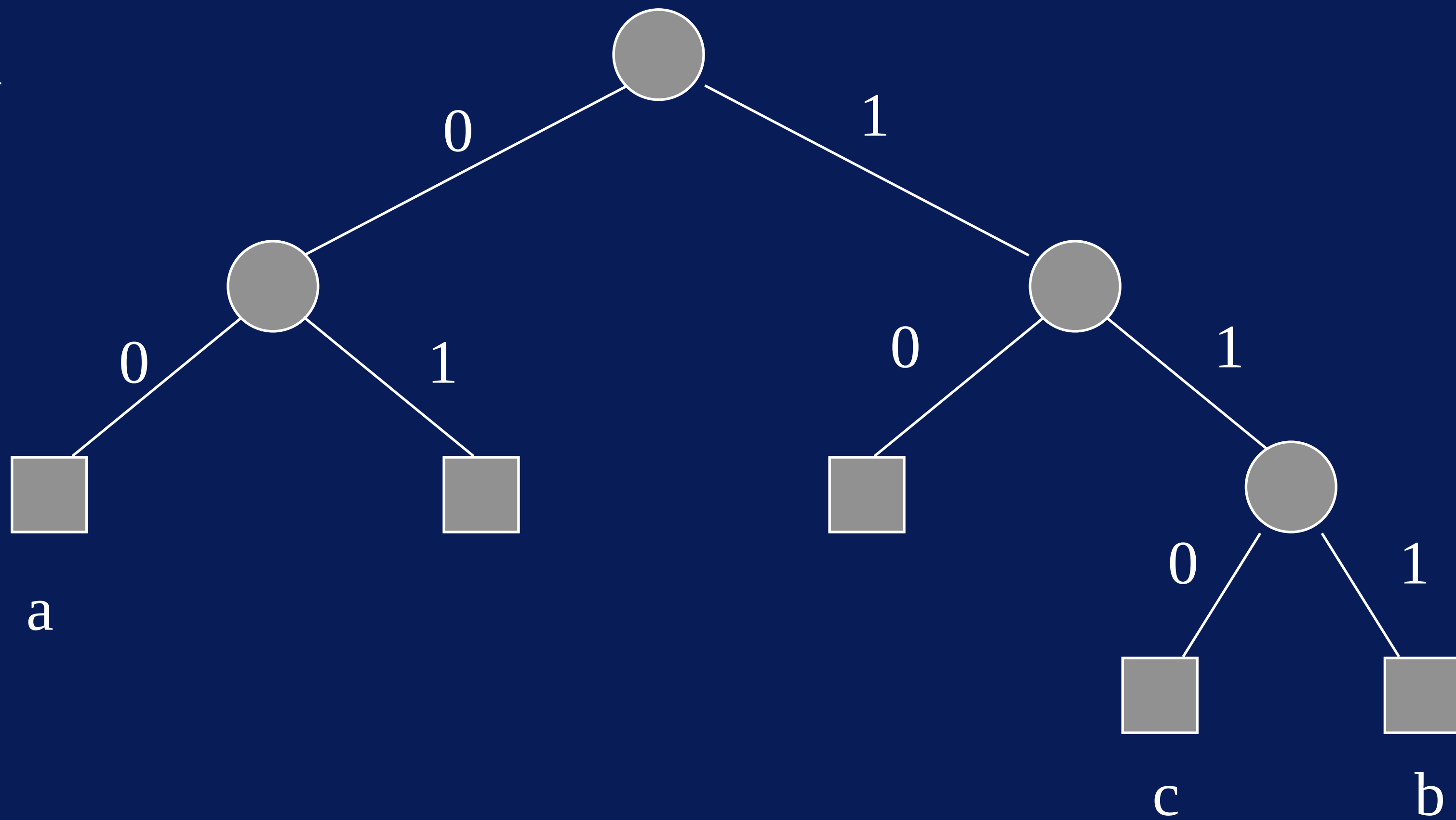
- What makes a good variable length code?
- Let A = $a_1, ..., a_n$, n>=1, be the alphabet of source symbols
- Let P = $p_1, ..., p_n$, n>=1, be their probability of occurrence
- We obtain these probabilities by analysing are representative sample of the type of text we wish to encode

Optimal Variable-Length Codes

- Any binary tree with n external nodes labelled with the n symbols defines a prefix code
- Any prefix code for the n symbols defines a binary tree with at least n external nodes
- Such a binary tree with exactly n external nodes is a reduced prefix code (tree)
- Good prefix codes are always reduced

Non-Reduced Prefix Code (Tree)

a 000
b 111
c 110



Optimal Variable-Length Codes

- Comparison of prefix codes - compare the number of bits in the encoded text
- Let $A = a_1, \ldots, a_n$, $n>=1$, be the alphabet of source symbols
- Let $P = p_1, \ldots, p_n$ be their probability of occurrence
- Let $W = w_1, \ldots, w_n$ be a prefix code for $A = a_1, \ldots, a_n$
- Let $L = l_1, \ldots, l_n$ be the lengths of $W = w_1, \ldots, w_n$

Optimal Variable-Length Codes

- Given a source text T with $f_1, \ldots, f_n$ occurrences of $a_1, \ldots, a_n$ respectively
- The total number of bits when T is encoded is
  $\sum_{i=1}^{n} f_i l_i$
- The total number of source symbols is
  $\sum_{i=1}^{n} f_i$
- The average length of the W-encoding is
  $\text{Alength}(T, W) = \sum_{i=1}^{n} f_i l_i \; / \; \sum_{i=1}^{n} f_i$

Optimal Variable-Length Codes

- For long enough texts, the probability p_i of a given symbol occurring is approximately

  $$p_i = f_i \;/\; \sum_{i=1}^{n} f_i$$

- So the expected length of the W-encoding is

  $$\text{Elength}(W, P) = \sum_{i=1}^{n} p_i \, l_i$$

Optimal Variable-Length Codes

- To compare two different codes W_1 and W_2 we can compare either
  - Alength(T, W_1) and Alength(T, W_2) or
  - Elength(W_1, P) and Elength(W_2 , P)
- We say W_1 is no worse than W_2 if

  Elength(W_1, P) <= Elength(W_2 , P)
- We say W_1 is optimal if

  Elength(W_1, P) <= Elength(W_2 , P)

  for all possible prefix codes W_2 of A

Optimal Variable-Length Codes

- Huffman's Algorithm
- We wish to solve the following problem:
- Given n symbols
  $A = a_1, ..., a_n, n>=1$
  and the probability of their occurrence
  $P = p_1, ..., p_n$, respectively,
  construct an optimal prefix code for A and P

Optimal Variable-Length Codes

- This problem is an example of a global optimization problem
- Brute force (or exhaustive search) techniques are too expensive to compute:

  Given A and P
  Compute the set of all reduced prefix codes
  Choose the minimal expected length

Optimal Variable-Length Codes

- This algorithm takes $O(n^n)$ time, where n is the size of the alphabet
- Why? because any binary tree of size n-1 (i.e. with n external nodes) is a valid reduced prefix tree and there are n! ways of labelling the external nodes
- Since n! is approximately n^n we see that there are approximately $O(n^n)$ steps to go through when constructing all the trees to check

Optimal Variable-Length Codes

- Huffman's Algorithm is only $O(n^2)$
- This is significant: if n = 128 (number of symbols in a 7-bit ASCII code)
- $O(n^n) = 128^{128} = 5.28 \times 10^{269}$
- $O(n^2) = 128^2 = 1.6384 \times 10^4$
- There are 31536000 seconds in a year and if we could compute 1000 000 000 steps a second then the brute force technique would still take 1.67×10^{253} years

Optimal Variable-Length Codes

- The age of the universe is estimated to be between 7 and 20 billion years, i.e.,

  $7x10^9$ and $20x10^9$ years

- A long way off 1.67 x 10^{253} years!

Optimal Variable-Length Codes

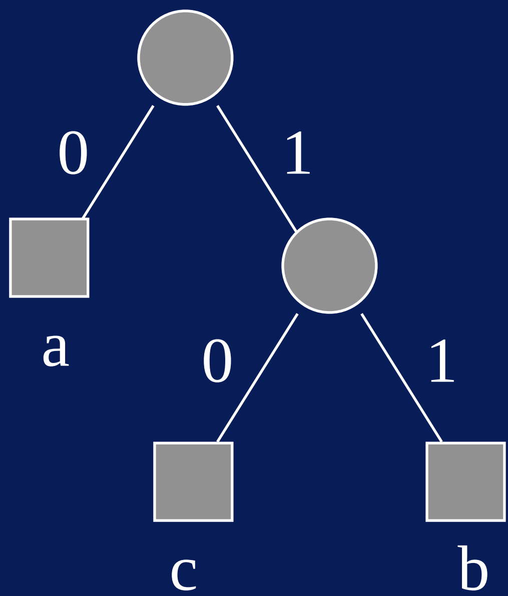
- Huffman's Algorithm uses a technique called Greediness
- It uses local optimization to achieve a globally optimum solution
  - Build the code incrementally
  - Reduce the code by one symbol at each step
  - Merge the two symbols that have the smallest probabilities into one new symbol

Optimal Variable-Length Codes

- Before we begin, note that we’d like a tree with the symbols which have the lowest probability to be on the longest path
- Why?
- Because the length of the codeword is equal to the path length and we want
  - short codewords for high-probability symbols
  - longer codewords for low-probability

Text, Codes, and Compression

A binary code tree and its prefix code



a 0
b 11
c 10

Huffman's Algorithm

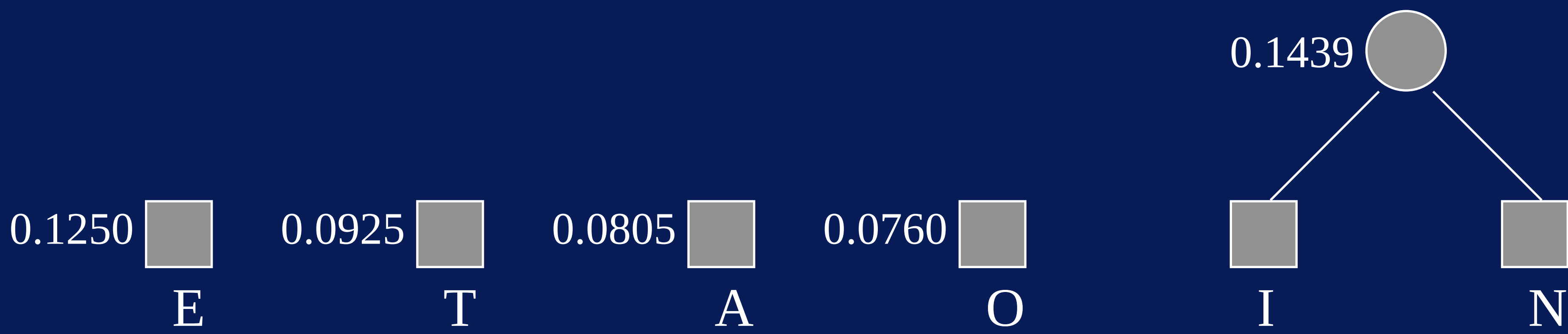
- We will treat Huffman's Algorithm for just six letters, i.e, n = 6, and there are six symbols in the source alphabet.
- These are, with their probabilities,
  - E - 0.1250
  - T - 0.0925
  - A - 0.0805
  - O - 0.0760
  - I - 0.0729
  - N - 0.710

Huffman’s Algorithm

- Step 1:
- Create a forest of code trees, one for each symbol
- Each tree comprises a single external node (empty tree) labelled with its symbol and weight (probability)

Huffman's Algorithm

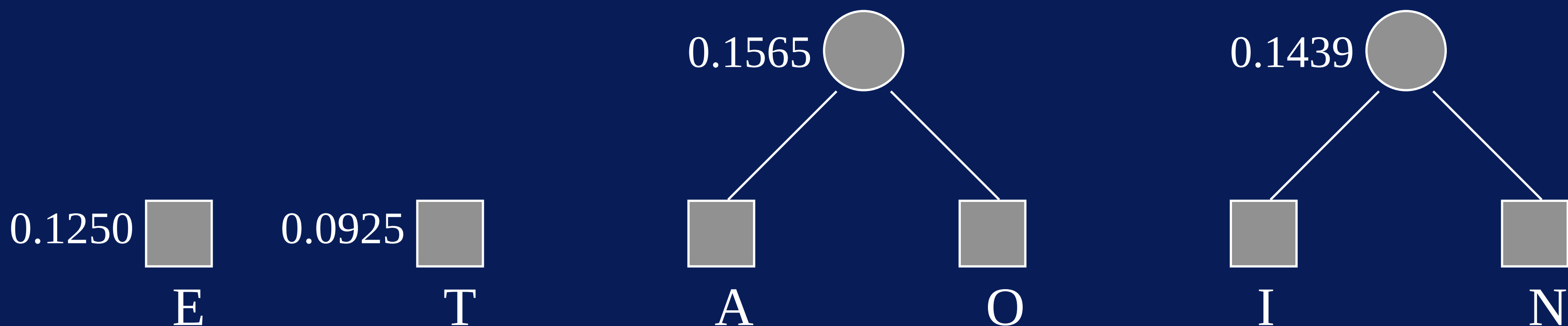
- Step 2:
  - Choose the two binary trees, B1 and B2, that have the smallest weights
  - Create a new root node with B1 and B2 as its children and with weight equal to the sum of these two weights



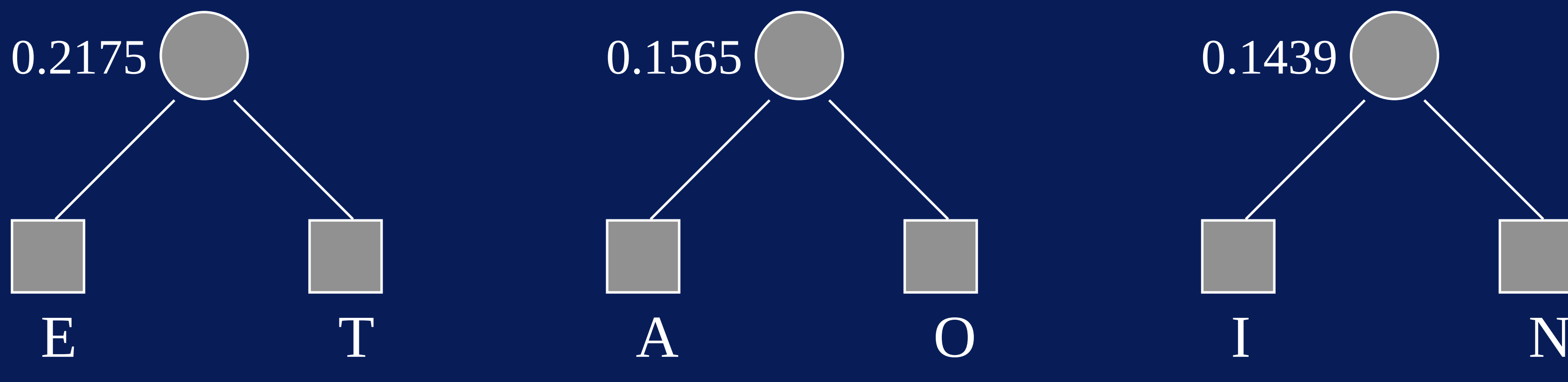
Huffman’s Algorithm

- Step 3:
  - Repeat step 2!

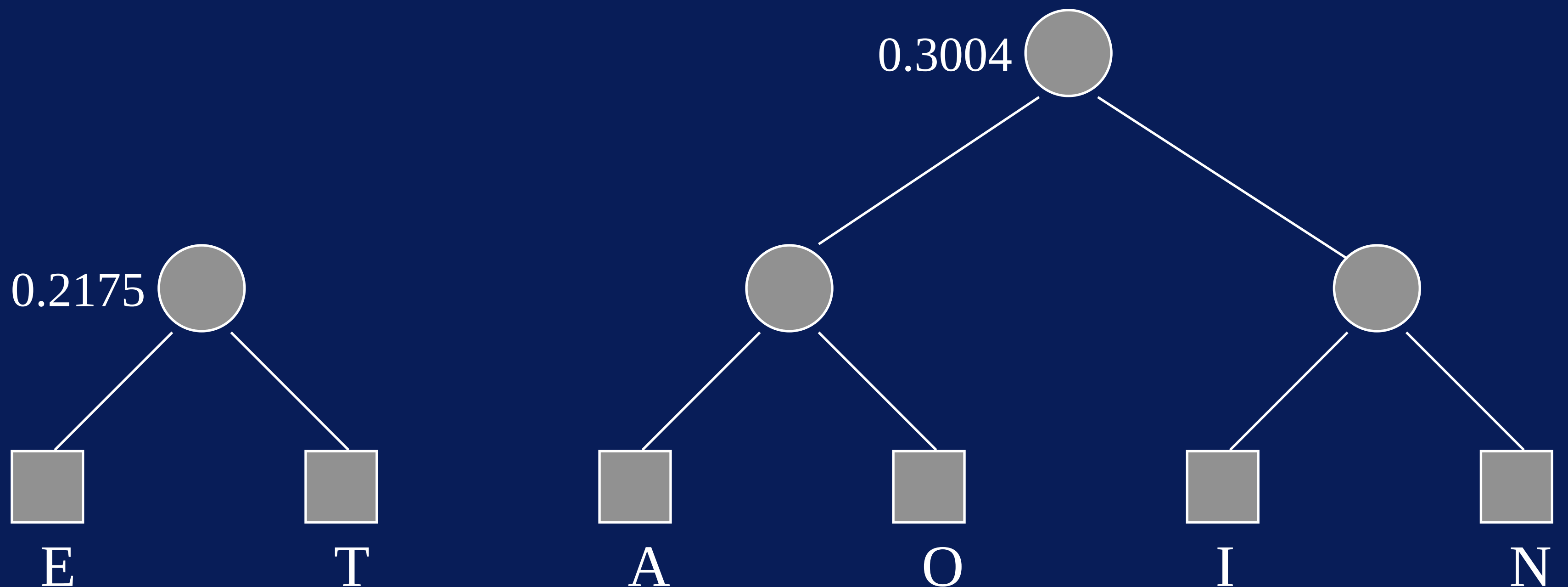
Huffman's Algorithm



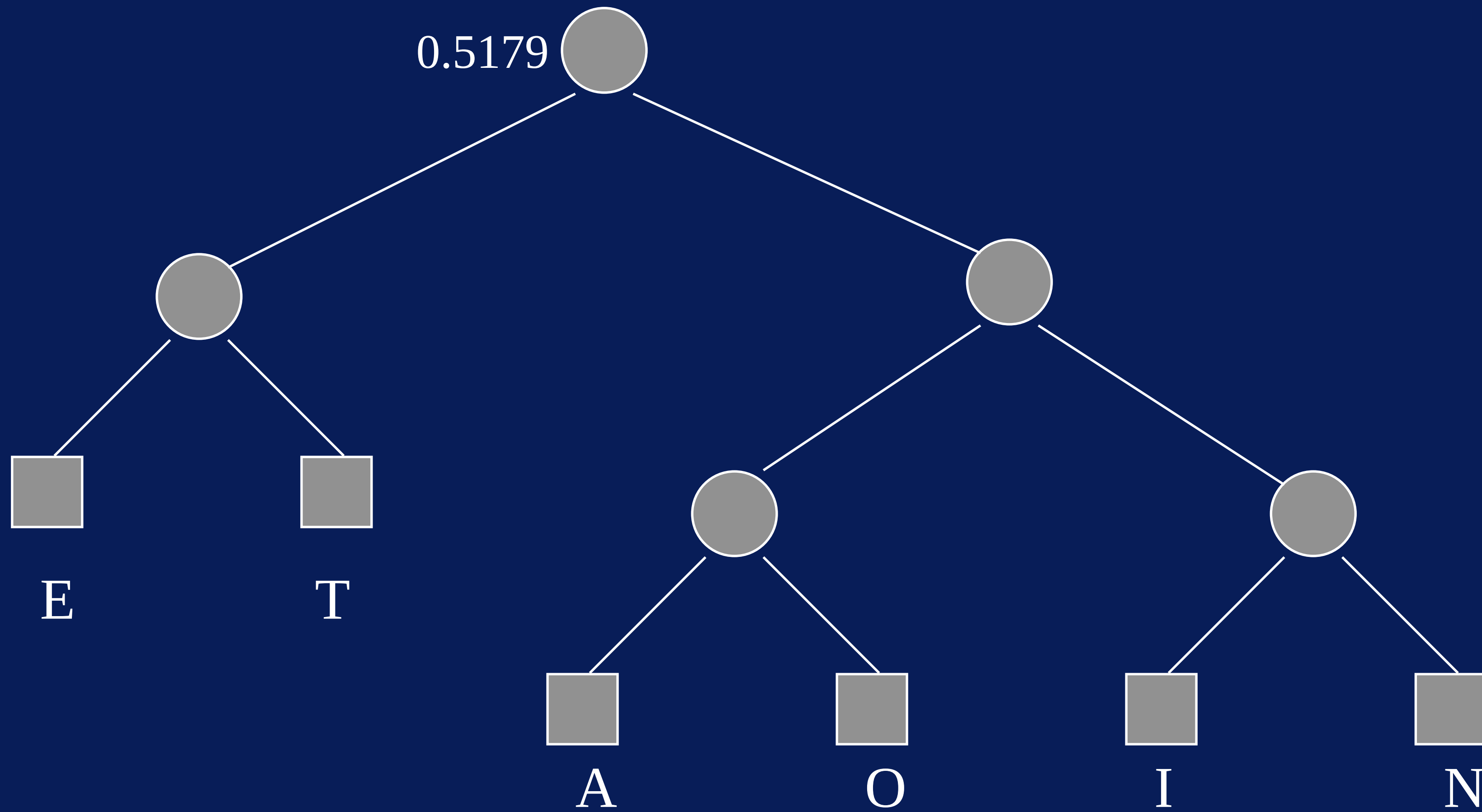
Huffman's Algorithm



Huffman's Algorithm



Huffman's Algorithm



Huffman's Algorithm

- The final prefix code is:
  - A 100
  - E 00
  - I 110
  - N 111
  - O 101
  - T 01

Huffman’s Algorithm

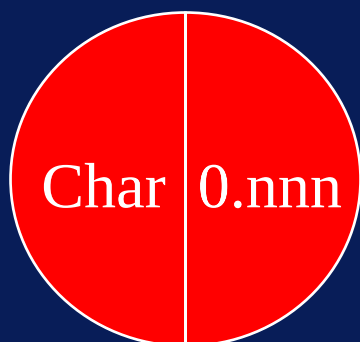
- Three phases in the algorithm
- Initialize the forest of code trees
- Construct an optimal code tree
- Compute the encoding map

Huffman’s Algorithm

- Phase 1: Initialize the forest of code trees
  - How will we represent the forest of trees?
  - Better question: how will we represent our tree ... have to store both alphanumeric characters and probabilities?
  - Need some kind of composite node
  - Opt to represent this composite node as an INTERNAL node

Huffman's Algorithm

- Consequently, the initial tree is simply one internal node
- That is, it is a root (with two external nodes)

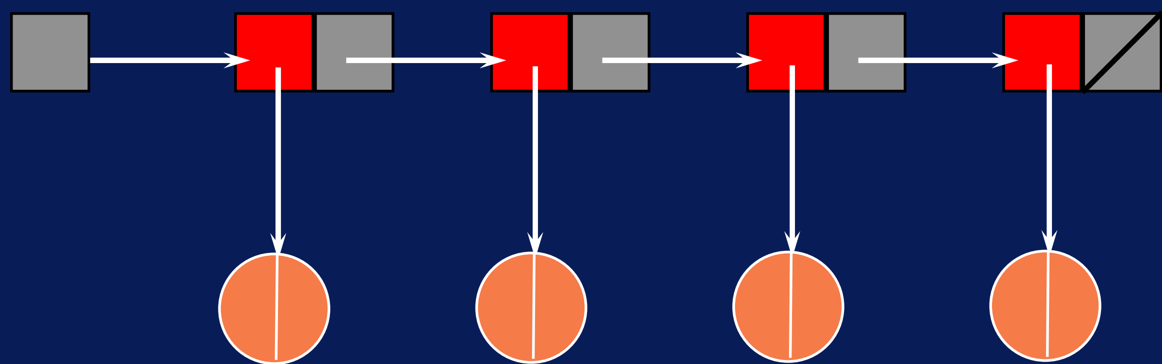


Huffman’s Algorithm

- So, to create such a tree we simply invoke the following operations:
  - Initialize the tree … tree()
  - Add a node … addnode(char, weight, T)

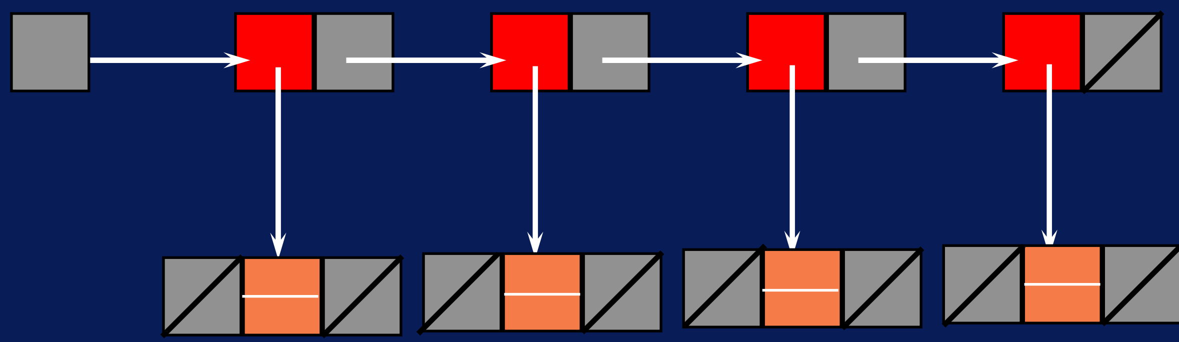
Huffman's Algorithm

- We must also keep track of our forest
- Could represent it as a linked list of pointers to Binary trees ...



Huffman's Algorithm

- Represented as:



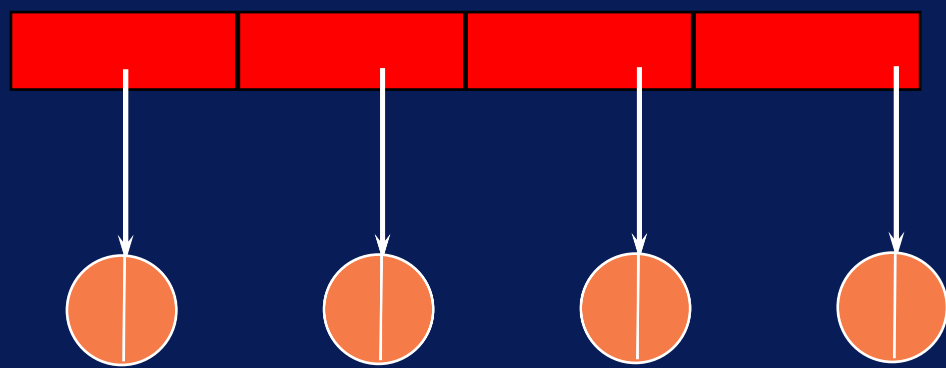
Huffman's Algorithm

- Is there an alternative?
- Question: why do we use dynamic datastructures?
- Answer:
  – When we don't know in advance how many elements are in our data set
  – When the number of elements varies significantly
- Is this the case here?
- No!

Huffman's Algorithm

- So, our alternatives are? ......
- An array, indexed by number, of type ...
- *binary_tree*, i.e., each element in the array can point to a binary code tree

Huffman's Algorithm



Huffman's Algorithm

- What will be the dimension of this array?
- n, the number of symbols in our source alphabet since this is the number of trees we start out with in our forest initially

Huffman’s Algorithm

- Phase 2: construct the optimal code tree

Huffman's Algorithm

Pseudo-code algorithm

Find the tree with the smallest weight - A, at element i

Find the tree with the next smallest weight - B, at element j

Construct a tree, with right sub-tree A, left sub-tree B, with root having weight = sum of the roots of A and B

Let array element i point to the new tree

Delete tree at element j

Huffman's Algorithm

let n be the number of trees initially

Repeat

```
Find the tree with the smallest weight - A, at
  element i
Find the tree with the next smallest weight - B,
  at element j

Construct a tree, with right sub-tree A, left
  sub-tree B, with root having weight = sum of
  the roots of A and B

Let array element i point to the new tree
Delete tree at element j
```

Until only one tree left in the array

Huffman's Algorithm

- Phase 3: Compute the encoding map
  - We need to write out a list of source symbols together with their prefix code
  - We need to write out the contents of each external node (or each frontier internal node) together with the path to that node
  - We need to **traverse** the binary code tree in some manner

- But …. we want to print out the symbol and the prefix code:

  i.e. the symbol at the leafnode

  and the path by which we got to that node

- How will we represent the path?
- As an array of binary values (representing the left and right links on the path)

Huffman's Algorithm

```
// new tree definition

struct node
{
   char symbol;
   float probability;
   node *pleft, *pright;
};
```

Huffman’s Algorithm

```
class tree
{
public:
    tree();
    ~tree();
    void add(int n) {addnode(n,root);}
    void print() {pr(root,0);}
    node* &search(int n);
    int delnode(int x);
private
    node *root;
    void deltree(node *p);
    void addnode(int n, node* &p);
    void pr(const node *p, int nspace) const;
};
```

Huffman's Algorithm

```
class tree // modified for this application
{
public:
    tree();
    ~tree();
    void add(char s, float p) {addnode(s,p,root);}
    void print() {pr(root,0);}
    node* &search(int n);
    int delnode(int x);
private
    node *root;
    void deltree(node* &p); // NB
    void addnode(char s, float p, node* &p);
    void pr(const node *p, int nspace) const;
};
```

Huffman's Algorithm

```
void tree::deltree(node* &p) {
// modified parameter to reference parameter
  if (p != NULL) {
    deltree(p->pleft);
    deltree(p->pright);
    delete p;
    p = NULL;  // return null pointer
  }
}
```

Huffman's Algorithm

```
class forest {
public:
  forest(int size);
  ~forest();
  void initialize_forest();
  void add_to_tree(int tree_number,
                   char symbol, float probability);
  void print_forest() const;
  void print_tree(int tree_number);
  void join_trees(int tree_1, int tree_2);
  int empty_tree(int tree_number);
  float root_probability(int tree_number);
private:
  tree tree_array[MAXIMUM_NUMBER_OF_TREES];
  int forest_size;
};
```

Huffman's Algorithm

```
// inorder traversal from previous part of the course
//
// recursive function to print the contents of the
// binary search tree

void tree::prorder(const node *p) const
{
   if (p!=NULL)
   {
      prorder(p->pleft);
      cout << p->data << " ";
      prorder(p->pright);
   }
}
```

Huffman’s Algorithm

```
// inorder traversal to print only leaf nodes

void tree::leafnode_traversal(const node *p) const
{
   if (p != NULL) {
      if (at_leafnode) {    // PSEUDO CODE
         visit this node
      }
      else {
         leafnode_traversal(p->pleft);
         leafnode_traversal(p->pright);
      }
   }
}
```

Huffman's Algorithm

```
// inorder traversal to print only leaf nodes

void tree::leafnode_traversal(const node *p) const
{
   if (p != NULL) {
      if ((p->pleft == NULL) &&
          (p->right == NULL)) {     // leafnode
         cout << p->symbol << p->probability <<endl;
      }
      else {
         leafnode_traversal(p->pleft);
         leafnode_traversal(p->pright);
      }
   }
}
```

Huffman's Algorithm

```
// pseudocode version of compute_map
// to traverse tree and print leaf node and path
// to leaf node

void tree::traverse_leaf_nodes(const node *p, path)
{
   if (at leaf node) {
      print out symbol and path
   }
   else {
      add_to_path(path, 0);  // left
      traverse_leaf_nodes(p->pleft, path);
      remove_element_from_path(path);
```

Huffman’s Algorithm

```
add_to_path(path, 1);  // right
traverse_leaf_nodes(p->pright, path);
remove_element_from_path(path);
```

Huffman’s Algorithm

```
// Definition of path

#define MAX_PATH_LENGTH 20
class path {
public:
   path();
   ~path();
   add_to_path(int direction);
   remove_from_path();
   print_path();
private:
   int path_components[MAX_PATH_LENGTH];
   int path_length;
}
```

Huffman’s Algorithm

```
// Definition of path

path::path()
{
   int i;
   for (i=0; i<MAX_PATH_LENGTH; i++) {
      path_components[i] =  0;
   }
   path_length = 0;
}
```

Huffman's Algorithm

```
// Definition of path

path::~path()
{
}
```

Huffman's Algorithm

```
// Definition of path

path::add_to_path(int direction)
{
    if (path_length < MAX_PATH_LENGTH) {
        path_components[path_length] = direction;
        path_length++;
    }
    else {
        cout << "Error maximum path length reached";
    }
}
```

Huffman’s Algorithm

```
// Definition of path

path::remove_from_path()
{
   if (path_length > 0) {
      path_length--;
   }
   else {
      cout << “Error: no path exists”;
   }
}
```

Huffman’s Algorithm

```
// Definition of path

path::print_path()
{
   for (i=0; i<path_length; i++) {
      cout << path_components[i];
   }
   cout << “ “;
}
```

Huffman’s Algorithm

```
// Definition of traverse_leaf_nodes
// to traverse tree and print leaf node and path
// to leaf node

void tree::traverse_leaf_nodes(const node *p, path &code)
{
  if (p != NULL) {
    if ( (p->pleft == NULL) &&
          (p->pright == NULL)) {  // leaf node
        cout << p->symbol << " ";
        code.print_path();
        cout << endl;
    }
    else {
```

Huffman’s Algorithm

```
        code.add_to_path(0);  // left
        traverse_leaf_nodes(p->pleft, code);
          code.remove_from_path();

          code.add_to_path(1);  // right
          traverse_leaf_nodes(p->pright, code);
          code.remove_from_path();
      }
    }
  }
```

Huffman's Algorithm

```
void tree::compute_map() {
                // new function to print leaf nodes
  path code;          // and the path to leaf nodes
  traverse_leaf_nodes(root, code);
}
```

Huffman’s Algorithm

```
void forest::compute_map() {
  int i;

  for (i=0; i<MAXIMUM_NUMBER_OF_TREES; i++) {
    if (tree_array[i].empty_tree() == FALSE) {
        tree_array[i].compute_map();
    }
  }
}
```