Algorithmics I

Section 1 – Sorting and Tries

Dr. Gethin Norman

School of Computing Science
University of Glasgow

gethin.norman@glasgow.ac.uk

Sorting – Recap

Naïve sorting algorithms: $O(n^2)$ in the worst/average case

– Selectionsort, Insertionsort, Bubblesort

Clever sorting algorithms: $O(n \log n)$ in the worst/average case

– Mergesort, Heapsort (which we have just seen)

The fastest sorting algorithm in practice is Quicksort

– $O(n \log n)$ on average

– but no better than $O(n^2)$ in the worst case (unless a clever variant is used)

Question: can we come up with a sorting algorithm that is better than $O(n \log n)$ in the worst case?

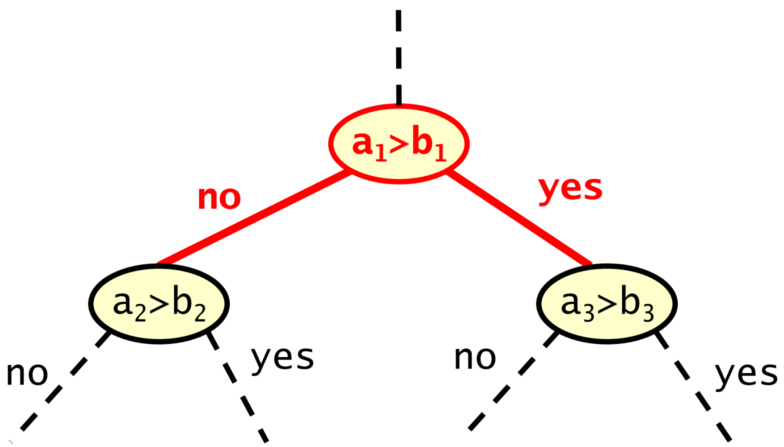
– for example a $O(n)$ algorithm

Sorting – Comparison based sorting

Claim: no sorting algorithm that is based on pairwise comparison of values can be better than `O(n log n)`

Justification: describe the algorithm by a **decision tree** (binary tree)

- each node represents a comparison between two elements
- path branches left or right depending on the outcome of the comparison



Sorting – Comparison based sorting

Claim: no sorting algorithm that is based on pairwise comparison of values can be better than `O(n log n)`

Justification: describe the algorithm by a **decision tree** (binary tree)

- each node represents a comparison between two elements
- path branches left or right depending on the outcome of the comparison
- an execution of the algorithm is a path from the root to a leaf node
- the number of leaf nodes in the tree must be at least the number of 'outcomes' of the algorithm
- therefore number of leaf nodes equals the possible orderings of n items
- that is there are least n! leaf nodes (remember permutations from AF2)

Sorting – Comparison based sorting

We have shown the decision tree has at least $n!$ leaf nodes

The worst-case complexity of the algorithm is no better than $O(h)$

- where h is the height of the tree
- an execution is a path from the root node to a leaf node
- we perform an operation an each branch node so h operations in the worst case

A decision tree is a binary tree (two branches 'yes' and 'no') and hence the number of leaf nodes is less than or equal to $2^{h+1}-1$

- a binary tree of height h has at most $2^{h+1}-1$ nodes

Combining these properties it follows that $n! \leq 2^{h+1}-1 \leq 2^{h+1}$

Sorting – Comparison based sorting

We have shown: complexity is no better than **$O(h)$** and **$2^{h+1} \geq n!$**

- h is the height of the decision tree
- n is the number of items to be sorted

Taking **$\log_2$** of both sides of **$2^{h+1} \geq n!$** yields:

$$
\begin{aligned}
h+1 &\geq \log_2(n!) & & \\
&> \log_2(n/2)^{n/2} & & \text{(since } n! > (n/2)^{n/2}\text{)} \\
&= (n/2)\log_2(n/2) & & \text{(since } \log a^b = b \log a\text{)} \\
&= (n/2)\log_2 n - (n/2)\log_2 2 & & \text{(since } \log a/b = \log a - \log b\text{)} \\
&= (n/2)\log_2 n - n/2 & & \text{(since } \log_a a = 1\text{)}
\end{aligned}
$$

Giving a complexity of at least **$O(n \log n)$** as required

Sorting – Radix sorting

We haven shown no sorting algorithm that is based on pairwise comparisons can be better than `O(n log n)` in the worst case

- therefore to improve on this worst case bound, we have to devise a method based on something other than comparisons

Radix sort uses a different approach to achieve an `O(n)` complexity

- but the algorithm has to exploit the structure of the items being sorted, so may be less versatile
- in practice, it is faster than `O(n log n)` algorithms only for very large `n`

Assume items to sort can be treated as bit-sequences of length `m`

- let `b` be a chosen factor of `m`
- so `b` and `m` are constants for any particular instance

Sorting – Radix sorting – Algorithm

Each item has bit positions labelled `0,1,…,m-1`

- bit 0 being the least significant (i.e. the right-most)

The algorithm uses `m/b` iterations

- in each iteration the items are distributed into 2^b buckets
- a bucket is just a list
- the buckets are labelled $0,1,\ldots,2^b-1$ (or, equivalently, $\underbrace{00...0}_{\text{length } b}$ to $\underbrace{11...1}_{\text{length } b}$)
- during the i^{th} iteration an item is placed in the bucket corresponding to the integer represented by the bits in positions $b\times i-1,\ldots,b\times(i-1)$
  - e.g. for $b=4$ and $i=2$

item = 0010100100110001

Sorting – Radix sorting – Algorithm

Each item has bit positions labelled `0,1,…,m-1`

- bit 0 being the least significant (i.e. the right-most)

The algorithm uses `m/b` iterations

- in each iteration the items are distributed into 2^b buckets
- a bucket is just a list
- the buckets are labelled $0,1,\ldots,2^b-1$ (or, equivalently, $\underbrace{00\ldots0}_{\text{length } b}$ to $\underbrace{11\ldots1}_{\text{length } b}$)
- during the i^{th} iteration an item is placed in the bucket corresponding to the integer represented by the bits in positions $b\times i-1,\ldots,b\times(i-1)$
  - e.g. for $b=4$ and $i=2$, consider bits in position $7,\ldots,4$

    `item = 0010100100110001`

  - 0011 represents the integer 3
  - so item is placed in the bucket labelled 3 (or, equivalently, 0011)
- at the end of an iteration the buckets are concatenated to give a new sequence which will be used as the starting point of the next iteration

Sorting – Radix sorting – Example

Suppose we want to sort the following sequence with Radix sort

15	43	5	27	60	18	26	2

Binary encodings are given by

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

- items have bit positions 0,…,5, hence $m=6$
- b must be a factor of m, so lets choose $b=2$

This means in Radix sort we have:

- $2^b=2^2=4$ buckets labelled 0,1,2,3 (or equivalently 00,01,10,11) and $m/b = 3$ iterations are required

Sorting – Radix sorting – Example

Sequence:

15	43	5	27	60	18	26	2

Binary encodings:

15 = 001111	43 = 101011	5 = 000101	27 = 011011
60 = 111100	18 = 010010	26 = 011010	2 = 000010

First iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 1st iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 1,…,0
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

```
1st iteration:
bucket 00: 60
bucket 01:  5
bucket 10: 18 26  2
bucket 11: 15 43 27
new sequence: 60 5 18 26 2 15 43 27
```

Sorting – Radix sorting – Example

New sequence:

60	5	18	26	2	15	43	27

Binary encodings:

60 = 111100	5 = 000101	18 = 010010	26 = 011010
2 = 000010	15 = 001111	43 = 101011	27 = 011011

Second iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 2nd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 3,…,2
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

```
2nd iteration:
bucket 00: 18  2
bucket 01:  5
bucket 10: 26 43 27
bucket 11: 60 15
new sequence: 18 2 5 26 43 27 60 15
```

Sorting – Radix sorting – Example

New sequence:

```
18  2  5  26  43  27  60  15
```

Binary encodings:

```
18 = 010010    2 = 000010    5 = 000101   26 = 011010
43 = 101011   27 = 011011   60 = 111100   15 = 001111
```

Third (and final) iteration of radix

- items are distributed into 4 buckets (a bucket is just a list)
- during the 3rd iteration, an item is placed in a bucket corresponding to the integer represented by the bits in positions 5,…,4
- buckets concatenated at the end of an iteration to give input sequence for the next iteration

```
3rd iteration:
bucket 00:  2  5 15
bucket 01: 18 26 27
bucket 10: 43
bucket 11: 60
sorted sequence: 2 5 15 18 26 27 43 60
```

Sorting – Radix sorting – Pseudocode

```
// assume we have the following method which returns the value
// represented by the b bits of x when starting at position pos
private int bits(Item x, int b, int pos)

// suppose that:
//     a is the sequence to be sorted
//     m is the number of bits in each item of the sequence a
//     b is the 'block length' of radix sort

int numIterations = m/b; // number of iterations required for sorting
int numBuckets = (int) Math.pow(2, b); // number of buckets

// represent sequence a to be sorted as an ArrayList of Items
ArrayList<Item> a = new ArrayList<Item>();

// represent the buckets as an array of ArrayLists
ArrayList<Item>[] buckets = new ArrayList[numBuckets];
for (int i=0; i<numBuckets; i++) buckets[i] = new ArrayList<Item>();
```

Sorting – Radix sorting – Pseudocode

```
for (int i=1; i<=numIterations; i++){

   // clear the buckets
   for (int j=0; j<numBuckets; j++) buckets[j].clear();

   // distribute the items (in order from the sequence a)
   for (Item x : a){
     // find the value of the b bits starting from position (i-1)*b in x
     int k = bits(x, b, (i-1)*b); // find the correct bucket for item x
     buckets[k].add(x); // add item to this bucket
   }

   a.clear(); // clear the sequence

   // concatenate the buckets (in sequence) to form the new sequence
   for (j=0; j<numBuckets; j++) a.addAll(buckets[j]);

 }
```

Sorting – Radix sorting – Correctness

Let x and y be two items with x<y

- need to show that x precedes y in the final sequence

Suppose j is the last iteration for which relevant bits of x and y differ

- since x<y and j is the last iteration that x and y differ

  the relevant bits of x must be smaller than those of y
- therefore x goes into an ‘earlier’ bucket than y

  and hence x precedes y in the sequence after this iteration

- since j is the last iteration where bits differ:
  in all later iterations x and y go in the same bucket
  so their relative order is unchanged

Sorting – Radix sorting – Complexity

Number of iterations is m/b and number of buckets is 2^b

During each of the m/b iterations

- the sequence is scanned and items are allocated buckets: $O(n)$ time
- buckets are concatenated: $O(2^b)$ time

Therefore the overall complexity is $O(m/b \cdot (n+2^b))$

- this is $O(n)$, since m and b are constants

Time-space trade-off

- the larger the value of b, the smaller the multiplicative constant (m/b) in the complexity function and so the faster the algorithm will become
- however an array of size 2^b is required for the buckets
  therefore increasing b will increase the space requirements

Tries (retrieval)

Binary search trees are comparison-based data structures

Tries are to binary trees as Radixsort is to comparison-based sorting

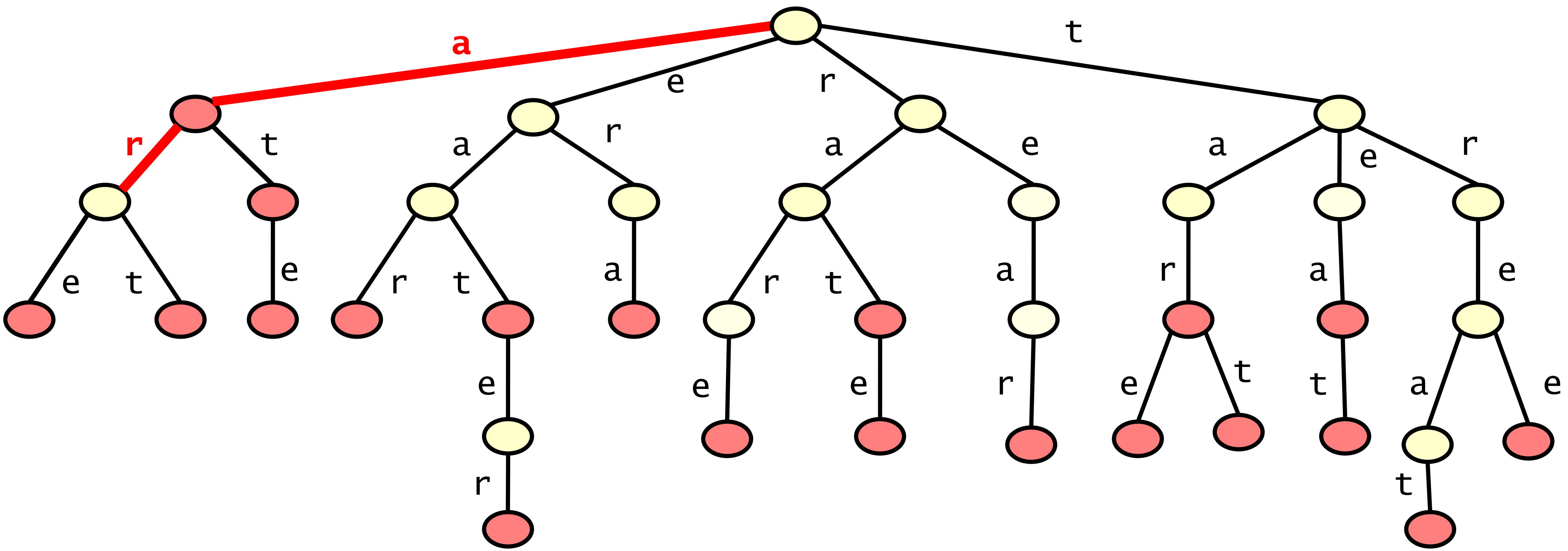
- stored items have a key value that is interpreted as a sequence of bits, or characters, …
- there is a multiway branch at each node where each branch has an associated symbol and no two siblings have the same symbol
- the branch taken at level i during a search, is determined by the i^{th} element of the key value (i^{th} bit, i^{th} character, …)
- tracing a path from the root to a node spells out the key value of the item

Example: use a trie to store items with a key value that is a string

- say the words in a dictionary

Tries – Examples

An example trie containing words from a 4 letter alphabet

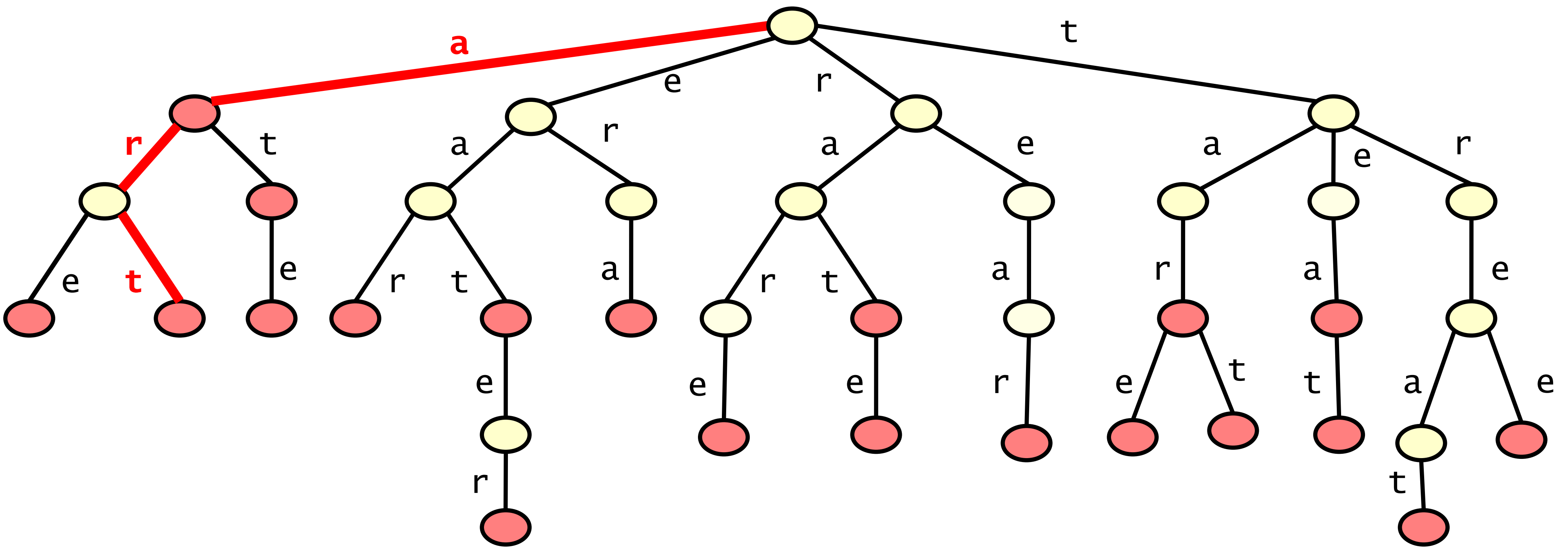


- Two kinds of nodes
  - nodes representing words
  - internal/intermediate nodes

path represents the string: **ar**
(not a word)

Tries – Examples

An example trie containing words from a 4 letter alphabet



- Two kinds of nodes
  - nodes representing words
  - internal/intermediate nodes

path represents the word: **art**

Tries – Search algorithm (pseudo code)

```
// searching for a word w in a trie t
Node n = root of t; // current node (start at root)
int i = 0; // current position in word w (start at beginning)

while (true) {
  if (n has a child c labelled w.charAt(i)) {
    // can match the character of word in the current position
    if (i == w.length()-1) { // end of word
      if (c is an 'intermediate' node) return "absent";
      else return "present";
    }
    else { // not at end of word
      n = c; // move to child node
      i++; // move to next character of word
    }
  }
  else return "absent"; // cannot match current character
}
```

Tries – Insertion algorithm (pseudo code)

```
// inserting a word w in a trie t
Node n = root of t; // current node (start at root)

for (int i=0; i < w.length(); i++){ // go through chars of word
  if (n has no child c labelled w.charAt(i)){
    // need to add new node
    create such a child c;
    mark c as intermediate;
  }
  n = c; // move to child node
}
mark n as representing a word;
```

Tries – Algorithms

Deletion of a string from a trie

- exercise

Complexity of trie operations

- (almost) independent of the number of items
- essentially linear in the string length

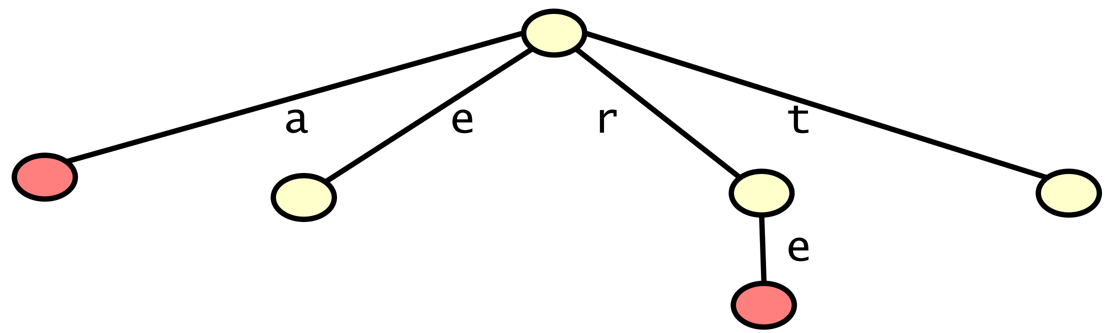
Tries – Implementation

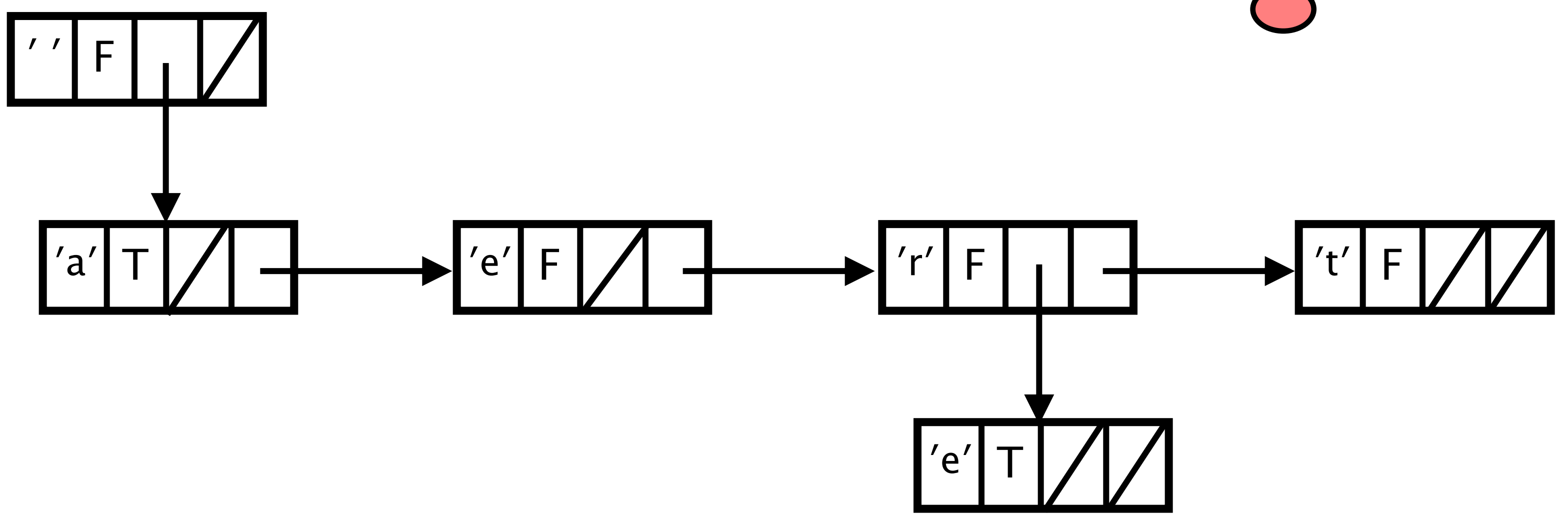
Various possible implementations

- using an array (of pointers to represent the children of each node)
- using a linked lists (to represent the children of each node)
- time/space trade-off

List implementation

- trie
- becomes the list





Tries – Class to represent dictionary tries

```
public class Node { // node of a trie
  private char letter; // label on incoming branch
  private boolean isWord; // true when node represents a word
  private Node sibling; // next sibling (when it exists)
  private Node child; // first child (when it exists)

  /** create a new node with letter c */
  public Node(char c){
    letter = c;
    isWord = false;
    sibling = null;
    child = null;
  }
// include accessors and mutators for the various components of class
}
public class Trie {
  private Node root;
  public Trie() {
    root = new Node(Character.MIN_VALUE); // null character in root
  }
```

Tries – Method to search

```
private enum Outcomes {PRESENT, ABSENT, UNKNOWN}
/** search trie for word w */
public boolean search(String w) {
  Outcomes outcome = Outcomes.UNKNOWN;
  int i = 0; // position in word so far searched (start at beginning)
  Node current = root.getChild(); // start with first child of root
  while (outcome == Outcomes.UNKNOWN) {
    if (current == null) outcome = Outcomes.ABSENT; // dead-end
    else if (current.getLetter() == w.charAt(i)) { // positions match
      if (i == w.length()-1) outcome = Outcomes.PRESENT; // matched word
      else { // descend one level…
         current = current.getChild(); // in trie
         i++; // in word being searched
      }
    }
    else current = current.getSibling(); // try next sibling
  }
  if (outcome != Outcomes.PRESENT) return false;
  else return current.getIsWord(); // true if current node represents a word
}
```

Tries – Method to insert

```
public void insert(String w){  /* insert word w into trie */
  int i = 0; // position in word (start at beginning)
  Node current = root; // current node of trie (start at root)
  Node next = current.getChild(); // child of current node we are testing
  while (i < w.length()) { // not reached the end of the word
    if (next.getLetter() == w.charAt(i)) { // chars match: descend a level
      current = next; // update current to the child node
      next = current.getChild(); // update child node
      i++; // next position in word
    } else if (next != null) next = next.getSibling(); // try next child
    else { // no more siblings: need new node
      Node x = new Node(s.charAt(i)); // label with ith element of word
      x.setSibling(current.getChild()); // sibling: first child of current
      current.setChild(x); // make it first child of current node
      current = x; // move to the new node
      next = current.getChild(); // update child node
      i++; // next position in word
    }
  }
  current.setIsWord(true); // current represents word w
}
```