Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Before we discuss an alternative to the multiway trie, it may be a good idea to think of a multiway trie node as an abstraction
- Think of what each node is and what it needs to do without considering the implementation details
- Let's look again at our multiway trie and think about its functionality on a node by node basis
  - We will consider the symbol table version here, since that is the version that Sedgwick implements in TrieST.java (and we modified in TrieSTNew.java for Assignment 1)

Abstracting a Multiway Trie Node

‘a’	‘b’		…	‘y’	‘z’	val
/		/	/		/	/

- Data (in one node)
  - Collection of references to the children of the node
    - A null value indicates no child corresponding to that character
  - Reference to the value that is stored for a key if the key is present in the symbol table
    - A null value indicates that the key is not present but rather only a prefix of something stored in the symbol table
- Methods (in one node)
  - We want to be able to get the reference associated with a character – go to a child
  - We want to be able to set the reference associated with a character – assign a child

- We want to be able to set the value of a Node
- We want to be able to get the value of a Node

- To express this in an abstract way we can use an interface in Java
  - The idea here is that our interface will define the functionality of a trie node
  - We can then implement this interface in several ways, based on our needs / goals
  - We can then define our trie based on this abstracted node
- See TrieNodeInt.java
  - See code and read comments
  - Note: This is not a standard Java interface

Abstracting a Multiway Trie Node

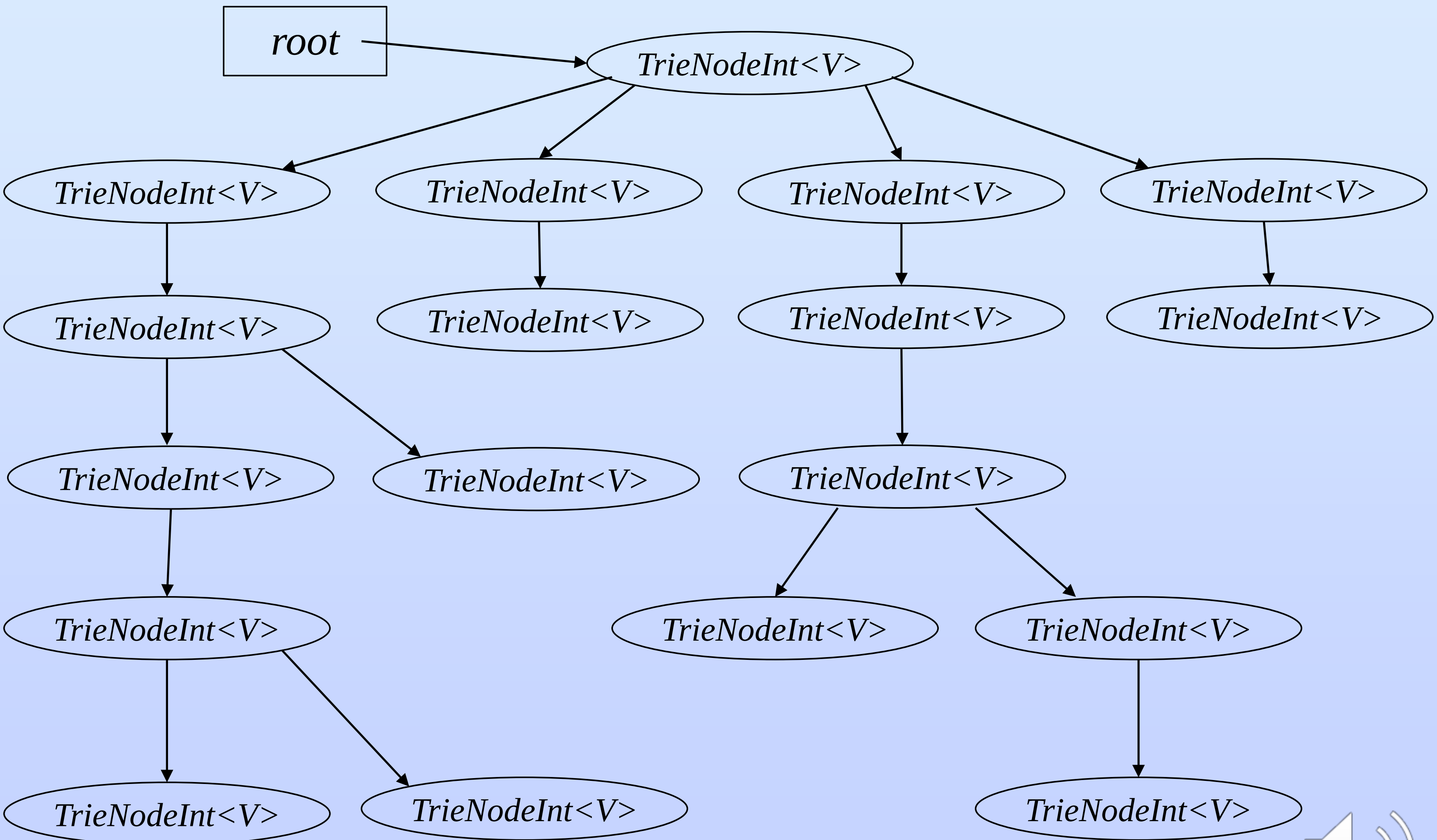
```
public interface TrieNodeInt<V>
{
// Return next node in trie corresponding to char c in current
// node, or null if there is no next node for that char.
       public TrieNodeInt<V> getNextNode(char c);

// Set next node in the trie corresponding to char c to arg
// node.  If the ref. at that pos. was prev. null, incr. degree
// of this node by one (since it now has one more branch).
       public void setNextNode(char c, TrieNodeInt<V> node);

// Return data at the curr. node (or null if there is no data)
       public V getData();

// Set the data at the current node to the data argument
       public void setData(V data);

// Return the degree of the current node.  This corresponds to
// the number of children that this node has.
       public int getDegree();
}
```

Abstracting a Multiway Trie Node

- Note that (not counting getDegree()) all of the ops are expressed in terms of three data types:
  - char c
    - This determines how to branch from the current node
    - Each different char value will determine a diff. branch
  - V data
    - This is the **value** stored for a given key in the symbol table
    - We can set it or get it
  - TrieNodeInt<V>
    - This is an abstraction of a node in our trie
    - Represented by the interface type
    - TrieNodeInt<V> is **self-referential**

- Nowhere in the interface does it say how we must implement these operations
- We could implement them using an array-based node
  - This is what we already have seen
- We could implement them in a different (perhaps completely different) way
- If our trie is made up of TrieNodeInt<V> nodes, to the trie it does not matter
  - As long as they function correctly

Abstracting a Multiway Trie Node



Multiway Trie Node with an Array

- Now let's look at our multiway trie impl. again, using the TrieNodeInt<V> interface
  - Trie idea is the same as before, but now using the abstract TrieNodeInt<V> for each node in the trie
    - We will use this instead of our Node from the original implementation
    - Recall that in Java we can use an interface variable to store any object that implements that interface
    - Thus a TrieNodeInt<V> reference can access any class that implements TrieNodeInt<V>
  - Initially in our TrieNodeInt<V> references we will store array based implementations for each node
    - We will call this class MTNode<T>
      - For Multiway Trie node

Multiway Trie Node with an Array

- Our overall trie class will be TrieSTMT<V>
  - Within this class our trie will be built using MTNode<V> objects
  - But they will be accessed using TrieNodeInt<V> references
- The TrieSTMT (MT stands for "Multiway Trie") class does not actually care how the TrieNodeInt<V> is implemented -- that is abstracted out
- See:
  - MTNode.java (implementation of TrieNodeInt)
  - TrieSTMT.java (trie using MTNode)
    - Compare to TrieSTNew.java
  - DictTestForInterface.java (program to test this)
  - We will look over these handouts during our synchronous lecture

Multiway Trie Node with an Array

- Let's look at *part* of MTNode.java

```
public class MTNode<V> implements TrieNodeInt<V>
{
    private static final int R = 256;
    protected V val;
    protected TrieNodeInt<V> [] next;
    protected int degree;

    public MTNode()
    {
        val = null;
        degree = 0;
        next = (TrieNodeInt<V> []) new TrieNodeInt<?>[R];
    }

    public TrieNodeInt<V> getNextNode(char c)
    {
        return next[c];
    }
    // see handout for rest of code
```

Multiway Trie Node with An Array

- TrieSTNew (partial)

```
public class TrieSTNew<Value>
{
  private static final int R = 256;
  private Node root;
  // ...
  public int searchPrefix(String key)
  {
    int ans = 0;
    Node curr = root;
    boolean done = false;
    int loc = 0;
    while (curr != null && !done)
    {
      if (loc == key.length())
      {
        if (curr.val != null)
            ans += 2;
        if (curr.degree > 0)
            ans += 1;
        done = true;
      }
      else
      {
        curr = curr.next[key.charAt(loc)];
        loc++;
      }
    }
    return ans;
  }
```

- TrieSTMT (partial)

```
public class TrieSTMT<V>
{
  private TrieNodeInt<V> root;
  // ...
  public int searchPrefix(String key)
  {
    int ans = 0;
    TrieNodeInt<V> curr = root;
    boolean done = false;
    int loc = 0;
    while (curr != null && !done)
    {
      if (loc == key.length())
      {
        if (curr.getData() != null)
            ans += 2;
        if (curr.getDegree() > 0)
            ans += 1;
        done = true;
      }
      else
      {
        curr =
           curr.getNextNode(key.charAt(loc));
        loc++;
      }
    }
    return ans;
  }
```

de la Briandais Trees

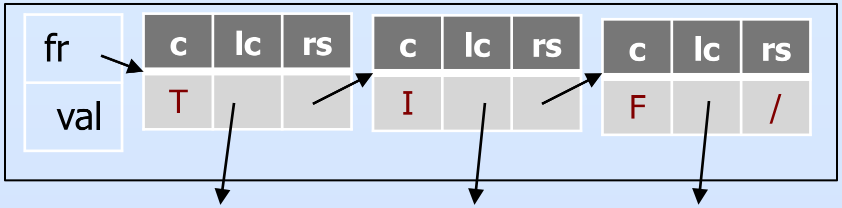
- Now we can look at an alt. implementation of TrieNodeInt<V> that could possibly save memory
- Consider a "node" from a multiway trie to actually be a linked-list of "nodelets" in a dlB
  - Each nodelet points to one **existing** child node
    - Any pointers that are not used are not included in the list
    - For example, let's say our trie had only three words:
      - > THIS, IS, FUN
    - Let's see how the first node in our trie would look
    - First let's remember how our array-based node would look:

…	'F'	…	'I'	…	'T'	…	val
/		/					/

- How many pointers do we need here?

de la Briandais Trees

- Now consider a de la Briandais Node for the same first node



- Clearly it is more complicated than the array version
  - But note how many pointers are needed
  - We are not allocating all poss. pointers
    - > Rather we are only allocating what we need

- dlB nodelets are uniform with two references each
  - One for sibling and one for a single child

de la Briandais nodelet

character (bit pattern)	ref to child node (next level)	ref to sibling node (same level)

de la Briandais Trees

- Now our multiway trie node will contain
  - A reference to the front of our linked list of nodelets
    - Each nodelet will correspond to a child of the node
    - If a child does not exist, the nodelet does not exist
    - Compare to the array of children we used previously
  - A reference to the value for that string (to store the value associated with a key)
- The functionality will be the same, since it will still implement the TrieNodeInt<V> interface
- Note that now we need to store the characters rather than use them as indexes
    - One node in the multiway trie is now represented by several “nodelets” in the DLB

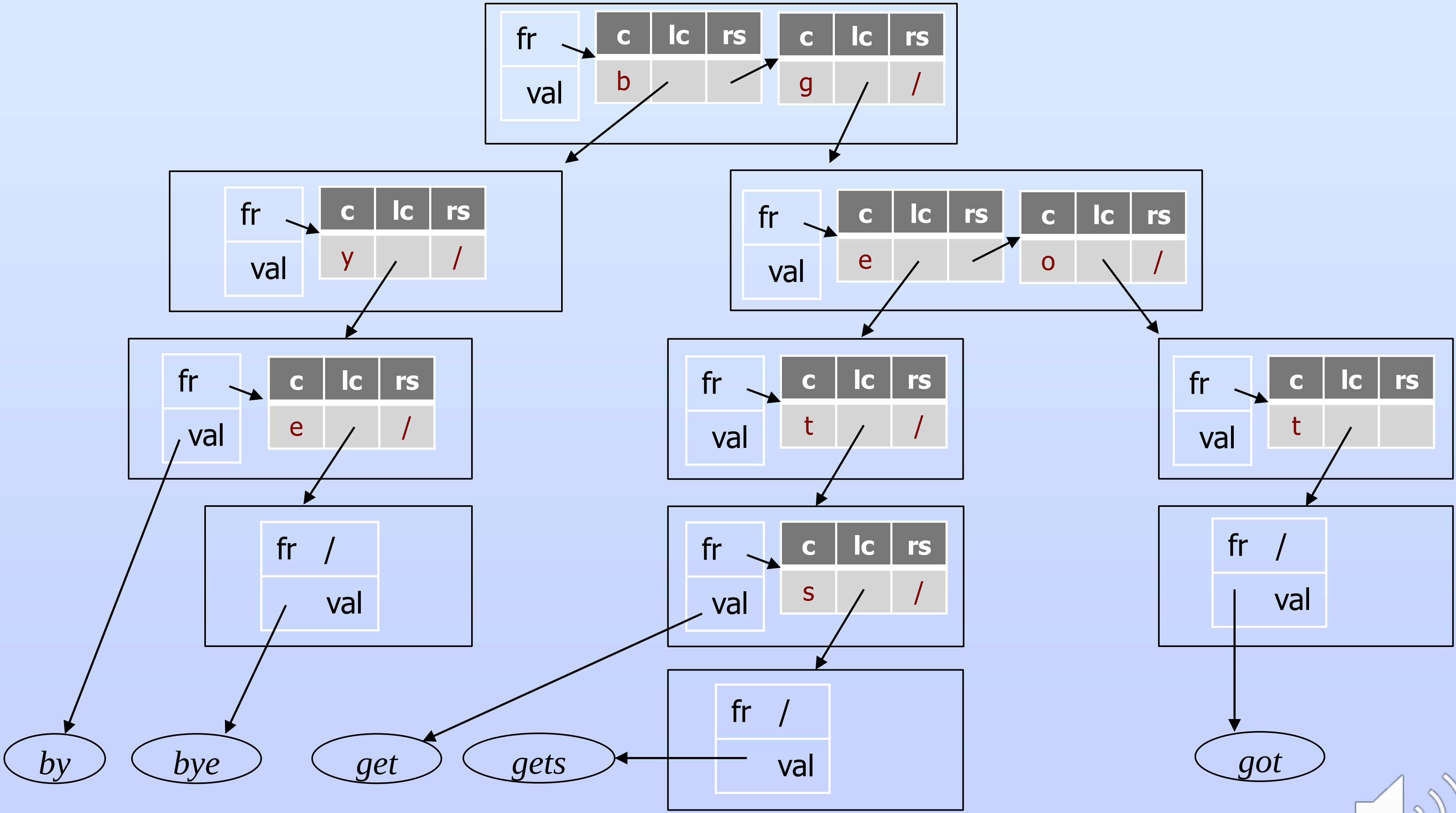
de la Briandais Trees

- Each character match causes us to follow a child pointer to the next level
- Each mismatch causes us to follow a sibling pointer
    - On the same level
    - If we get to NULL the key is not found
- So now, finding a child node corresponding to a character requires a sequential search of the list rather than a single direct access of an array
    - We will discuss the cost / benefit of this soon

- Let's look again at the example we used before for our TrieST, but now using a DLB
    - We will assume a symbol table, and we will assume that the keys and values are both the same entity (a String)
    - Compare the next slide to slide 25 of Lecture 3.

de la Briandais Tree

- Consider again bye, by, get, got and gets



de la Briandais Trees

- Run-time?
  - Assume we have S valid characters (or bit patterns) possible in our "alphabet"
    - Ex. 256 for ASCII
  - Assume our key contains K characters
  - In worst case we can have up to **Θ(KS)** character comparisons required for a search
    - Up to S comparisons to find the character in each node
    - K levels to get to the end of the key
  - How likely is this worst case?
    - Remember the reason for using dlB is that most of the levels will have very few characters
    - So practically speaking a dlB search will require **Θ(K)** time

de la Briandais Trees

- Implementing dlBs?
  - Note that our main TrieST class would have no changes, other than one line:
    - Original TrieSTMT.java class put() method:

```
private TrieNodeInt<V> put(TrieNodeInt<V> x, String key, V val, int d)
{
     if (x == null) x = new MTNode<V>();
     // rest omitted
```

    - New TrieSTDLB.java class put() method:

```
private TrieNodeInt<V> put(TrieNodeInt<V> x, String key, V val, int d)
{
     if (x == null) x = new DLBNode<V>();
     // rest omitted
```

    - Everything else in the class is based on the interface, and the details are abstracted out of our view

de la Briandais Trees

- Within the DLBNode<V> class we would need to implement the TrieNodeInt<V> interface
  - getNextNode() and setNextNode() will now require iteration through the nodelets within that node
    - Some special cases as any linked list implementation requires
    - If we want to keep the data sorted, we will need to insert nodelets into the correct alpha location within the list
      - Can't just put a new nodelet at the end
    - Degree value must be updated correctly as it was in MTNode
  - getData() and setData() are just as simple here as with the MTNode class
  - getDegree() simply returns the degree value for the node
- This is really cool!

- So how would these compare?
  - Run-time:
    - Clearly the array implementation cannot be beaten
    - MTNode is Theta(1) to get the child of a node, regardless of the number of children
    - DLB requires iterating through the list
      - The more children, the longer the access
  - Memory:
    - This depends on the number of children of a node
    - Consider memory for one child:
      - Array is just a reference (ex: 4 bytes)
      - DLB is char + sib ref + child ref (ex: 10 bytes)
    - So per child DLB uses more memory but…

  - Advantage of DLB is that we don't have the nodelets unless we need them
    - So for few children, DLB is clearly superior
    - For most / all children, MTNode will be superior
- It would be nice to allow for a hybrid trie
  - Starts out all nodes with DLBNodes
  - Once a node gets past a certain number of children (its degree) it is converted into an MTNode
  - Since both nodes implement TrieNodeInt the overall Trie class would be virtually the same
    - Would just need a test in the put() method to see if the degree of the node is past the threshold – if so convert to MTNode
- Maybe you should implement this?!