Lecture 12: Tries

CSE 373: Data Structures and Algorithms

Announcements

Slido Event #3322291
https://app.sli.do/event/1LRNfJyVj9LATb179RWGjL

Practice Midterms Posted

https://courses.cs.washington.edu/courses/cse373/23sp/#04-28

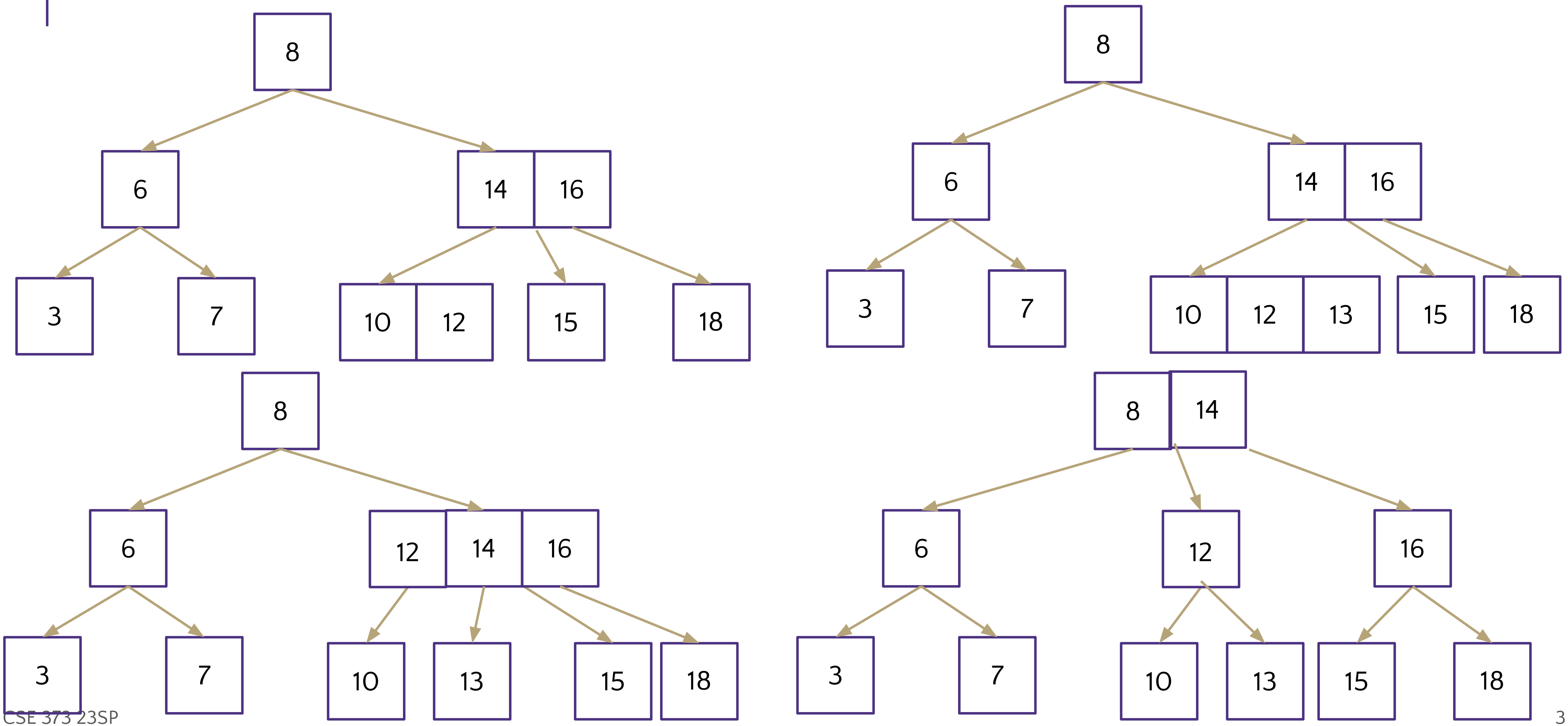
Project 2 Due Wednesday

Exercise 3 Due Monday

Exercise 4 Releases Monday

2-3 Insertions

Insert 12 and 13 into the following 2-3 tree



2-3 Trees

PROS

- All operations on 2-3 Tree have a logarithmic worst case
  - Because these trees are always balanced!
- Maintaining balance doesn't require complex rotations
- Storing multiple values per node improves runtime constants because of memory locality

CONS

- No height triggered balancing means 2-3 trees stay a little less balanced than AVLs on average
- Multiple node types cause implementation complexity
  - Make all nodes 2 nodes and you have more unused space

2-3 `insert()` code

```
class Node {
    int[] keys;
    Node[] children;
    int numKeys;
    boolean isLeaf;
...
}
```

```
public void insertNonFull(int key) {
   int i = numKeys - 1;
   if (isLeaf) {
      while (i >= 0 && keys[i] > key) {
         keys[i + 1] = keys[i];
         i--;
      }
      keys[i + 1] = key;
      numKeys++;
   } else {
      while (i >= 0 && keys[i] > key) {
         i--;
      }
      if (children[i + 1].numKeys == 2 * order - 1) {
         splitChild(i + 1, children[i + 1]);
         if (keys[i + 1] < key) {
            i++;
         }
      }
      children[i + 1].insertNonFull(key);
   }
}
```

```
public void splitChild(int i, Node y) {
   Node z = new Node(y.order, y.isLeaf);
   z.numKeys = order - 1;
   for (int j = 0; j < order - 1; j++) {
      z.keys[j] = y.keys[j + order];
   }
   if (!y.isLeaf) {
      for (int j = 0; j < order; j++) {
         z.children[j] = y.children[j + order];
      }
   }
   y.numKeys = order - 1;
   for (int j = numKeys; j >= i + 1; j--) {
      children[j + 1] = children[j];
   }
   children[i + 1] = z;
   for (int j = numKeys - 1; j >= i; j--) {
      keys[j + 1] = keys[j];
   }
   keys[i] = y.keys[order - 1];
   numKeys++;
}
```

2–3 Trees

PROS

- All operations on 2–3 Tree have a logarithmic worst case
  - Because these trees are always balanced!
- Maintaining balance doesn't require complex rotations
- Storing multiple values per node improves runtime constants because of memory locality

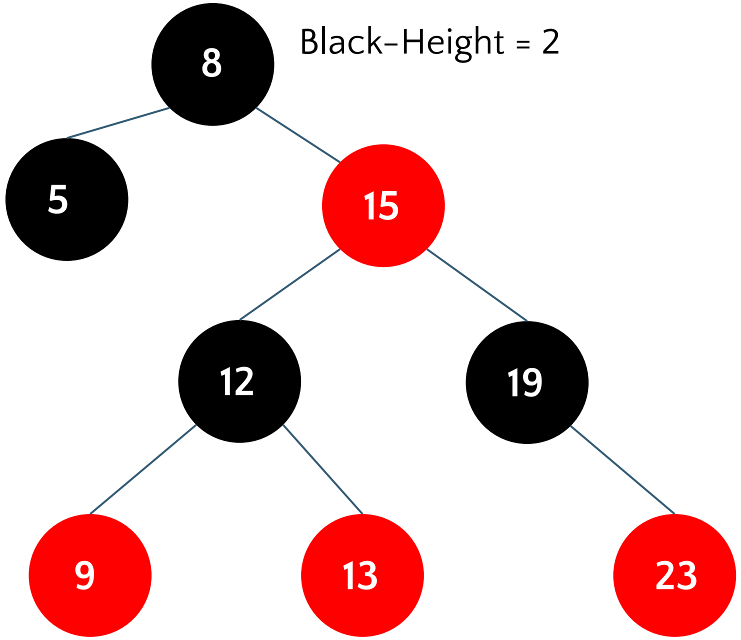
CONS

- No height triggered balancing means 2–3 trees stay a little less balanced than AVLs on average
- Multiple node types cause implementation complexity
  - Make all nodes 2 nodes and you have more unused space

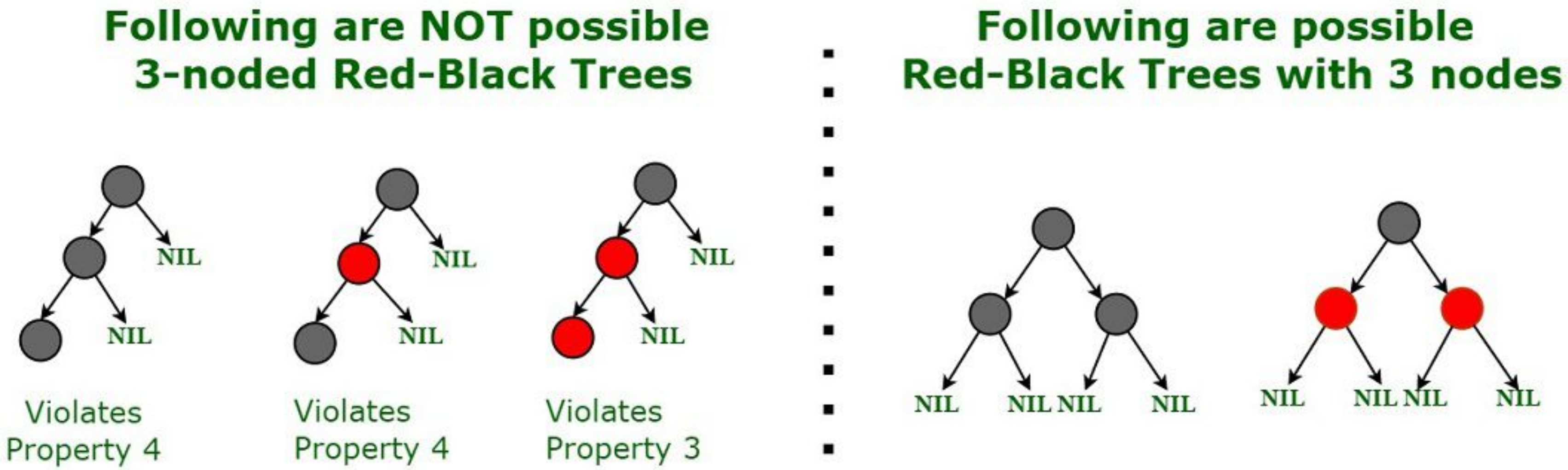
Meet Red Black Trees

All paths from a node to the NULL descendants contain the same number of black nodes

Black-Height = 2



1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. Every leaf (e.i. NULL node) must be colored BLACK.



Red Black Insertions

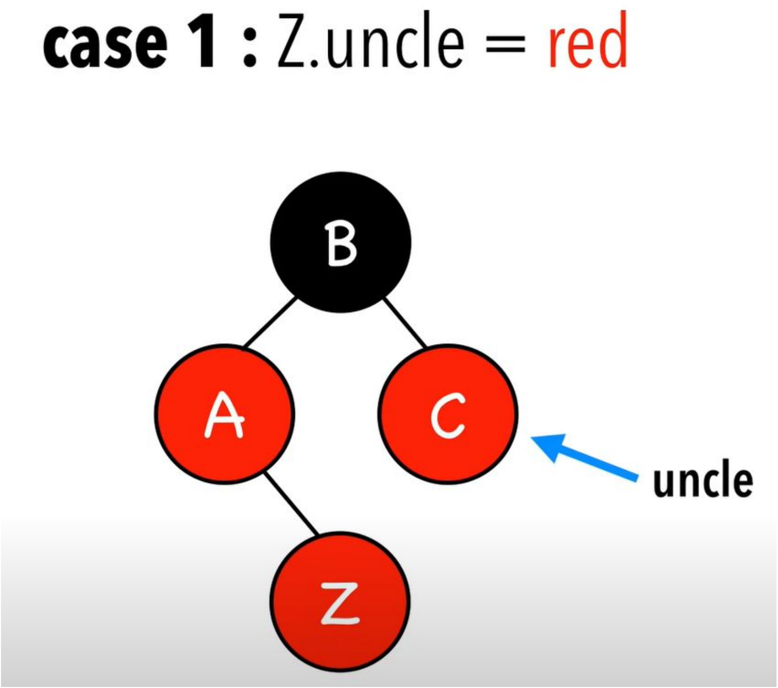
Insertion cases:

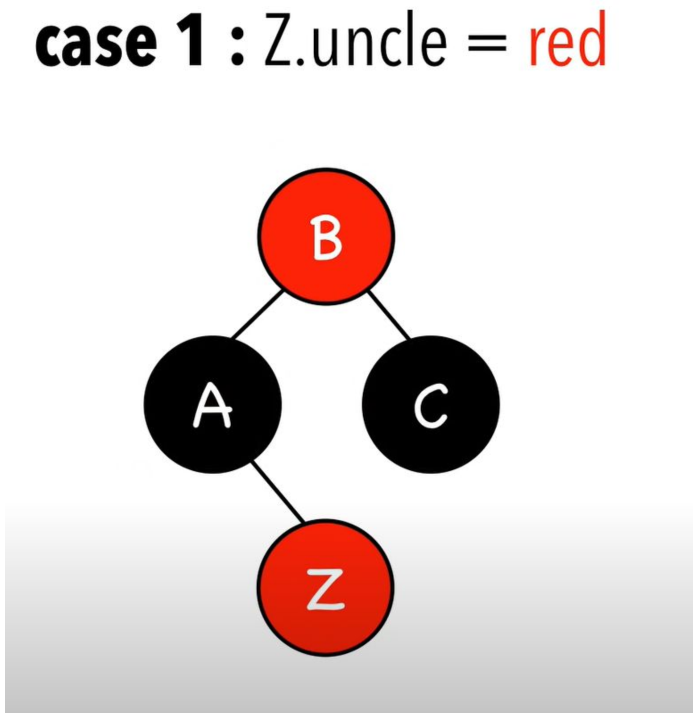
0. Node is the root
    a. Color node black
1. Node's uncle is red
    b. recolor
2. Node's uncle is black (Triangle)
    c. Rotate node's parent
3. Node's uncle is black (line)
    d. Rotate nodes' grandparent & recolor

[Red Black Tree Insertions (Video 5min)]

Node's uncle is red

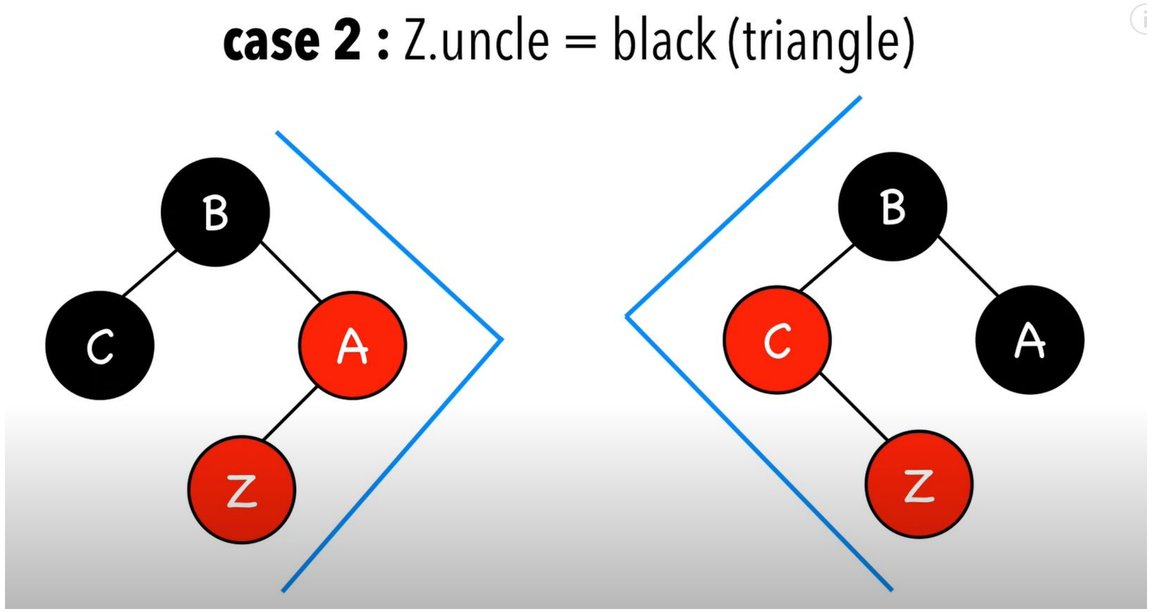
Recolor parent, uncle and grandparent

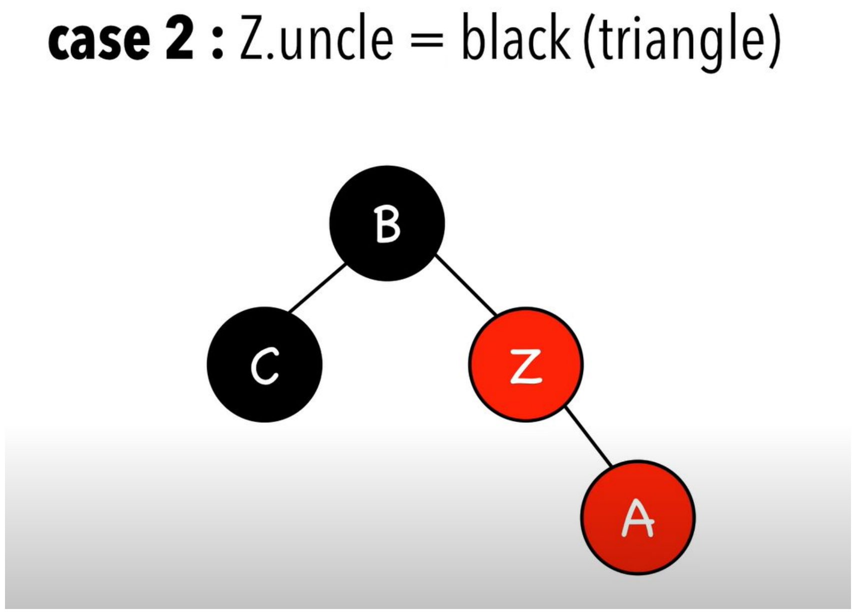




Uncle is black (triangle)

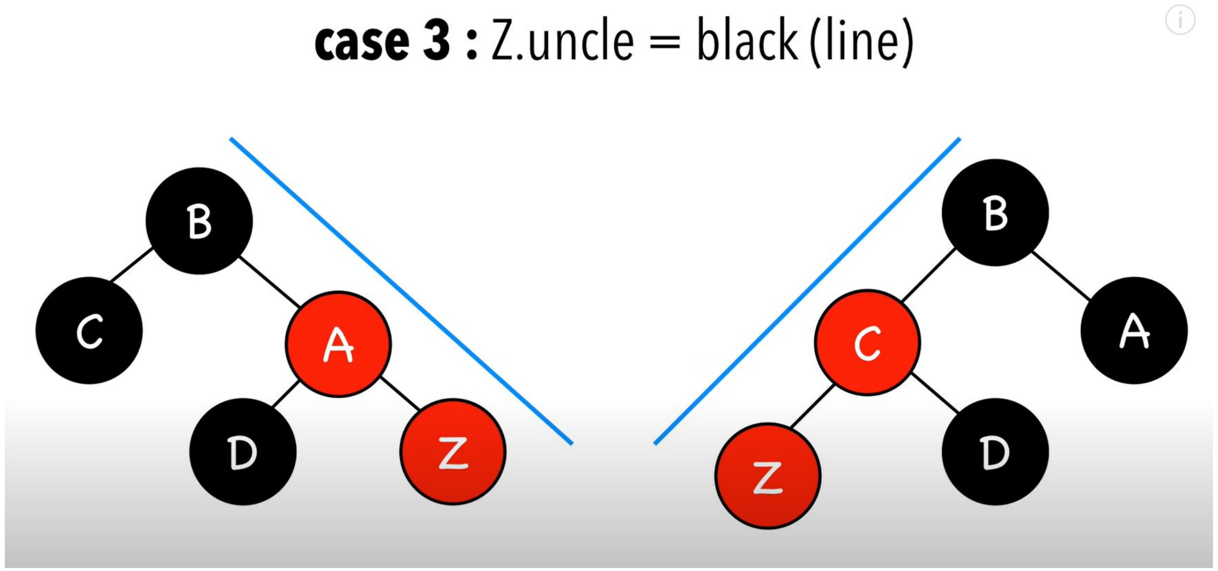
Rotate inserted Nodes parent in opposite direction of inserted node

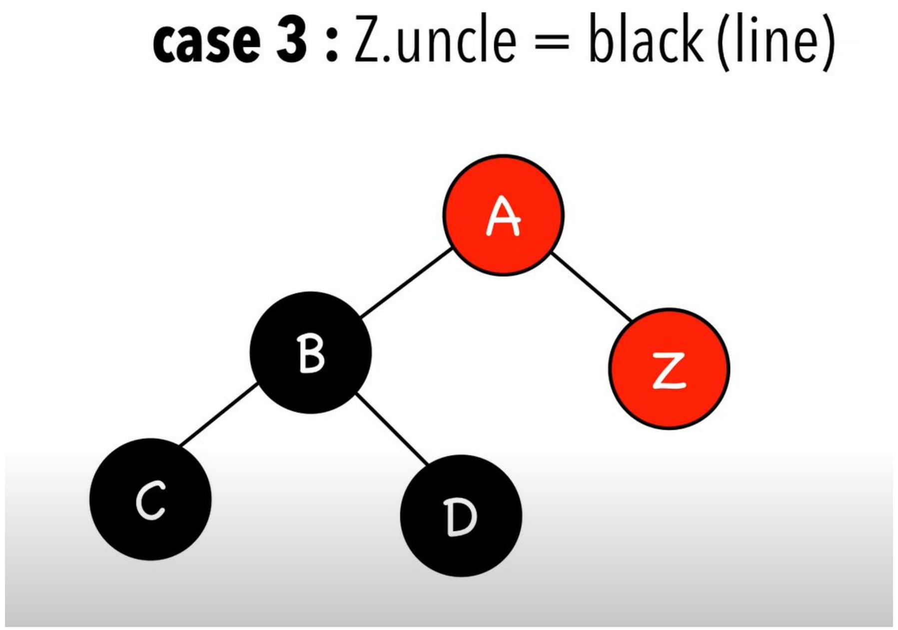


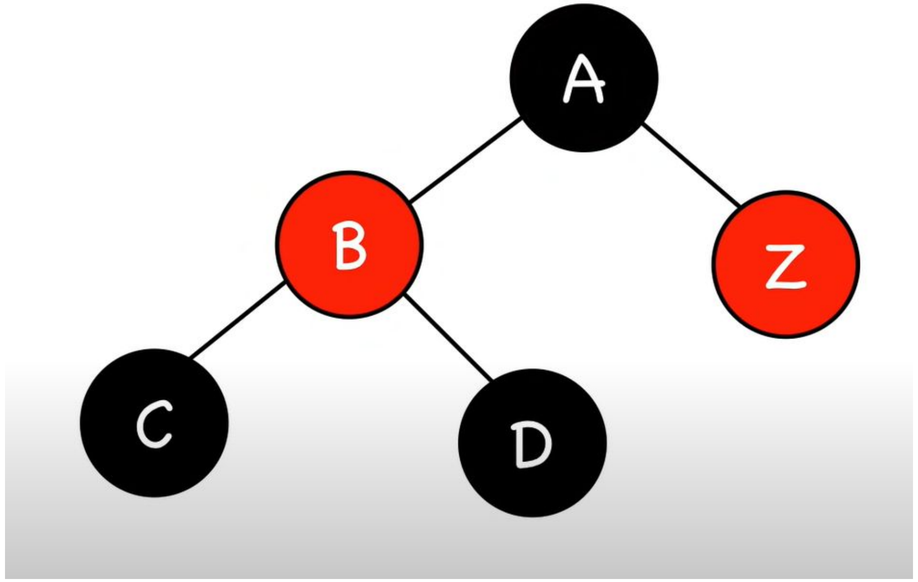


Uncle is black (line)

Rotate node's grandparent, then recolor







AVL vs Red Black Trees

Red Black Trees:

- Easier to implement than AVL
  - Left Leaning Red Black trees are even **easier** to implement
- Better performance for insertion and deletion because the balancing mechanism is less strict than AVL

AVL Trees:

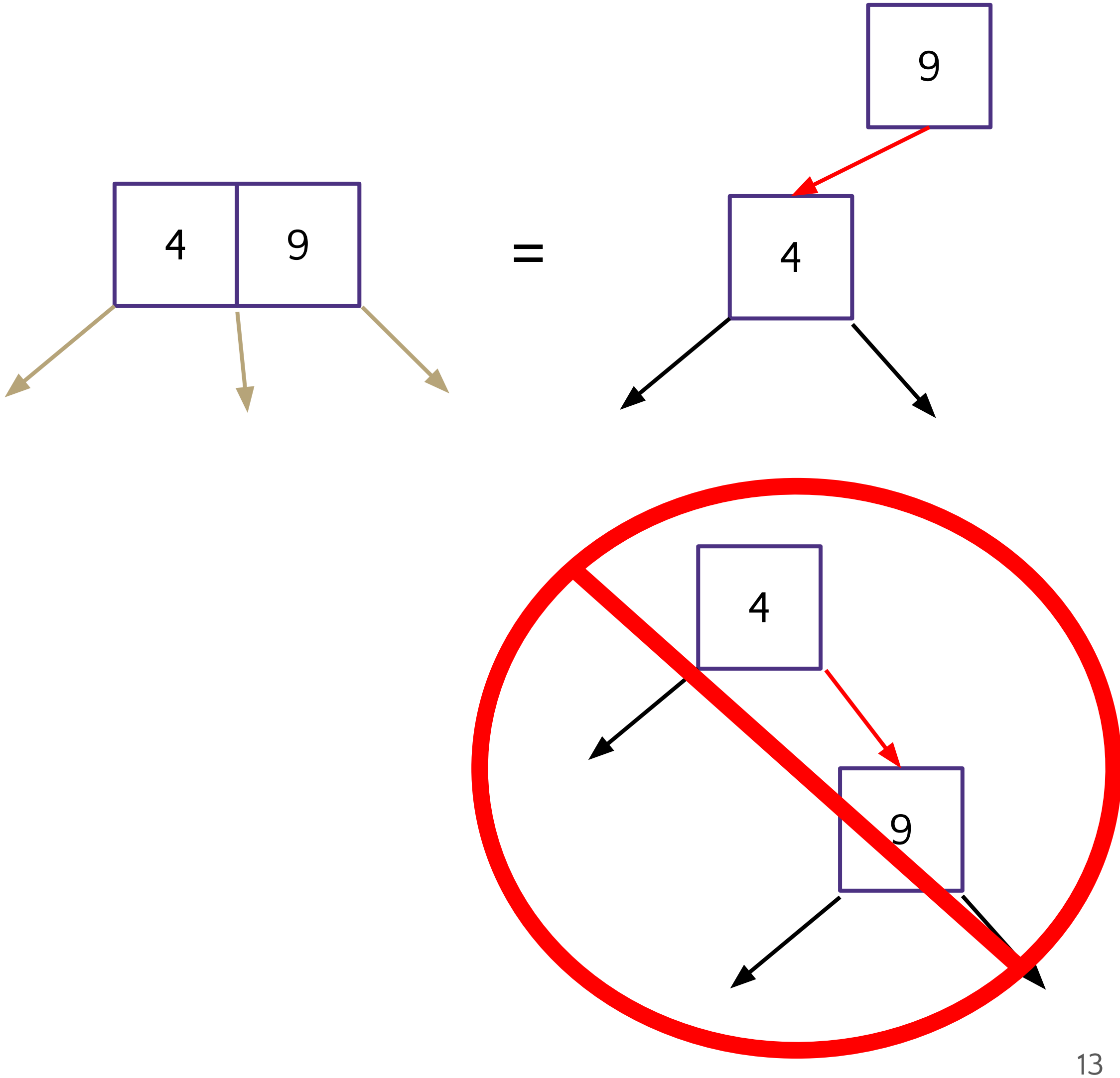
- Have better look up performance because of their strict balance requirements

Left Leaning Red Black Trees

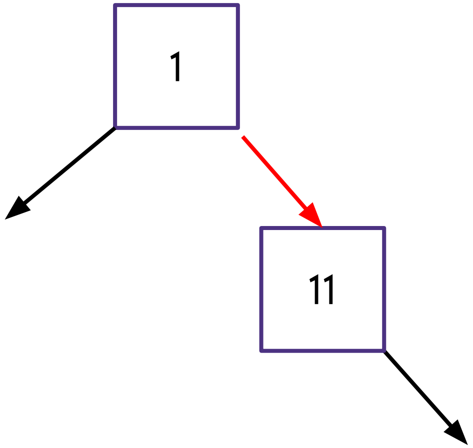
A translation of 2 3 trees using nodes with only 1 value

- Red links connect two nodes that would exist within the same node in a 2-3 tree
- Black links are "standard" connections
- Red links are always on the left
- A "balanced" LLRB has the same number of black links to leaf
  - Red links don't count towards path length

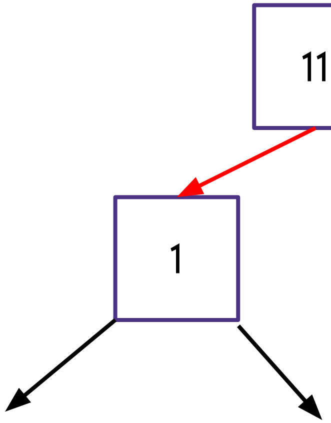
A proposed improvement to the Red Black tree from its original designer Robert Sedgewick

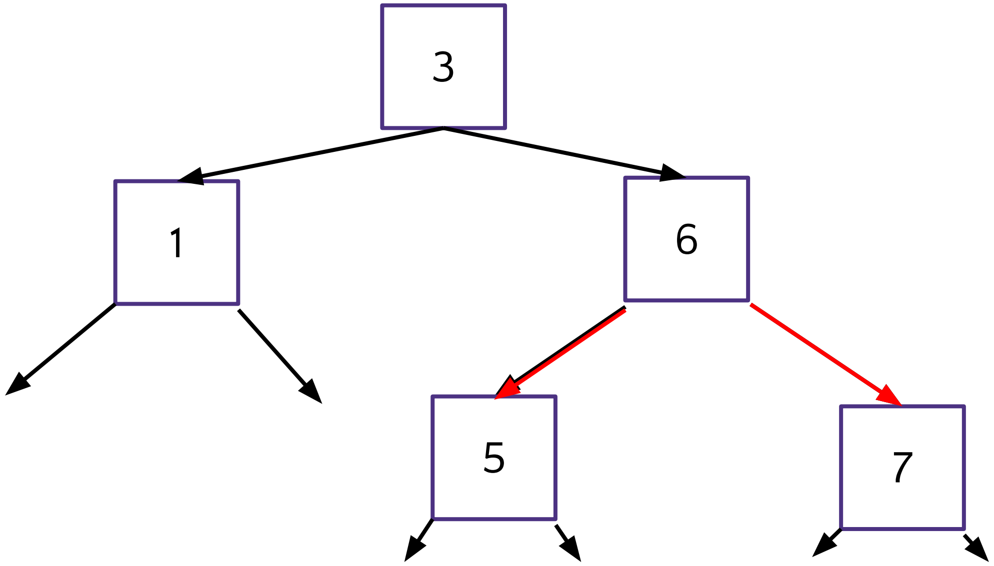


Valid Left Leaning Red Black Tree?

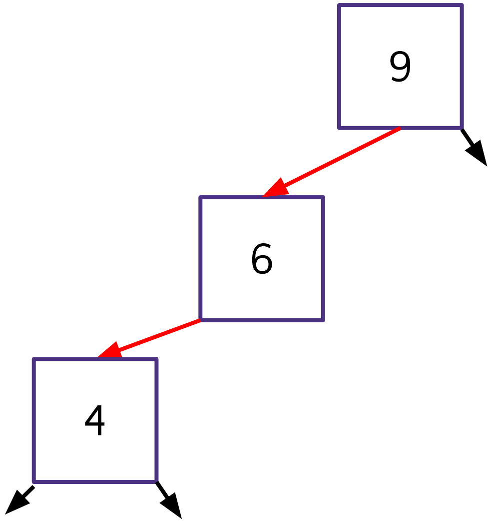


Right red link





Different length paths



Sequential Red nodes

LLRB `insert()` code

Regular BST code
LLRB additions

```
public class LLRB<Key extends Comparable<Key>, Value> {
   private static final boolean RED = true;
   private static final boolean BLACK = false;
   private Node root;
   private class Node {
      private Key key;
      private Value val;
      private Node left, right;
      private boolean color;
      Node(Key key, Value val) {
         this.key = key;
         this.val = val;
         this.color = RED;
      }
 }

 public Value search(Key key) {
    Node x = root;
    while (x != null) {
       int cmp = key.compareTo(x.key);
       if (cmp == 0) return x.val;
       else if (cmp < 0) x = x.left;
       else if (cmp > 0) x = x.right;
    }
    return null;
 }
```

```
public void insert(Key key, Value value) {
   root = insert(root, key, value);
   root.color = BLACK;
}

private Node insert(Node h, Key key, Value value) {
   if (h == null) return new Node(key, value);
   if (isRed(h.left) && isRed(h.right)) colorFlip(h);
   int cmp = key.compareTo(h.key);
   if (cmp == 0) h.val = value;
   else if (cmp < 0) h.left = insert(h.left, key, value);
   else h.right = insert(h.right, key, value);
   if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
   if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
   return h;
}
```

Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Trie Introduction

Implementation

Prefix Matching

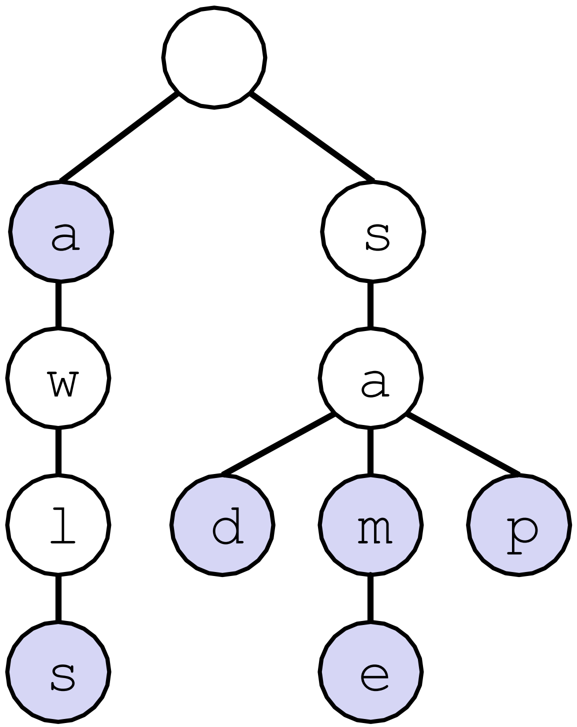
Interview Question Prep

The Trie: A Specialized Data Structure

- Tries view its keys as:
  - a **sequence of characters**
  - some (hopefully many!) sequences share common prefixes

- `sap`
- `sad`
- `awls`
- `a`
- `same`
- `sam`

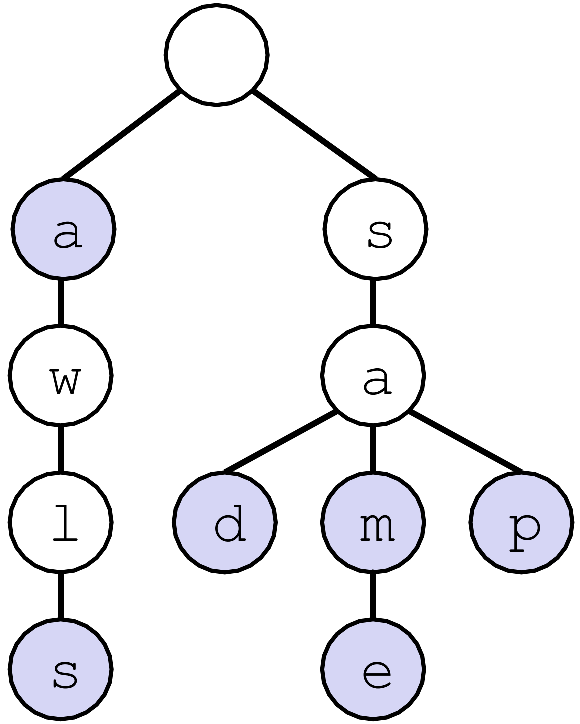
Set ADT



Trie

Trie: An Introduction

- Each level of the tree represents an index in the string
  - Children at that level represent possible
  - characters at that index
- This abstract trie stores the set of strings:
  - `awls, a, sad, same, sap, sam`
- How to deal with `a` and `awls`?
  - Mark which nodes *complete* a string (shown in purple)

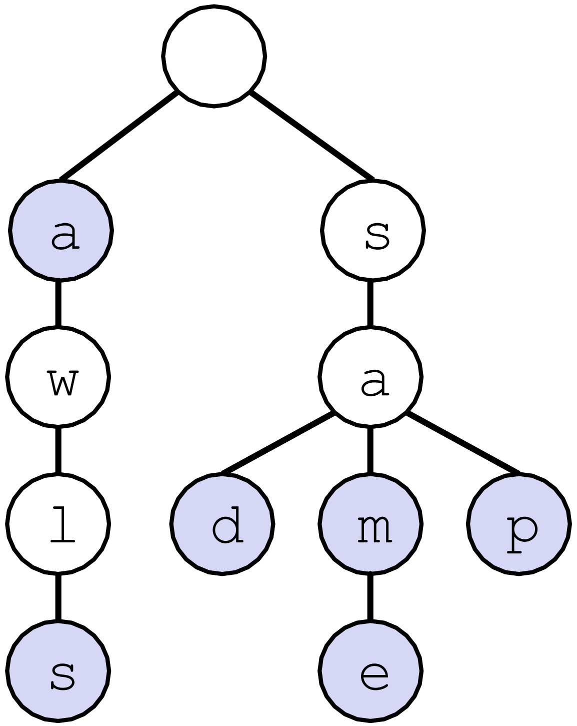


Searching in Tries

Two ways to fail a contains() check:

1. If we fall off the tree
2. If the final node isn't purple (not a key)

Input String	Fall Off? / Is Key?	Result
`contains("sam")`	hit / purple	True
`contains("sa")`	hit / white	False
`contains("a")`	hit / purple	True
`contains("saq")`	fell off / n/a	False



Keys as “a sequence of characters” (1 of 2)

- Most dictionaries treat their keys as an “atomic blob”: you can’t disassemble the key into smaller components
- Tries take the opposite view: keys are a **sequence of characters**
  - `Strings` are made of `Characters`
- But “characters” don’t have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg, 蛋糕)
  - `List<E>`
  - `byte[]`

Keys as “a sequence of characters” (2 of 2)

- But “characters” don’t have to come from the Latin alphabet
  - `Character` includes most Unicode codepoints (eg 蛋糕)
  - `List<E>`
  - `byte[]`
- Tries are defined by 3 types instead of 2:
  - An “alphabet”: the domain of the characters
  - A “key”: a sequence of “characters” from the alphabet
  - A “value”: the usual Dictionary value

Trie Introduction

Implementation

Prefix Matching

Interview Question Prep

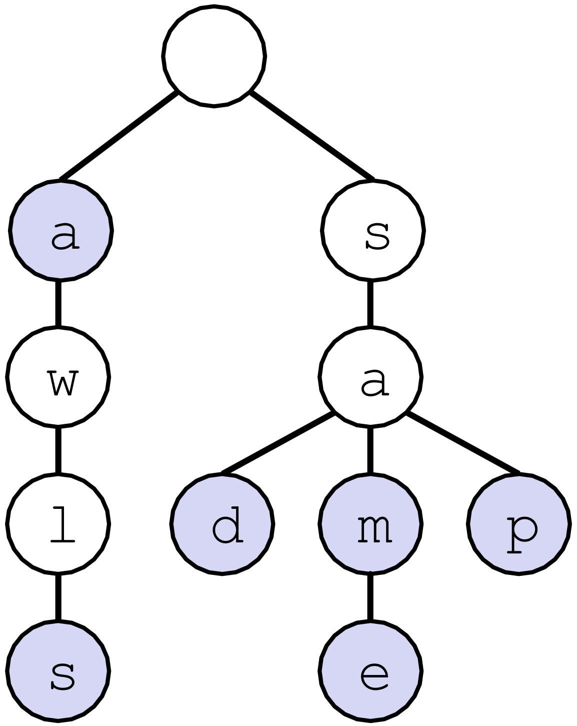
ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

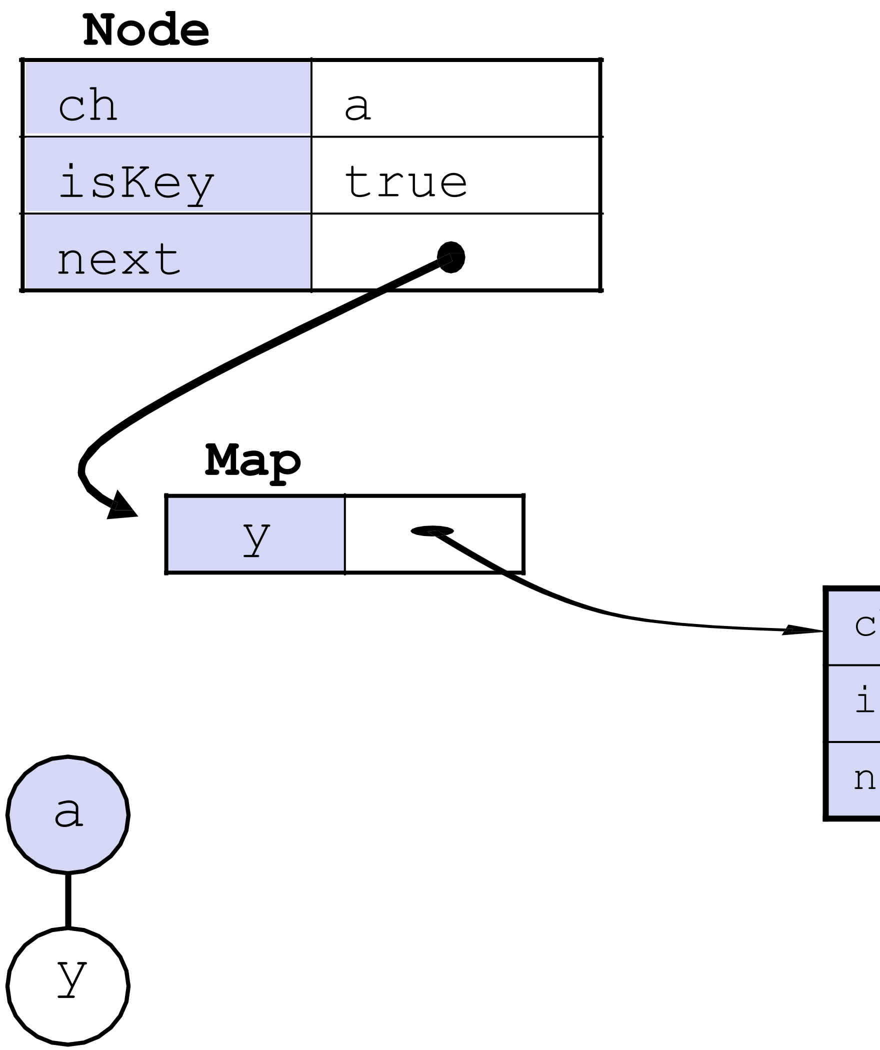
Simple Trie Implementation*

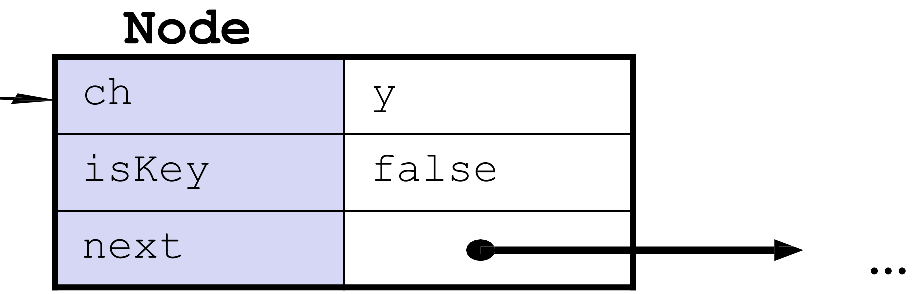
```
public class TrieSet {
   private Node root;

   private static class Node {
      private char ch;
      private boolean isKey;
      private Map<char, Node> next;
      private Node(char c, boolean b) {
         ch = c;
         isKey = b;
         next = new HashMap();
      }
   }
}
```



Simple Trie Node Implementation



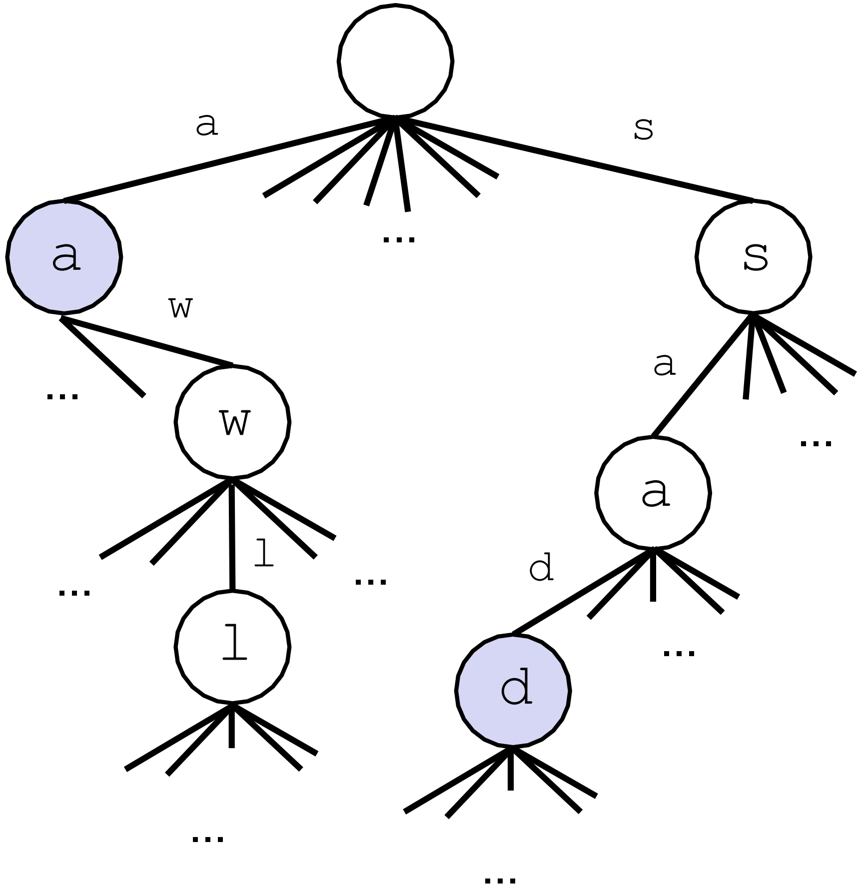



```
private static class Node {
  private char ch;
  private boolean isKey;
  private Map<char, Node> next;
   ...
}
```

Simple Trie Implementation

```
  public class TrieSet {
    private Node root;

   private static class Node {
     private char ch;
     private boolean isKey;  private
     Map<char, Node> next;
     private Node(char c, boolean b) {
       ch = c;
       isKey = b;
       next = new HashMap();
     }
   }
}
```



Trie Introduction

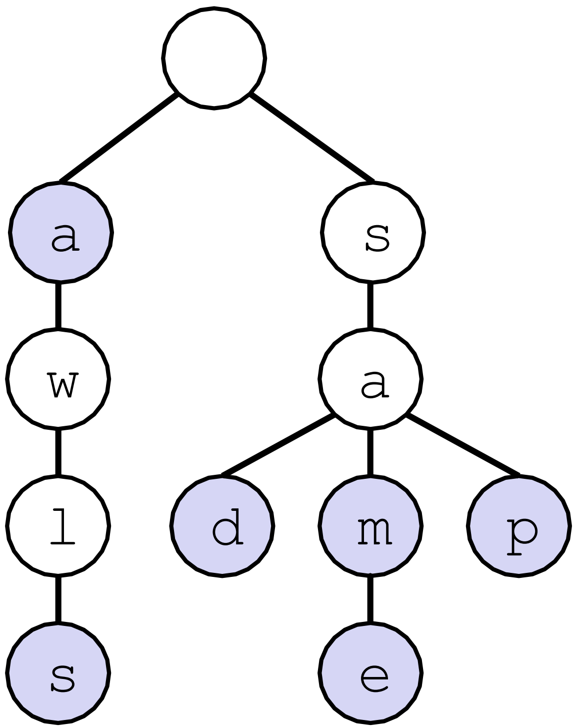
Implementation

Prefix Matching

Interview Question Prep

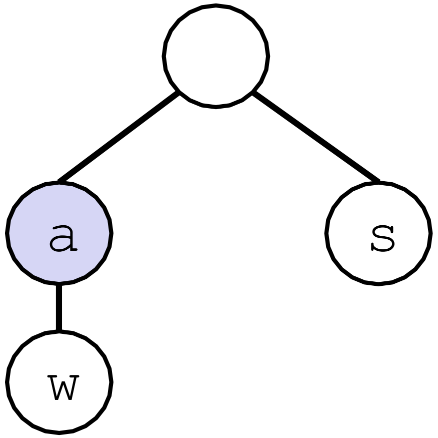
Trie-Specific Operations

- The main appeal of tries is prefix matching!
  - Why? Because they view their keys as sequences that can have prefixes
- **Longest prefix**
  - `longestPrefixOf("sample")`
  - Want: `{"sam"}`
- **Prefix match**
  - `findPrefix("sa")`
  - Want: `{"sad", "sam", "same", "sap"}`



Related Problem: Collecting Trie Keys

- Imagine an algorithm that collects *all* the keys in a trie:
  - `collect():` `["a","awls","sad","sam","same","sap"]`
- It could be implemented as follows:




```
\\Create an empty list of results x
\\For each character c in root.next.keys():
  \\call collectHelper(c, x, root.next.get(c))
\\return x
```

Summary

- A trie data structure implements the Dictionary and Set ADTs
- Tries have many different implementations
  - Could store HashMap/TreeMap/any-dictionary within nodes
  - Much more exotic variants change the trie's representation, such as the Ternary Search Trie
- Tries store sequential keys
  - … which enables very efficient prefix operations like `findPrefix`

Trie Introduction

Implementation

Prefix Matching

Interview Question Prep

Interview Prep

- Any time you see word/letter parsing!
  - fast run time with tries (quick word / letter lookup)
  - not just for interview, but real-world applications
- Interviewer's favorite "gimmick" question
  - came up for me
- Example Problem:
  Find first 'k' maximum occurring words in a given set of strings
  - see if you can do this problem on your own

Questions?

Your toolbox so far…

ADT

- List – flexibility, easy movement of elements within structure
- Stack – optimized for first in last out ordering
- Queue – optimized for first in first out ordering
- Dictionary (Map) – stores two pieces of data at each entry

<– It's all about data baby!

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

Data Structure Implementation

- Array – easy look up, hard to rearrange
- Linked Nodes – hard to look up, easy to rearrange
- Hash Table – constant time look up, no ordering of data
- BST – efficient look up, possibility of bad worst case
- AVL Tree – efficient look up, protects against bad worst case, hard to implement

Review: Dictionaries

Why are we so obsessed with Dictionaries?

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
`put(key,value)`	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
`get(key)`	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
`remove(key)`	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data

Things to consider:

- What functionality is needed?
  - What operations need to be supported?
  - Which operations should be prioritized?
- What type of data will you have?
  - What are the relationships within the data?
  - How much data will you have?
  - Will your data set grow?
  - Will your data set shrink?
- How do you think things will play out?
  - How likely are best cases?
  - How likely are worst cases?

Example: Class Gradebook

You have been asked to create a new system for organizing students in a course and their accompanying grades

What functionality is needed?

What operations need to be supported?

- Add students to course
- Add grade to student's record
- Update grade already in student's record
- Remove student from course
- Check if student is in course
- Find specific grade for student

Which operations should be prioritized?

What type of data will you have?

What are the relationships within the data?

- Organize students by name, keep grades in time order…

How much data will you have?

- A couple hundred students, < 20 grades per student

Will your data set grow? A lot at the beginning,

Will your data set shrink? Not much after that

How do you think things will play out?

How likely are best cases?

How likely are worst cases?

- Lots of add and drops?
- Lots of grade updates?
- Students with similar identifiers?

Example: Class Gradebook

What data should we use to identify students? (keys)

- Student IDs – unique to each student, no confusion (or collisions)
- Names – easy to use, support easy to produce sorted by name

How should we store each student's grades? (values)

- Array List – easy to access, keeps order of assignments
- Hash Table – super efficient access, no order maintained

Which data structure is the best fit to store students and their grades?

- Hash Table – student IDs as keys will make access very efficient
- AVL Tree - student names as keys will maintain alphabetical order

Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

Update number of plays for a song
Add a new song to an artist's collection
Add a new artist and their songs to the service
Find an artist's most popular song
Find service's most popular artist
more…

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

Artists need to be associated with their songs, songs need t be associated with their play counts
Play counts will get updated a lot
New songs will get added regularly

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Some artists and songs will need to be accessed a lot more than others
Artist and song names can be very similar

Practice: Music Storage

- How should we store songs and their play counts?
    - Hash Table – song titles as keys, play count as values, quick access for updates
    - Array List – song titles as keys, play counts as values, maintain order of addition to system

- How should we store artists with their associated songs?
    - Hash Table – artist as key,
        - Hash Table of their (songs, play counts) as values
        - AVL Tree of their songs as values
    - AVL Tree – artists as key, hash tables of songs and counts as values

Questions?

That's all!