Radix Sorting and Searching

Rasmus Pagh

Based on slides by Kevin Wayne, Princeton
Today

Radix sorting
• Binary quicksort
• LSD radix sort
• Analysis
• Answer to “What is the fastest way to sort a million 32-bit integers?”

Radix search
• Tries
• Multi-way tries
• Analysis
The String data type

**String data type.** Sequence of characters (immutable).

**Indexing.** Get the \( i^{th} \) character.

**Substring extraction.** Get a contiguous sequence of characters from a string.

**String concatenation.** Append one character to end of another string.

![Diagram showing string indexing and substring extraction](image)
The String data type: performance

String data type. Sequence of characters (immutable).
Underlying implementation. Immutable char[] array, offset, and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>guarantee</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Memory. $40 + 2N$ bytes for a virgin String of length $N$.

use byte[] or char[] instead of String to save space
The StringBuilder data type

**StringBuilder data type.** Sequence of characters (mutable).

**Underlying implementation.** Doubling `char[]` array ("ArrayList") and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>String guarantee</th>
<th>String extra space</th>
<th>StringBuilder guarantee</th>
<th>StringBuilder extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>charAt()</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>substring()</code></td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td><code>concat()</code></td>
<td>N</td>
<td>N</td>
<td>1 *</td>
<td>1 *</td>
</tr>
</tbody>
</table>

* amortized

**Remark.** StringBuffer data type is similar, but thread safe (and slower).
String vs. StringBuilder

Challenge. How to reverse a string?

A.

```java
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

B.

```java
public static String reverse(String s)
{
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```
String vs. StringBuilder

**Challenge.** How to reverse a string?

A.  
```java
public static String reverse(String s) {
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
    {
        rev += s.charAt(i);
    }
    return rev;
}
```

B.  
```java
public static String reverse(String s) {
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
    {
        rev.append(s.charAt(i));
    }
    return rev.toString();
}
```
Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>no</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39N \lg N$ *</td>
<td>$1.39N \lg N$</td>
<td>$c \lg N$</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>no</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

* probabilistic

Lower bound. $\sim N \lg N$ compares are required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?
A. Yes, if we don't depend on compares.
Binary quicksort

Recall quicksort (recursive pseudocode):
qsort(i..j)
  if i=j return
  m=partition(i..j)
  qsort(i..m)
  qsort(m+1..j)

Alternative partitioning (integers):
- Split according to most significant bit on first recursion level.
- In ith recursion level, split by ith bit.
This is sometimes called “binary quicksort”.

Problem session: Suppose we are sorting b-bit integers.
What is the time complexity of binary quicksort in terms of n and b?
Is this faster or slower than a normal ~ n log n sorting algorithm?
Key-indexed counting: assumptions about keys

**Assumption.** Keys are integers between 0 and \( R - 1 \).

**Implication.** Can use key as an array index.

**Applications.**
- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm.

**Remark.** Keys may have associated data \( \Rightarrow \) can't just count up number of keys of each value.

<table>
<thead>
<tr>
<th>input name</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>Garcia</td>
<td>4</td>
</tr>
<tr>
<td>Harris</td>
<td>1</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>Johnnson</td>
<td>4</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>Thomas</td>
<td>4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>White</td>
<td>2</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sorted result (by section)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harris 1</td>
</tr>
<tr>
<td>Martin 1</td>
</tr>
<tr>
<td>Moore 1</td>
</tr>
<tr>
<td>Anderson 2</td>
</tr>
<tr>
<td>Martinez 2</td>
</tr>
<tr>
<td>Miller 2</td>
</tr>
<tr>
<td>Robinson 2</td>
</tr>
<tr>
<td>White 2</td>
</tr>
<tr>
<td>Brown 3</td>
</tr>
<tr>
<td>Davis 3</td>
</tr>
<tr>
<td>Jackson 3</td>
</tr>
<tr>
<td>Jones 3</td>
</tr>
<tr>
<td>Taylor 3</td>
</tr>
<tr>
<td>Williams 3</td>
</tr>
<tr>
<td>Garcia 4</td>
</tr>
<tr>
<td>Johnson 4</td>
</tr>
<tr>
<td>Smith 4</td>
</tr>
<tr>
<td>Thompson 4</td>
</tr>
<tr>
<td>Thomas 4</td>
</tr>
<tr>
<td>Thompson 4</td>
</tr>
<tr>
<td>Wilson 4</td>
</tr>
</tbody>
</table>

*Typical candidate for key-indexed counting*

Input sorted result

keys are small integers
**Goal.** Sort an array $a[]$ of $N$ integers between $0$ and $R - 1$.

- Count frequencies of each letter using key as index.
- 
- 

```java
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Key-indexed counting

**Goal.** Sort an array \( a[] \) of \( N \) integers between 0 and \( R - 1 \).

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.

```java
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]
Key-indexed counting

**Goal.** Sort an array `a[]` of `N` integers between 0 and `R - 1`.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
- Copy back into original array.

```java
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Key-indexed counting: analysis

**Proposition.** Key-indexed counting uses $8N + 3R$ array accesses to sort $N$ records whose keys are integers between 0 and $R - 1$.

**Proposition.** Key-indexed counting uses extra space proportional to $N + R$.

**Stable?** Yes!

**In-place?** No.
› key-indexed counting
› LSD radix sort
› MSD radix sort
› 3-way radix quicksort
› suffix arrays
Least-significant-digit-first string sort

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).

```
 0  d a b
 1  a d d
 2  c a b
 3  f a d
 4  f e e
 5  b a d
 6  d a d
 7  b e e
 8  f e d
 9  b e d
10  e b b
11  a c e
```
Least-significant-digit-first string sort

**LSD string (radix) sort.**

- Consider characters from right to left.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).

![Sort key diagram]

Sort must be stable (arrows do not cross)
Least-significant-digit-first string sort

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).

Sort must be stable (arrows do not cross)
Least-significant-digit-first string sort

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).

```
<table>
<thead>
<tr>
<th>sort key</th>
<th>sort key</th>
<th>sort key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 dab</td>
<td>0 dab</td>
<td>0 acce</td>
</tr>
<tr>
<td>1 add</td>
<td>1 cab</td>
<td>1 dab</td>
</tr>
<tr>
<td>2 cab</td>
<td>2 ebb</td>
<td>2 bad</td>
</tr>
<tr>
<td>3 fad</td>
<td>3 aad</td>
<td>3 bed</td>
</tr>
<tr>
<td>4 fde</td>
<td>4 fad</td>
<td>4 bbe</td>
</tr>
<tr>
<td>5 bad</td>
<td>5 bab</td>
<td>5 cba</td>
</tr>
<tr>
<td>6 dad</td>
<td>6 dad</td>
<td>6 dab</td>
</tr>
<tr>
<td>7 bee</td>
<td>7 fed</td>
<td>7 add</td>
</tr>
<tr>
<td>8 fed</td>
<td>8 bed</td>
<td>8 fde</td>
</tr>
<tr>
<td>9 bed</td>
<td>9 fde</td>
<td>9 bed</td>
</tr>
<tr>
<td>10 ebe</td>
<td>10 fde</td>
<td>10 fde</td>
</tr>
<tr>
<td>11 ace</td>
<td>11 ace</td>
<td>11 fee</td>
</tr>
</tbody>
</table>
```

Sort must be stable (arrows do not cross)
Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.
public class LSD
{
    public static void sort(String[] a, int W)
    {
        int R = 256;
        int N = a.length;
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--)
        {
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;

            for (int r = 0; r < R; r++)
                count[r+1] += count[r];

            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];

            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
## LSD string sort: example

<table>
<thead>
<tr>
<th>Input</th>
<th>d = 6</th>
<th>d = 5</th>
<th>d = 4</th>
<th>d = 3</th>
<th>d = 2</th>
<th>d = 1</th>
<th>d = 0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4PGC938</td>
<td>2IYE230</td>
<td>3CIO720</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>2IYE230</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>3CIO720</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>1ICK750</td>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2IYE230</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>10HV845</td>
<td>3CIO720</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>4JZY524</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
</tr>
<tr>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2IYE230</td>
<td>3ATW723</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>3CIO720</td>
<td>10HV845</td>
<td>4PGC938</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>3ATW723</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>4PGC938</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>3ATW723</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>4PGC938</td>
<td>4JZY524</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
</tr>
</tbody>
</table>
Summary of the performance of sorting algorithms

Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>(N^2/2)</td>
<td>(N^2/4)</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>(N)</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>(1.39 \times N \lg N) (*)</td>
<td>(1.39 \times N \lg N)</td>
<td>(c \lg N)</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>(2 \times N \lg N)</td>
<td>(2 \times N \lg N)</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>(2 \times W \times N)</td>
<td>(2 \times W \times N)</td>
<td>(N + R)</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length \(W\) keys

Q. What if strings do not have same length? Something better for large \(W\)?
A. See further algorithms in book: MSD radix sort, 3-way string quicksort.
Problem. Sort 1 million 32-bit integers.

Ex. Google interview (or presidential interview).

Which sorting method to use?

• Insertion sort.
• Mergesort.
• Quicksort.
• Heapsort.
• LSD string sort.
String sorting challenge 2a

Problem. Sort 1 million 32-bit integers.

Ex. Google interview (or presidential interview).

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.
Case study: Sorting 32-bit integers

Can view 32-bit integers as:
- Strings of length \( W=1 \) over alphabet of size \( R=2^{32} \).
- Strings of length \( W=2 \) over alphabet of size \( R=2^{16} \).
- Strings of length \( W=4 \) over alphabet of size \( R=2^{8} \).
- ...  

Trade-off:
- Each LSD sort takes time \( N+R \).
- Last term negligible when \( N=10^6 \), and \( R=2^{16} \).
Hard drive
Latency: 5ms
(15 million cycles)

SSD
Latency: 50µs
(150,000 cycles)

Intel Core 2 Duo Processor
3.0GHz, E8400 Wolfdale

Core 1
Throughput: ~1 instruction per cycle. One cycle takes ~0.33 nanoseconds. The exact number of cycles depends on the instruction.

Core 2
Same as Core 1.

L1 Data Cache
32 KB
Latency: 1ns (3 cycles)

L1 Instruction Cache
32 KB

L1 Data Cache

L1 Instruction Cache

L2 Cache
6MB
Latency: 4.7ns (14 cycles)

Front Side Bus
1333MHz DDR3
Bandwidth: 10GB/s

PCI Express x16. 8GB/s (each way)

Intel X48 Northbridge chip

RAM Modules - 8GB
Latency: ~83 ns (~250 cycles)

http://duartes.org/gustavo/blog
Computer architecture in a nutshell

- Key concepts:
  - CPU, memory (RAM), words.
  - Machine code instructions: Computation, (conditional) jumps, indirect addressing.
  - Pipelining and branch prediction.
  - Superscalar execution, data dependencies.
  - Caching, latency, external memory (hard disk/SSD), memory hierarchy.
  - Parallelism: SIMD, multicore.
Performance factors

Facts about a typical modern computer:

1) Slow: conditional code (if-then, case, ...), especially if condition is “unpredictable”.
2) Slow: random access to RAM.
3) Fast: sequential RAM access.
4) Very fast: access to memory locations that were recently accessed.
5) Fast: Loops where the iterations are independent.
6) Very slow: random access to disk! (SSD still bad.)
Performance factors and sorting

• Pros and cons of quicksort:
  – Good use of cache (sequential access).
  – Inner loop of partition has unpredictable conditional code.

• Pros and cons of radix sort:
  – Can be implemented with little conditional code (counting version).
  – Random memory access in every step!

• Quicksort uses more instructions than LSD radix sort if strings are shorter than roughly \((\log n)^2\).
### Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 $N \log N$ *</td>
<td>1.39 $N \log N$</td>
<td>$c \log N$</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>2 $N \log N$</td>
<td>2 $N \log N$</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>2 $N W$</td>
<td>2 $N W$</td>
<td>$N + R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>2 $N W$</td>
<td>$N \log_R N$</td>
<td>$N + D R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>1.39 $W N \log N$ *</td>
<td>1.39 $N \log N$</td>
<td>$\log N + W$</td>
<td>no</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length W keys
‡ average-length W keys
Review: summary of the performance of symbol-table implementations

Frequency of operations.

<table>
<thead>
<tr>
<th>implementation</th>
<th>typical case</th>
<th>ordered operations</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>red-black BST</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
</tr>
<tr>
<td>hashing</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</td>
</tr>
</tbody>
</table>

† under uniform hashing assumption

**Observation:** Time for `compareTo()`, `equals()`, and `hashcode()` is proportional to string length $W$.

**Q.** Can we do better?

**A.** Yes, if we can avoid examining the entire key, as with string sorting.
String symbol table basic API

String symbol table. Symbol table specialized to string keys.

```java
public class StringST<Value>

StringST() create an empty symbol table

void put(String key, Value val) put key-value pair into the symbol table

Value get(String key) return value paired with given key

boolean contains(String key) is there a value paired with the given key?
```

Goals:
- Avoid error probability of hashing
- Be faster and/or more flexible than binary search trees.
Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters and values in nodes (not keys).
- Each node has $R$ children, one for each possible character.
- For now, we do not draw null links.

Ex. she sells sea shells by the
Follow links corresponding to each character in the key.

- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.
Search in a trie

Follow links corresponding to each character in the key.

- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

\[
\begin{align*}
\text{get("shell")} \\
\text{get("shore")}
\end{align*}
\]
Insertion into a trie

Follow links corresponding to each character in the key.
• Encounter a null link: create new node.
• Encounter the last character of the key: set value in that node.
Trie representation: Java implementation

**Node.** A value, plus references to $R$ nodes.

```java
private static class Node {
    private Object value;
    private Node[] next = new Node[R];
}
```

Characters are implicitly defined by link index.

Each node has an array of links and a value.

Keys are not explicitly stored.
**Node.** A value, plus references to $R$ nodes.

```java
private static class Node {
    private Object value;
    private Node[] next = new Node[R];
}
```

Trie representation (R = 26)

Each node has an array of links and a value.

Characters are implicitly defined by link index.
Trie performance

Search miss.
• Could have mismatch on first character.
• Typical case: examine only a few characters (sublinear).

Search hit. Need to examine all $L$ characters for equality.

Space. $R$ null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

Bottom line. Fast search hit and even faster search miss, but wastes space.

Problem session. One of the data structures you have seen in the course can be used to decrease the space usage. Which one, and how much space would be needed using it?
String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c lg² N</td>
<td>c lg² N</td>
</tr>
<tr>
<td>hashing</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>R-way trie</td>
<td>L</td>
<td>log₂ R N</td>
</tr>
</tbody>
</table>

**R-way trie.**

- Method of choice for small $R$.
- Too much memory for large $R$.

**Challenge.** Use less memory, e.g., 65,536-way trie for Unicode!
Ternary search tries

TST. [Bentley-Sedgewick, 1997]

• Store characters and values in nodes (not keys).
• Each node has three children: smaller (left), equal (middle), larger (right).
Ternary search tries

**TST.** [Bentley-Sedgewick, 1997]

- Store characters and values in nodes (not keys).
- Each node has **three** children: smaller (left), equal (middle), larger (right).

TST representation of a trie
Search in a TST

Follow links corresponding to each character in the key.
• If less, take left link; if greater, take right link.
• If equal, take the middle link and move to the next key character.

**Search hit.** Node where search ends has a non-null value.

**Search miss.** Reach a null link or node where search ends has null value.
26-way trie vs. TST

26-way trie. 26 null links in each leaf.

26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.

TST (155 null links)
A TST node is five fields:

- A value.
- A character $c$.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```java
private class Node {
    private Value val;
    private char c;
    private Node left, mid, right;
}
```

Trie node representations

- standard array of links ($R = 26$)
- ternary search tree (TST)

link for keys that start with $s$

link for keys that start with $su$
### String symbol table implementation cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>search hit</th>
<th>search miss</th>
<th>insert</th>
<th>space (references)</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>$L + c \lg^2 N$</td>
<td>$c \lg^2 N$</td>
<td>$c \lg^2 N$</td>
<td>$4N$</td>
</tr>
<tr>
<td>hashing</td>
<td>$L$</td>
<td>$L$</td>
<td>$L$</td>
<td>$4N$ to $16N$</td>
</tr>
<tr>
<td>R-way trie</td>
<td>$L$</td>
<td>$\log R N$</td>
<td>$L$</td>
<td>$(R + 1)N$</td>
</tr>
<tr>
<td>TST</td>
<td>$L + \log N$</td>
<td>$\log N$</td>
<td>$L + \log N$</td>
<td>$4N$</td>
</tr>
</tbody>
</table>

**Remark.** Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

**Bottom line.** TST is as fast as hashing (for string keys), space efficient.
TST vs. hashing

Hashing.
• Need to examine entire key.
• Search hits and misses cost about the same.
• Need good hash function for every key type.
• No help for ordered symbol table operations.

TSTs.
• Works only for strings (or digital keys).
• Only examines just enough key characters.
• Search miss may only involve a few characters.
• Can handle ordered symbol table operations (plus others!).
Summary

String keys (or keys interpreted as strings) can sometimes be handled more efficiently than keys that are only comparable:
- Radix sorting
- Radix search trees

More on strings in guest lecture on December 6:
Jesper Larsson, Data Compression
(Covers SW 5.5, which is part of the course syllabus.)

Next week: Trial exam.
Preparations: Print a copy of the June 2011 exam.
Trial exam: 12.00-14.00 (if you prefer to do this on your own, no problem)
Run-through / self-grading: 14.15-?