**Tries and Digital Search Trees I**

**Digital Search:**
- Keys are strings over some alphabet $\Sigma$
- Example: $\Sigma = \{a, b, c, \ldots\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, T, C, G\}$
- Assume chars coded as ints: $a = 0$, $b = 1$, $z = k - 1$
- Search $\sim$ length of query string [O(1) time per node]

**Analysis:**
- Space: Smaller by factor $K$
- Search Time: Larger by factor of $K$

**Example:**
- $\Sigma = \{a, b, a, b, c, c, a, c, a, b, c, c\}$

**Example:**
- $\Sigma = \{a, b, a, b, c, c, a, c, a, b, c, c\}$
- Assume $\Sigma = \{a, b, c\}$
- Keys: $\{aab, aba, abc, caa, cab, cbc\}$

**Example:**
- $\Sigma = \{a, b, a, b, c, c, a, c, a, b, c, c\}$
- Keys: $\{aab, aba, abc, caa, cab, cbc\}$

**Node:** Multiway of order $k$

**Example:**
- $\Sigma = \{a = 0, b = 1, c = 2\}$
- Keys: $\{aab, aba, abc, caa, cab, cbc\}$

**Example:**
- $\Sigma = \{a = 0, b = 1, c = 2\}$
- Keys: $\{aab, aba, abc, caa, cab, cbc\}$

**How to save space?**
- Store 1 char. per node

**Analysis:**
- Search: $\sim$ length of query string [O(1) time per node]
- Space: No. of nodes $\sim$ total no. of chars in all strings
- Space $\sim k \cdot \text{(no. of nodes)}$

**Digital Search:**
- Keys are strings over some alphabet $\Sigma$
- Example: $\Sigma = \{a, b, c, \ldots\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, T, C, G\}$
- Assume chars coded as ints: $a = 0$, $b = 1$, $z = k - 1$
- Search $\sim$ length of query string [O(1) time per node]
Patricia Tries:
- Improves trie by compressing degenerate paths
- PATRICIA = Practical Alg. to Retrieve Info. Coded in Alpha.
- Late 1960's: Morrison & Guchtenberger
- Each node has index field, indicates which char to check next (Increase with depth)

Dealing with long Paths:
- To get both good space & query time efficiency, need to avoid long degenerate paths.
- Path compression!

Example:
\[ ID S \]
\[ S_0: ajam... aj \]
\[ S_1: pajam... paj \]
\[ S_2: apaja... ap \]
\[ S_3: mapaj... map \]
\[ S_4: amapaj... amap \]
\[ S_5: pamapa... pam \]

Example: \[ S = \text{pamapajama} \]
- Def: Substring identifier for \[ S \] is shortest prefix of \[ S \] unique to this string
- \[ S_1: \text{ama} \]
- \[ E_1: \text{ID}(S_1) = \text{"amap"} \]
- \[ E_2: \text{ID}(S_7) = \text{"ama"} \]

Suffix Trees:
- Given single large text \[ S \]
- Substring queries: "How many occurrences of "tree" in CMSC 420 notes"
- Notation: \[ S = a_0a_1a_2...a_{n-1} \]
- Suffix: \[ S_i = a_ia_{i+1}...a_{n-1} \]
- Q: What is minimum substring needed to identify suffix \[ S_i \]?
Example: \( S = \text{pamapajama}$

Suffix Trees (cont.)

- \( S \) - text string \(|S| = n\)
- \( S_i = i^{th} \) suffix

Substring ID = min substr. needed to identify \( S_i \)

A suffix tree is a Patricia trie of the \( n+1 \) substring identifiers

PR k-d tree: Can be used for answering same queries as point k-d tree (orth: range, near neigh)

Geometric Applications:

PR k-d Tree: k-d tree based on midpoint subdivision
Assume points lie in unit square

Substring Queries:

- How many occurrences of \( t \) in text?
- Search for target string \( t \) in trie
  - if we end in internal node (or midway on edge) - return no. of extern. nodes in this subtree
  - else (fall out at extern. node)
    - compare target with string
      - if matches - found 1 occurrence
      - else - no occurrences

Example:

Search("ama") \rightarrow \text{End at intern node} \rightarrow \text{Report: 2 occs.}

Search("amapaj") \rightarrow \text{End at extern node}

Search trees III

Tries and Digital Search Trees

- Analysis:
  - Space: \( O(n) \) nodes
  - \( O(n \cdot k) \) total space \((k = |S| = \sigma(1))\)
  - Search time: \( n \) total length of target string
  - Construction time: \( \sim O(n \cdot k) \) [nontrivial]

Final tree:

Example:

Example: \( F = \text{pamapajamaS} \rightarrow \text{supfixtre.es} \) (cont)

Claim: This is a trie!
Binary Encoding:
- Assume our points are scaled to lie in unit square $0 \leq x, y < 1$ (can always be done)
- Represent each coordinate as binary fraction:
  - $x = 0.a_1 a_2 a_3 \ldots, a_i \in \{0, 1\}$
  - $x = \Sigma a_i \cdot 2^{-i}$

Example:
- $x = 0.1010110 \ldots$

PR kd-Tree $\equiv$ Trie ??

- Approach: Show how to map any point in $\mathbb{R}^n$ to bit string
- Store bit strings in a trie (alphabet $\Sigma^* = \{0, 1\}^*$)
- Prove that this trie has same structure as $kd$-tree

Further Remarks:
- Techniques for efficiently encoding, building, serializing, compressing...
  - Can generalize to any dimension
  - $x = 0.a_1 a_2 a_3 \ldots, a_i \in \{0, 1\}$
  - $y = 0.b_1 b_2 b_3 \ldots, b_i \in \{0, 1\}$

Tries and Digital Search Trees IV

Bit Interleaving:
- Given a point $p = (x, y)$
  - $0 \leq x, y < 1$
  - Let: $x = 0.a_1 a_2 a_3 \ldots$ in binary
  - $y = 0.b_1 b_2 b_3 \ldots$

Define:
- $\Phi(x, y) = a_1 b_1 a_2 b_2 a_3 b_3 \ldots$
  - Called Morton Code of $p$

How do we extend to 2-D?

Lemma: Given a pt set $P \subseteq \mathbb{R}^2$
- (in unit square $[0, 1]^2$) let $P = \{ p_1, \ldots, p_n \}$ where $p_i = (x_i, y_i)$
- Let $\Phi(P) = \{ \Phi(p_1), \Phi(p_2), \ldots, \Phi(p_n) \}$
  - (in binary strings)

Then the PR kd-tree for $P$ is equivalent to binary trie for $\Phi(P)$.

Proof: By induction on no. of bits
- Let $x = 0.a_1 a_2 \ldots, y = 0.b_1 b_2 \ldots$
  - and consider just $\Phi(x, y) = a_1 b_1 a_2 b_2 a_3 b_3 \ldots$

The PR kd-tree + binary trie assigns pts to same subtrees
- ($\ldots$ induction)
Deallocation Models:
- **Explicit**: (C++), programmer deletes, may result in leaks, if not careful
- **Implicit**: (Java, Python), runtime system deletes, Garbage collection, Slower runtime, Better memory compaction

Explicit Allocation/Deallocation:
- Heap memory is split into blocks whenever requests made
- Available blocks:
  - Merged when contiguous
  - Stored in available block list

Runtime System Mem. Mgr.:
- Stack - local vars, recursion
- Heap - for "new" objects
  - Don't confuse with heap data structure/heapsize

What happens when you do:
- `new` (Java)
- `malloc/free` (C)
- `new/delete` (C++)?

Block Structure:
- **Allocated**:
  - `malloc/free` (C)
  - `new/delete` (C++)
- **Available**:
  - Stored in available block list

Memory Management:
- **Guide**:
  - `prevInUse`: 1 if prev. contiguous block is allocated
  - `prev/next`: links in available list
  - `size`: total block size (includes headers)

Fragmentation:
- Results from repeated allocation and deallocation (Swiss-cheese effect)

How to select from available blocks?
- **First-fit**: Take first block from avail. list that is large enough
- **Best-fit**: Find closest fit from avail. list
- **Surprise**: First-fit is usually better - faster + avoids small fragments

External: Caused by pattern of alloc/dealloc

Internal: Induced by mem. manage. policies (not user)
Some C-style pointer notation

```c
void* - pointer to generic word of memory
void* p be of type void*:
p + 10 - 10 words beyond p
*(p+10) - contents of this
Let p point to head of block:
p->inUse, p->prevInUse, p->size
   - we omit bit manipulation
*(p+p->size-1) - references last word in this block
```

**Example:** Alloc b=59 → for head ∈ 60

### Allocation:

```c
malloc(b)
```

- Search avail. list for block of size \( b \geq b+1 \)
- If \( b \)’ close to \( b \): alloc entire block (unlink from avail list)
- Else: split block

**Deallocation:**

- If prev+next contiguous blocks are allocated → add this to avail
- Else - merge with either/both to make max. avail block

```c
Example:
```

**Memory Management II**

```c
(void*) alloc (int b) {
b++ // add +1 for header
p = search avail. list for block
   size >= b
   if (p == null) Error - Out of mem!
   if (p->size - b < TOO_SMALL) // update prevInUse for next contiguous block
      unlink p from avail. list
      q = p
   else .... (continued)
q->inUse = 1
(p->p.size) // new block header
p->p.size -= b // remove allocation
*(p+p->size-1) = p->size // size2
q = p + p->size // start of new block
q->p.size = b
q->prevInUse = 0
return q+1 // skip over header
```
**Buddy System:**
- Block sizes (including headers) are power of 2.
- Requests are rounded up (internal fragmentation).
- Block size $2^k$ starts at address that is multiple of $2^k$.
- $k =$ level of a block.

**Structure:**
- Laminar subdivision

**In practice:** There is a minimum allowed block size.

**Buddy system only allows allocations aligning with these blocks.**

**Coping with External Fragmentation:**
- Unstructured allocation can result in severe external fragmentation.
- Can we compress? Problem of pointers.
- By adding more structure we can reduce external frag. at cost of internal frag.

**Memory Management III**

**Merging:**
- When two adjacent blocks are available, we don't always merge them.
- Must have same size: $2^k$.
- Must be buddies - siblings.
- Def: $buddy_k(x) = \{ x + 2^k \text{ if } 2^k \text{ divides } x, x - 2^k \text{ otherwise}\}$
- $buddy_k(x) = (1 << k) \& x$ [Bit manipulation]

**Allocation:**
- $k = \lceil \lg (b+1) \rceil$ add 1 for header.
- if avail[k] non empty:
  - return entry + delete.
- else:
  - find avail[j] $\neq \emptyset$ for $j > k$.
  - split this block.

**Big Picture:**
- Avail list is organized by level: avail[k].
- Block header structure same as before except:
  - prevInUse $\neq$ not needed size 2.
Announcements:

1. Prog assign 3 - Due, Mon, May 9, 11:59 pm
   - Optional Prog Assign 0+1a+1b 2 drop lowest

2. HW 4 - Due, Tue, May 10 11:00 am
   - Optional - Drop lowest HW
   - HW grades - more variation
   - Questions on final exam - mod from HW4

3. Final Exam
   - Fri, May 13 4-6 pm, Tydings Bldg 0130
Example: \( \text{alloc}(2) \xrightarrow{\text{round up}} \text{alloc}(4) \)

\[
\begin{align*}
\text{avail} & \quad 0 \quad 3 \quad 4 \\
\text{free} & \quad 4 \quad 12 \quad \text{use} \\
\text{use} & \quad 16 \quad 21 \quad 22 \\
\end{align*}
\]

Fibonacci Buddy System

\[
\begin{array}{c}
2 \\
4 \quad 2 \\
\end{array}
\]