COMP 7720 - Online Algorithms

Self-Adjusting Trees & Paging

Shahin Kamali

Lecture 7 - Sep. 28, 2017

University of Manitoba
Review & Plan
Today’s objectives

- Self-Adjusting Trees
  - Splay trees
- Paging Problem
Self-Adjusting Trees
Self-Adjusting Lists

- The input is a set of requests to items in a list of length L
  - The goal is to update the list to adjust it into patterns in the input.
  - There is a lot of locality in the input sequence:
    ⟨ 2 2 2 2 2 1 1 3 3 3 3 3 3 1 1 2 2 2 ⟩
  - Move-To-Front is the best deterministic list-update algorithm
Self-Adjusting Binary Search Trees

- The input is a set of requests to items in a BST of size $N$.
  - The goal is to update the tree to adjust it into patterns in the input.
- There is a lot of locality in the input sequence.
- Can we apply Move-To-Front for trees?
Splay Trees Idea

- When there is a request to item $a$, adjust the tree so that $a$ becomes root in the new tree!
- Use tree rotations to ‘bubble up’ the accessed item.
- We say that we splay $a$ to become root in the adjusted tree
  - It is a natural extension of Move-To-Front to the lists.
Splaying Rotations General Idea

- Consider accessed item $a$, its parent $p$ and grand-parent $g$ (if they exist).
- Reorder $a$, $p$, and $g$ so that $a$ appears ‘above’ the other two
  - If $a$ is smallest/largest, $p$ and $g$ will be in one side of $a$.
  - If $a$ is in between, $p$ and $g$ will be on its left and right.
- After re-ordering $a$, $p$, and $g$, ‘place’ the following four subtrees in their appropriate position to save BST property:
  - the two subtrees of $a$
  - the sibling subtree of $p$
  - the sibling subtree of $g$
Splay Example

E.g., Access \( a = 12 \)
Splaying Cases (a bit more formal)

- The accessed node \( a \) is either
  - Root
  - Child of the root
  - Has both parent \( (p) \) and grandparent \( (g) \):
    - Zig-zig pattern: \( g \rightarrow p \rightarrow a \) is left-left or right-right
    - Zig-zag pattern: \( g \rightarrow p \rightarrow a \) is left-right or right-left
Access root

- if \( x \) is root, do nothing!
Access child of root

- When $x$ is child of the root, do a single AVL rotation to move it above its parent
  - It is called a zig operation
Access LR or RL grandchild

- When $x$ is left-child (resp. right-child) of $P$ and $p$ is right-child (resp. left-child) of $g$, do an AVL double rotation.

- It is called a zig-zag operation
Access LL or RR grandchild

- Reverse the order of $a, p,$ and $g$.
  - It is called a zig-zig operation.
Splay Example

- E.g., Access $a = 6$
Splay Example

- E.g., Access $a = 4$
Splaying: Intuition

- The accessed node is moved to ‘front’ (i.e., is now root)
- Let $b$ be a node on the access path from root to the accessed node $a$. If $b$ is at depth $d$ before the splay, its at about depth $d/2$ after the splay.
  - Overall, nodes which are ‘deep’ on the access path tend to move closer to the root
- Splaying gets amortized $O(\log N)$ amortized access time.
BST-Update problem

- BST-Update problem:
  - The input is an online sequence of requests to items in a BST.
  - Each probe for finding an item $x$ has cost 1.
  - On the path traversed from the root to $x$, the algorithm can make any number of rotations at a cost of 1 per rotation.

- **Dynamic Optimality Conjecture:** Splay tree is a competitive solution, i.e., it has a competitive ratio independent of the size $N$ of tree and length $n$ of sequence.
  - We know the competitive ratio of splay trees is $O(\log N)$

- The best existing algorithm is provided by self-adjusting Tango Trees, and has a competitive ratio of $O(\log \log N)$
Potential Project Topics

- Write a survey of the self-adjusting data structures (other than linked lists).
  - In particular, think of BSTs and other structures.
  - For example, is there any self-adjusting hash table? what about self-adjusting skip lists?

- Think about advice BST-Update algorithms with advice?
  - How many bits are sufficient to achieve an optimal algorithms?
Paging Problem
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size
- The input is an online sequence of requests to pages of size 1.
  - To serve a request to page $x$, it should be in the cache
- In case $x$ is not in the cache, a fault of cost 1 has happened
  - The goal is to minimize the total number of faults
- To bring $x$ to the cache, we might need to evict a page.
  - A paging algorithm is defined through its eviction policy

Cost (number of faults):

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e$$

\begin{array}{|c|c|c|}
\hline
 a & b & c \\
\hline
 a & b \\
\hline
 a & b & c & d \\
\hline
 a & e & c & d \\
\hline
\end{array}

\begin{array}{|c|c|c|}
\hline
 a \\
\hline
 a & b & c \\
\hline
 a & e \\
\hline
\end{array}
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 5 6 7

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a$$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>e</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>e</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>e</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>e</td>
<td>c</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
First-In-First-Out (FIFO)

- FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 5 6 7

\[ \sigma = a\ b\ c\ b\ a\ d\ c\ e\ f\ a \]

\begin{array}{|c|c|c|c|}
\hline
a & b & c & d \\
\hline
e & b & c & d \\
\hline
e & f & c & d \\
\hline
e & f & a & d \\
\hline
\end{array}
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>e</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>f</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>f</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```
Optimal Paging Algorithm

Theorem

*Furthest-In-Future (FIF) is the optimal offline algorithm for paging.*

- We will see the proof in the next class!