**TDDB56 – DALGOPT-D**

Algorithms and optimization

Lecture 7

Splay Trees.
Priority Queues, Heap

---

**Splay Tree – basic idea...**

Recall the basic BST:
- Simple insert and reasonable delete when balanced, but...
- The "balance" is determined by order of inserts and deletes...

Combine with the "keep recent objects first" heuristics for lists?
- Often-used elements should be near the root!

---

**Splay Tree properties**

- A new operation Splay(r, T) modifies tree T:
  - Element r becomes the new root of T if it exists
  - Otherwise, the new root will be the inorder predecessor (or successor) of the "non-existent" r
- All operations are implemented using Splay:
  - LookUp(k, T):
    - Splay(k, T); if root(T) = k, return <k,i>
  - Insert(k, i, T):
    - Splay(k, T); if root(T) = k, update <k,i>, else insert new root <k,i> in T.
  - Delete(k, T): ....special, but it also involves "Splay" ☺
The Splay($k$, $T$) operation:
1. Perform a normal search for $k$, remember all nodes we pass...
2. Label the last node we inspect $P$
   - If $k$ is in $T$, then $k$ is in node $P$
   - Otherwise $P$ is an empty (external) child
3. Return back to root, at each node do a rotation to move $P$ up the tree... (3 cases)

Case 1: Parent($P$) is the root: rotate around $P$

Case 2: $P$ and Parent($P$) are both left children (or both right children): perform two rotations to shift up $P$:

Case 3: One of $P$ and Parent($P$) is a left child and the other is a right child: Perform two rotations in different directions:

Note: These rotations may increase the height of the tree...

Delete ($k$, $T$)
- We need help procedure Concat ($T_i$, $T_j$) where $T_1$ and $T_2$ are trees such that: $\forall k_i \in T_i, \forall k_j \in T_j : k_i < k_j$
- $\text{Concat} (T_i, T_j)$:
  - $\text{Splay}(\infty, T_i)$ ...will re-structure $T_i$ to have the largest element as root, and the root has no right child.
  - $\text{setRightChild}(T_i, T_j)$ ...reinstall $T_j$ as right child.
- $\text{Delete}(k, T)$:
  - $\text{Splay}(k, T)$ ...if root does not contain $k$, ok
  - $\text{Concat}(\text{leftChild}(k), \text{rightChild}(k))$

$\infty$ is a dummy key larger than all existing, valid keys...
Splay Trees - performance

- Each operation may face a totally unbalanced tree – thus not guaranteed to operate in $O(\log n)$ in worst case
- Amortized time is logarithmic:
  - Any sequence of length $m$ of these operations, starting with an empty tree, will take a total amount of $O(m \log n)$ time...
  - Thus, the amortized cost/time is $O(\log n)$ although individual op’s may be significantly worse...

Priority Queues

Commonly encountered situation:
- Waiting list (tasks, passengers, vehicles entering a ferry, phone calls)
- If a resource is freed, choose an element from the waiting list
- The choice is based on some partial ordering:
  - some tasks are more essential to achieve the goal,
  - some passengers should be served before the others (children, sick people)
  - fire dept. and first aid vehicles have priority

How to organize prioritized service?

ADT Priority Queue

- Linearly ordered set $K$ of keys
- We store pairs $<k, i>$ (as in Dictionary), multiple pairs with same key are allowed.
- A frequent operation is retrieving pairs with minimal key.

Operations on a Priority Queue $PQ$:

- $\text{MakeEmpty}(PQ)$
- $\text{IsEmpty}(PQ)$
- $\text{Insert}(PQ, k, i)$
- $\text{FindMin}(PQ)$ find $(k, i) \in PQ$ with minimal $k$, return $(k, i)$
- $\text{DeleteMin}(PQ)$ delete $(k, i) \in PQ$ with minimal $k$, return $i$
- $\text{DecreaseKey}(PQ, k, i, k')$ change priority $k$ of $(k, i) \in PQ$ to $k'$

Implementing Priority Queues

Searching for a minimal element in a search tree (BST, AVL, 2-3, ...)

- BST does not admit repeated keys, extension needed
- Skip list: OK, $\text{FindMin}$ in $O(1)$ time, but worst-case insert/delete $O(n)$

Another idea: keep the minimal element as the root of the tree

- Partially ordered tree

This is a complete binary tree!

Partially ordered binary tree:
- the key of a parent is less than or equal to the key of each child
- "last" leaf

This is also called a HEAP
**Updates on a Heap Structure**

- **DeleteMin** = deletion of the root
  - Replace root by *last leaf*
  - Restore partial order by swapping nodes downwards “down-heap bubbling”

- **Insert**
  - Insert new node after last leaf
  - Restore partial ordering by “up-heap bubbling”

**HEAP Properties:**
- `size()`, `FindMin()`: \( O(1) \)
- `insertItem()`, `DeleteMin()`: \( O(\log n) \)

Recall vector representation of BST!
- A complete binary tree...
- Compact vector representation
- Bubble-up and bubble-down have fast implementations

**Variant of Heaps:**
- Kind of partial ordering:
  - `minKey` in the root
  - `maxKey` in the root

- Kind of vector representation:
  - Forward level-order numbering (starting with 0 or 1)
  - Backward level-order numbering (starting with 0 or 1)