Red-Black Tree

• A Binary Search Tree.
• Every node in this tree is colored in either Red or Black.
• A historically popular alternative to the AVL tree.
• Operation on red-black trees take $O(\log n)$ time in the worst case.

Red-Black Tree

• Root Property: The root is black
• External Property: Every external node is black
• Internal Property: The children of a red node are black
• Depth Property: All the external nodes have the same black depth
Height of a Red-Black Tree

- **Theorem:** A red-black tree storing \( n \) items has height \( \Theta(\log n) \)

  **Proof:**

  **Depth Property:** All external nodes have same black depth \( d \).
  
  If all nodes were black then
  
  \( d \leq \log(n+1) \)

  **Internal node Property:** The children of a red node are black.
  
  i.e. \( h \leq 2d \)

  Thus
  
  \( \log(n+1) \leq h \leq 2 \log(n+1) \)

  so height is \( \Theta(\log n) \)

By the above theorem, searching in a red-black tree takes \( \Theta(\log n) \) time.

Insertion

- we execute the insertion algorithm for binary search trees
- Let the newly inserted node is root then color it black else color it red
- We preserve the root, external, and depth properties
  - If the parent of the node is black, we also preserve the internal property and we are done
  - Else (parent is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Example: Sequence 6, 3, 8, 4

Insert 4

Remedying a Double Red

- Consider a double red with child \( z \) and parent \( v \), and let \( w \) be the sibling of \( v \)

  **Case 1:** \( w \) is black
  
  - Restructure: Same as done for AVL trees.
Restructuring
- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- The internal property is restored and the other properties are preserved

Remedying a Double Red
- Consider a double red with child \( z \) and parent \( v \), and let \( w \) be the sibling of \( v \)
  
  **Case 2: \( w \) is red**
  - The double red corresponds to an overflow
  - Recolor and continue up
Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling.
- The parent \( v \) and its sibling \( w \) become black and the grandparent \( u \) becomes red, unless it is the root.
- The double red violation may propagate to the grandparent \( u \).

Analysis of Insertion

```
Algorithm insertItem(k, e)
1. We search for key k to locate the external insertion node z
2. We add the new item (k, e) at node z and color z red
3. while doubleRed(z)
   if isBlack(sibling(parent(z)))
     z ← restructure(z)
     return
   else if sibling(parent(z)) is red
     z ← recolor(z)
```

Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees.
- Let \( v \) be the internal node removed, \( w \) the external node removed, and \( r \) the sibling of \( w \).
  - If either \( v \) or \( r \) was red, we color \( r \) black and we are done.
  - Else \( (v \) and \( r \) were both black) we color \( r \) double black, which is a violation of the internal property requiring a reorganization of the tree.
- Example the deletion of 4 is simple:
Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v or r was red, we color r black and we are done
  - else (v and r were both black) we color r double black, which is a violation of the internal property requiring a reorganization of the tree
- Example where the deletion of 8 causes a double black:

Remedying a Double Black

- The algorithm for remedying a double black node r with sibling w considers three cases
  - Case 1: w is black and has a red child
    - We perform a restructuring, equivalent to a transfer, and we are done
  - Case 2: w is black and its children are both black
    - We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation
  - Case 3: w is red
    - We perform an adjustment, after which either Case 1 or Case 2 applies
- Deletion in a red-black tree takes $O(\log n)$ time

Remedying a Double Black

Case 1: w is black and has a red child
- We perform a restructuring
- We color r black
- We color a and c black.
- We color b with the same color as the parent of r.
Remedying a Double Black

Case 2: \( w \) is black and its children are both black
- We perform a re-coloring
  - We color \( r \) black
  - We color \( w \) red
  - If the parent is red we color it black else we color it double black.
Summary of Red-Black Trees

- Insertion or deletion may cause local perturbation
  - two consecutive red nodes or a double-black node.
- The perturbation is either
  - resolved locally (restructuring)
  - propagated to a higher level in the tree by re-coloring
- $O(1)$ time for a restructuring or re-coloring
- At most one restructuring per insertion and at most two restructuring per deletion.
- $O(\log n)$ re-coloring
- Total time: $O(\log n)$

Skip Lists

Skip List

Question:
- Can we create a structure that adds the best properties of Array and Linked list Data Structure?
  - Query: $O(\log n)$ in sorted Arrays
  - Insert/Removal: $O(1)$ in Linked List
What is a Skip List

- A skip list for a set $S$ of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that:
  - Each list $S_i$ contains the special keys $+\infty$ and $-\infty$
  - List $S_h$ contains the keys of $S$ in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
  - List $S_h$ contains only the two special keys

- We show how to use a skip list to implement the dictionary ADT.

Search

- We search for a key $x$ in a skip list as follows:
  - We start at the first position of the top list.
  - At the current position $p$, we compare $x$ with $y \leftarrow \text{key}(\text{after}(p))$.
    - $x = y$: we return $\text{element}(\text{after}(p))$.
    - $x > y$: we "scan forward".
    - $x < y$: we "drop down".
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY.

Example: search for 78

Insertion

- To insert an item $(x, o)$ into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with $i$ the number of times the coin came up heads.
  - If $i \geq h$, we add to the skip list new lists $S_{i+h}, \ldots, S_h$, each containing only the two special keys.
  - We search for $x$ in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than $x$ in each list $S_0, S_1, \ldots, S_i$.
  - For $j = 0, \ldots, i$, we insert item $(x, o)$ into list $S_j$ after position $p_j$.

Example: insert key 15, with $i = 2$.
Deletion

- To remove an item with key $x$ from a skip list, we proceed as follows:
  - We search for $x$ in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key $x$, where position $p_j$ is in list $S_j$.
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$.
  - We remove all but one list containing only the two special keys.
- Example: remove key 34

![Diagram showing deletion process]

Implementation

- We can implement a skip list with quad-nodes.
- A quad-node stores:
  - item
  - link to the node before
  - link to the node after
  - link to the node below
  - link to the node above
- Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them.

Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm.
- We use the following two basic probabilistic facts:
  - Fact 1: The probability of getting $i$ consecutive heads when flipping a coin is $1/2^i$.
  - Fact 2: If each of $n$ items is present in a set with probability $p$, the expected size of the set is $np$.
- Consider a skip list with $n$ items:
  - By Fact 1, we insert an item in list $S_i$ with probability $1/2^i$.
  - By Fact 2, the expected size of list $S_i$ is $n/2^i$.
- The expected number of nodes used by the skip list is
  \[
  \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} < 2n
  \]
- Thus, the expected space usage of a skip list with $n$ items is $O(n)$.
Height

- The running time of the search and insertion algorithms is affected by the height $h$ of the skip list.
- We show that with high probability, a skip list with $n$ items has height $O(\log n)$.
- We use the following additional probabilistic fact:
  Fact 3: If each of $n$ events has probability $p$, the probability that at least one event occurs is at most $np$.
- Consider a skip list with $n$ items.
  - By Fact 1, we insert an item in list $S_i$ with probability $1/2^i$.
  - By Fact 3, the probability that list $S_i$ has at least one item is at most $2^{-i}$.
  - By picking $i = 3\log n$, we have the probability that $S_{3\log n}$ has at least one item is at most $2^{-3\log n} = 2/n^3$.
  - Thus a skip list with $n$ items has height at most $3\log n$ with probability at least $1 - 2/n^3$.

Search and Update Times

- The search time in a skip list is proportional to:
  - the number of drop-down steps, plus
  - the number of scan-forward steps.
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability.
- To analyze the scan-forward steps, we use yet another probabilistic fact:
  Fact 4: The expected number of coin tosses required in order to get tails is 2.
- When we scan forward in a list, the destination key does not belong to a higher list.
  - A scan-forward step is associated with a former coin toss that gave tails.
  - By Fact 4, in each list the expected number of scan-forward steps is $2$.
  - Thus, the expected number of scan-forward steps is $O(\log n)$.
  - We conclude that a search in a skip list takes $O(\log n)$ expected time.
  - The analysis of insertion and deletion gives similar results.

Summary

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm.
- In a skip list with $n$ items, the expected space used is $O(n)$.
- The expected search, insertion and deletion time is $O(\log n)$.
- Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability.
- Skip lists are fast and simple to implement in practice.