Hashing

To store n keys, we need a table slightly bigger than n of size m

Load factor $\lambda = n/m$. Running time increases as $\lambda \to 1$

Hash Function : $h : \texttt{Keys} \to \mathbb{R}$

Scatters keys randomly and need to handle collision

Good Hash Function: Efficient to compute, low collisions

Use every bit of the key and break up clusters

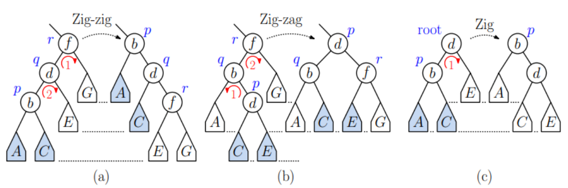
Common Examples:
- Division hash: $h(x) = x \bmod m$
- Multiplicative hash: $h(x) = (ax \bmod p) \bmod m$
  a, p - large prime numbers
- Linear hash: $h(x) = ((ax+b) \bmod p) \bmod m$
  a, b, p - large primes

Assume keys can be interpreted as ints

Modding by prime scatters keys, m may not be primed

If quadratic probe, table size m prime, $\lfloor m/2 \rfloor$ probe sequences distinct

Splay Trees




```
insertion(x) {
    splay(x) // pull x's pred/succ to root
    q = new Node(x, v)
    if (p.key < x) { // p is predecessor?
        q.left = p // put p to our left
        q.right = p.right // ... and p's right to our right
    p.right = null
}

}
find(x) {
    splay(x)
    if (root.key == x) { return root }
    else return null;
}
delete(x) {
        Node p = splay(x) // pull x to root
        if (p.key != x) Error! "Non-existant" // Oops! x is not here
        p.right.splay(x) // splay x in p's right subtree
        Node q = p.right // q is p's inorder successor
        q.left = p.left // transfer left child to q
        root = q // make q the new roo
}
```

Start with an empty dict, any sequence of m accesses take time O(mlogn+nlogn) DFingerThm: Cost of a search starting from element y to x is the log of the of elements between

Working-Set Thm: if we accessed an element t steps ago, then the time to access it now is rougly log t.

Treaps

Insertion times increase as we walk down the tree

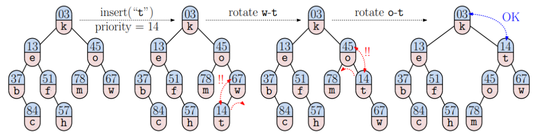
If keys are inserted in random order, expected height of BST O(logn).

Treap behave as if keys are inserted in random order

Timestamp increases from root to leaf. Assign random priority to keys

Treap has O(logn) expected height over all n! orderings

Insertion: BST insert, rotate left/right if parent priority is bigger



Deletion: Set node priority to ∞, rotate to leaf, delete

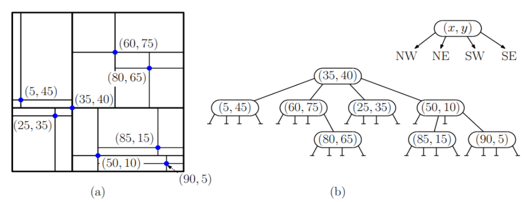
With standard binary search trees, the expectation was over all n! insertion orders. With treaps, the expectation was over all n! orders of the priority values. The latter is preferred, because the data structure's expected performance is not dependent on the insertion order

If just two keys have the same priority, their parent/child relationship might be affected, but the structure will be fine.

1) If $a > b^k$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log\log n\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

QuadTrees

Each child is a division of a subspace/quadrant. In d-dimensions, we have 2^d child. Above dim 3, these trees are too large



Skip List

Total space $= \sum_{i=0}^{\infty} n/2^i = 2n$

```
Value forwardSearch(SkipNode p, Key y) {
    int i = 0 // start at the lowest level
    while (i >= 0) { // (level will rise then fall)
        if (p.next[i].key <= y) { // can we move forward?
            if (i+1 < p.next.length) i++ // move up, if possible
            else p = p.next[i] // move horizontal, if not
        else i-- // drop down a level
    }
    return (p.key == y ? p.value : null) // return value if found
}

Value getMinK(int k) {
    int i = topmostLevel // start at topmost nonempty level
    SkipNode p = head // start at head node
    int count = k // number of items remaining
    while (count > 0) { // repeat until exhausting the span
        if (p.span[i] <= count) {
            count -= p.span[i] // decrement count by the number span
            p = p.next[i] // advance along same level
        }
        else i-- // drop down a level
    }
    return p.key // return final element
}
```

Insertion: Save references on nodes where next is greater than the new value

In expectation the number of nodes visited at any level is 2

We start with n nodes at level 0, on average pn nodes survive to level 1, p^2n survive to level 2, and in general we expect p^in to survive to level i.

Let h denote the number of levels in the skip list. (We actually don't care what this value is.) Summing the number of nodes that contribute to each level of the skip list, and employing the fact that, for $0 < c < 1$, $\sum_{i=0}^{\infty} c^i = 1/(1-c)$, the total expected number of links is

$$\sum_{i=0}^{h} p^i n \leq n \sum_{i=0}^{\infty} p^i = \frac{n}{1-p};$$

Observe that for any constant p, this is $O(n)$.

Kd-Tree

Deletion: Find minimum node in the right subtree with the minimum x/y according to cutdim. If right is empty, find min node in left and make left the new right. This can lead to $O(\sqrt{n})$ height over large insert/deletes

Splay: Any individual operation can be $\Omega(n)$

If each element of a splay tree is accessed in ascending (or descending) order the total time for all these accesses is O(n). (Scanning theorem)

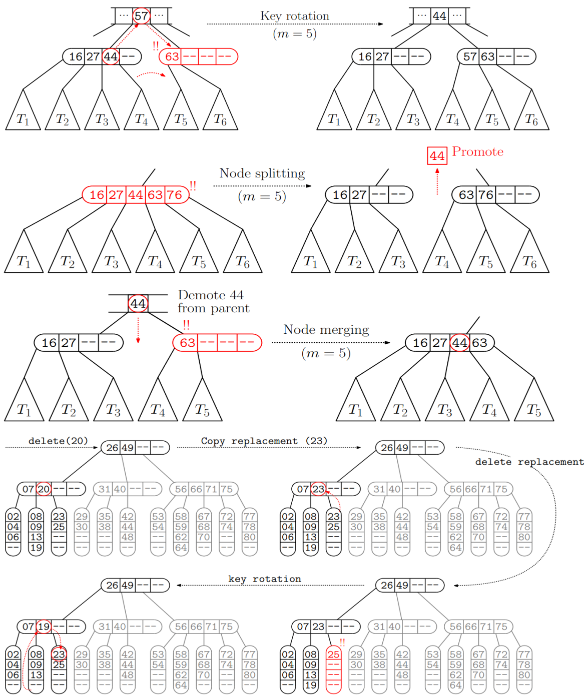
B-Trees

Designed so that each node fits in one page of memory

B-tree: of order m (≥ 3) Root of leaf or has ≥ 2 children

Non root nodes have $\lceil m/2 \rceil$ to m children, all leaves same level

B-trees (order m) have height $\frac{lg(n)}{\gamma}$, $\gamma = lg\frac{m}{2}$



Misc

Guaranteed to succeed in finding slot: Linear Probing
Double hashing, where the table size m and secondary hash function g(x) are relatively prime

```
TreapNode expose(Key x, TreapNode p) {
    if (p == null) // error - key not in tree
        throw Exception("Key not found");
    else if (x < p.key) { // x is smaller - search left
        p.left = expose(x, p.left);
        return rotateRight(p); // rotate the exposed node up
    }
    else if (x > p.key) { // x is larger - search right
        p.right = expose(x, p.right);
        return rotateLeft(p); // rotate the exposed node up
    }
    else { // found it
        p.priority = Integer.MIN_VALUE; // set priority to -infinity
        return p;
```

Querying Kd-Tree

```
int rangeCount(Rectangle R, KDNode p, Rectangle cell) {
    if (p == null) return 0 // empty subtree
    else if (R.isDisjointFrom(cell)) // no overlap with range?
        return 0
    else if (R.contains(cell)) // the range contains our entire cell?
        return p.size // include all points in the count
    else { // the range stabs this cell
        int count = 0
        if (R.contains(p.point)) // consider this point
        count += 1
        // apply recursively to children
        count += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.point))
        count += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return count
    }
}

Point rayShoot(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best
    else if (cell.high.x < q.x || cell.high.y < q.y) // no overlap
        return best
    else {
        if (p.point.x >= q.x && p.point.y >= q.y && p.point.x < best.x)
            best = p.point // p.point is new best
        // get child cells
        Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
        if (cell.lo.y >= q.y && p.cutDim == 0) {
            if (p.point.x > q.x) // no need to search right
                best = rayShoot(q, p.left, leftCell, best)
            else // no need to search left
                best = rayShoot(q, p.right, rightCell, best)
        } else {
            best = rayShoot(q, p.left, leftCell, best)
            best = rayShoot(q, p.right, rightCell, best)
        }
    return best;
    }
}
```

$O(n^{1-1/d}$, orthogonal range query time

ScapeGoat Trees

Height will always be O(logn) because of rebuild event

$\left\lfloor \frac{k}{2} \right\rfloor$, m increment m for inserts, if $m > 2n$ rebuild

Any m operations take $O(mlogm)$, amortized $= O(logm)$

Must there be a scapegoat? The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has (roughly) more than twice as many nodes as its sibling. We call such a node on the search path a *scapegoat candidate*. A short way of summarize the above process is "rebuild the scapegoat candidate that is closest to the insertion point.'

You might wonder whether we will necessarily encounter an scapegoat candidate when we trace back along the search path. The following lemma shows that this is always the case.

Lemma: Given a binary search tree of n nodes, if there exists a node p such that depth$(p) > \log_{3/2} n$, then p has an ancestor (possibly p itself) that is a scapegoat candidate.

Proof: The proof is by contradiction. Suppose to the contrary that no node from p to the root is a scapegoat candidate. This means that for every ancestor node u from p to the root, we have size$(u.\text{child}) \leq \frac{2}{3} \cdot$ size(u).

We know that the root has a size of n. Therefore, its child on the search path has size at most $(2/3)n$, its grandchild has size at most $(2/3)((2/3)n) = (4/9)n$, and generally the node at depth i along the search path as size at most $(2/3)^i n$.

Let d denote the depth of p. We know what its subtree rooted at p must have at least one node (namely p itself), and therefore

$$1 \leq \text{size}(p) \leq \left(\frac{2}{3}\right)^d n.$$

Solving for d, we have

$$\left(\frac{3}{2}\right)^d \leq n \implies d \leq \log_{3/2} n.$$

```
Node buildSubtree(Key[] A) { // A is a sorted array of keys
    k = A.length
    if (k == 0) return null // empty array
    else {
        j = floor(k/2) // median of the array
        Node p = new Node(A[j]) // ...this is the root
        p.left = buildSubtree(A[0..j-1])
        p.right = buildSubtree(A[j+1..k-1])
        return p // return root of the subtree
    }
}
```

B+Trees

- Internal and leaf nodes are different in structure:
  - Internal nodes store keys only, no values. The keys in the internal nodes are used solely for locating the leaf node containing the actual data, so it is not necessary that every key appearing in an internal node need correspond to an actual key-value pair.
  - All the key-value pairs are stored in the leaf nodes. There is no need for child pointers. (This also saves space.)
- Each leaf node has a *next-leaf* pointer, which points to the next leaf in sorted order.

Storing keys only in the internal nodes saves space, and allows for increased fan-out. This means the tree height is lower, which reduces number of disk accesses. Thus, the internal nodes are merely an *index* to locating the actual data, which resides at the leaf level. (The policy regarding which keys a subtree contains are changed. Given an internal node with keys $\langle a_1, \ldots, a_{j-1} \rangle$, subtree T_j contains keys x such that $a_{i-1} < x \leq a_i$.)

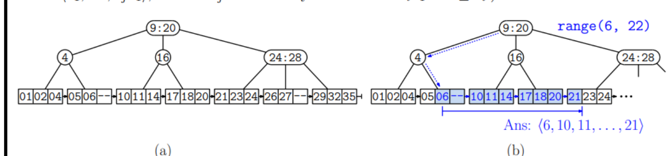


Fig. 9: B+ tree of order $m = 3$, where leaves can hold up to 3 keys.

The next-leaf links enable efficient *range reporting* queries. In such a query, we are asked to list all the keys in a range $[x_{\min}, x_{\max}]$. We simply find the leaf node for $x_{\min}$ and then follow next-leaf links until exceeding $x_{\max}$.