Set: Balanced Tree

Hwansoo Han
## Time Analysis in Operations

- **MEMBER, INSERT, DELETE**

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Balanced Search Tree</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

$N$: number of elements in set
2-3 Tree
Balanced Tree – 2-3 tree

- 2-3 tree
  - Each interior node has two or three children
  - Each path from the root to a leaf has the same length
  - Special cases: tree with zero node or one node
  - Each element in a set is placed at a leaf

- Each interior node keep one or two keys
  - The smallest element that is a descendant of the second child
  - The smallest element that is a descendant of the third child, if the third child exists
Balanced Tree – 2-3 tree

- Divide descendants into two or three groups
  - $x < \text{key}_{\text{first}}$: subtree descending from the 1st child
  - $\text{key}_{\text{first}} \leq x < \text{key}_{\text{second}}$: subtree descending from the 2nd child
  - $\text{key}_{\text{second}} \leq x$: subtree descending from the 3rd child

```
7
/|
7 16
/  |
5  8 12
/|
2 5 7 8 12
/|
2 5 7 8 12
/|
2 5 7 8 12
```
2-3 Tree – operations

- INSERT \((x, T)\)
  - Elements are all in the leaf nodes
  - Descending the tree \(T\), reach at a node which has leaves as children and \(x\) should be inserted as a child
    - If that node has only two children, add \(x\) as a child
      - Place \(x\) in appropriate order
      - Then, adjust the keys in the interior node
    - If three children, split that interior node.
      - Two smaller elements stay with the original node and two larger elements move to a new node.
      - Then the new node is inserted to the parent of that interior node
      - INSERT is performed recursively up to the root, if necessary
2-3 Tree - operations

- INSERT(18, T)
2-3 Tree - operations

- INSERT(10, T)
2-3 Tree - operations

- **DELETE** \((x, T)\)
  - After deletion of a leaf containing \(x\), its parent node \(node\) may have only one child
    - If \(node\) is root, delete \(node\) and make its lone child be the new root
    - Otherwise, look for \(p\), parent of \(node\). If any children adjacent to \(node\) to the right or left has three children, transfer proper one to \(node\)
    - If all adjacent nodes have two children, transfer the lone child of \(node\) to an adjacent sibling of \(node\), and delete \(node\)
2-3 Tree - operations

- DELETE (10, T)
2-3 Tree - operations

- DELETE (7, T)
2-3 Tree - operations

- Final result

```
  12 18
 /    |
5  8   16
 |     |
2 5 8 12 16
 |     |
2 5 8 12 16
```

```
B-Tree
B-Tree

- Balanced search tree
  - Height = $O(\log N)$ for the worst case
  - Keys are stored in internal nodes and leaves
- Designed to work well on storage devices (e.g. disk)
  - Show good performance on disk I/O operations
- Widely used in database systems
- Variants
  - $B^+\text{Tree}$: keys and records are stored in leaves
    - Copies of keys are stored in internal nodes
    - Leaves are linked in a list for a fast sequential search
  - $B^*\text{Tree}$: dense node (at least $2/3$ full, instead $1/2$ full)
B-Tree: Motivation

- Data structures on secondary storage
  - Main memory is fast but expensive
  - Magnetic disks are cheaper and high capacity, but slow

- B-trees try to read as much information as possible in every disk access operation
B-Tree: Definition

- B-tree $T$ is a rooted tree (with root $root[T]$)
  - Every node $x$ has following fields
    - $n[x]$: number of keys currently stored in node $x$
    - keys: $n[x]$ keys stored in non-decreasing order
      \[ key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \]
    - leaf$[x]$: a boolean value – true if $x$ is a leaf, false if $x$ is an internal node
    - child pointers: $c_1[x], c_2[x], \ldots, c_{n[x]}[x]$, if $x$ is an internal node
B-Tree: Definition (cont’d)

Properties (cont’d)

- keys $key_i[x]$ separate the ranges of keys stored in each subtree:
  - if $k_i$ is any key stored in a subtree with root $c_i[x]$, then
    $$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_n[x][x] \leq k_{n[x]+1}$$

- all leaves have the same height, which is the tree’s height $h$

- upper & lower bounds on the number of keys on a node

  minimum degree of B-tree: $t$ (a fixed integer $t \geq 2$)
    - lower bound: at least $t-1$ keys (at least $t$ children), except root
    - upper bound: at most $2t-1$ keys (at most $2t$ children)
Height of B-Tree

- Example (worst case): B-tree of height 3
  - Containing a **minimum** possible keys

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$2t$</td>
</tr>
<tr>
<td>3</td>
<td>$2t^2$</td>
</tr>
</tbody>
</table>

- Each node shows the number of keys contained
Height of B-Tree (cont’d)

- Number of disk accesses proportional to the height
- Worst-case height $h$, if number of keys $n$
  - $n \geq 1 + (t-1) \sum_{i=1}^{h} 2^{i-1} = 1 + 2(t-1) \frac{(t^h-1)}{(t-1)} = 2t^h - 1$
  - $h \leq \log_t \frac{n+1}{2} \approx O(\log_t n)$

- Main advantage of B-tree compared to 2-3 tree
  - Design B-tree of minimum degree $t$ can be much larger
  - Height can be reduced by a factor of $\log t$
    - Number of disk accesses can be reduced as much
Basic Operations on B-Tree

- Operations
  - SEARCH (MEMBER)
  - INSERT
  - DELETE

- Algorithms
  - Root of B-tree is always in main memory
  - DISK-READ of a node along the path downward the B-tree
B-Tree Search

- \( O(\theta h) \) (= \( O(t \log_t n) \)) algorithm
  - Number of disk pages accessed: \( O(h) = O(\log_t n) \)
  - While loop in a linear search of keys: \( O(t) \)

```plaintext
function B-TREE-SEARCH(x, k) returns (y, i) such that key_i[y] = k or NIL
    i ← 1
    while i ≤ n[x] and k > key_i[x]
        do i ← i + 1
    if i ≤ n[x] and k = key_i[x]
        then return (x, i)
    if leaf[x]
        then return NIL
    else DISK-READ(c_i[x])
        return B-TREE-SEARCH(c_i[x], k)
```

B-Tree Insert

- Splitting a full node $y$ ($2t-1$ keys): B-TREE-SPLIT-CHILD($x, i, y$)
  - Split into two $t-1$ nodes and move up median key $y$’s parent
  - $x$ should be guaranteed to be non-full
B-Tree Insert

- Key is always inserted in a leaf node
- Insert is done in a single pass down the tree
  - $O(h) = O(\log t n)$ disk accesses
  - $O(th) = O(t \log t n)$ CPU time
- While descending down the tree, full nodes are split
  - When passing up a median key, its parent node guaranteed to be non-full
B-Tree Insert – Example (I)

Initial tree
\( t=3 \)

2 inserted

17 inserted
B-Tree Insert – Example (II)

Initial tree

\[ t=3 \]

- 1 2 3 4 5
- 10 11 14 15
- 17 18 19
- 21 22
- 25 26

12 inserted

- 1 2 3 4 5
- 10 11 12
- 14 15
- 17 18 19
- 21 22
- 25 26

6 inserted

- 1 2 4 5 6
- 10 11 12
- 14 15
- 17 18 19
- 21 22
- 25 26
B-Tree Delete

- Keys are deleted from any node
- **Delete is done in a single pass down the tree**
  - similar to insert, but with a few special cases
  - $O(h) = O(\log_t n)$ disk accesses
  - $O(th) = O(t \log_t n)$ CPU time
- Descending down the tree to a node with at least $t$ keys
  - Otherwise, rebalance the node with siblings and parent
  - Key in a leaf can be deleted right away
  - Key in an internal node is deleted and replaced by its predecessor (or successor) in a leaf
B-Tree Delete

- \(k\): the key to be deleted, \(x\): the node containing the key

  Case 1: if \(k\) is in leaf \(x\), simply delete \(k\) from \(x\)

  Case 2: if \(k\) is in internal node \(x\),
  - a/b) if child \(y\) (or \(z\)) that precedes (or follows) \(k\) in node \(x\) has at least \(t\) keys, find \(k'\) which is predecessor (or successor) in subtree rooted at \(y\) (or \(z\)). Recursively delete \(k'\) and replace \(k\) with \(k'\) in \(x\)
  - c) if both children \(y\) and \(z\) have \(t-1\) keys, merge \(k\) and all of \(z\) into \(y\), so that \(k\) and pointer to \(z\) are eliminated from node \(x\). Recursively, delete \(k\) from \(y\).

  Case 3: if \(k\) is not present in internal node \(x\), determine subtree rooted at \(c_i[x]\) that must contain \(k\). If the subtree has only \(t-1\) keys, do the following steps
  - a) if a sibling has \(t\) keys, move a key from \(x\) down to \(c_i[x]\) and move a key from \(c_i[x]\)'s immediate left or right sibling up to node \(x\)
  - b) if all of siblings have \(t-1\) keys, merge two siblings and move key from \(x\) down to the merged node (median key in the merged node)
B-Tree Delete – case 1

- Delete 6
B-Tree Delete – case 2a (2b)

- Delete 13
B-Tree Delete – case 2c

- Delete 7

```
3 7 12
1 2 4 5 10 11 14 15
```

```
3 12
1 2 4 5 10 11 14 15
```

```
16
20 23
17 18 19
21 22
24 26
```
B-Tree Delete – case 3b

- Delete 4
B-Tree Delete – case 3a

Delete 2
Skip List

Hwansoo Han (SKKU)
Dynamic Search Structures

- Examples
  - Treaps
  - B-tree
  - skip list
- Dynamic
  - insert, delete, search
- Fast operation - $O(\log N)$
  - $O(\log N)$ with a high probability
Skip List

- Operations in $O(\log N)$ time with a high probability
  - search, insert, delete
  - same asymptotic time complexity as B-tree
  - simpler implementation with skip pointers
Skip List

- Operations in $O(\log N)$ time with a high probability
  - search, insert, delete
  - same asymptotic time complexity as B-tree
  - simpler implementation with skip pointers
“Express Lane”

- Skipping along the express lane

- Skipping every other node
  - additional memory usage: $N/2$
  - search time: $\lceil N/2 \rceil + 1$ (at max)
Another “Express Lane”

- Add another express lane

- Skipping 3 nodes and link the 4th node
  - additional memory usage: $3/4 \times N \ (= N/2 + N/4)$
  - search time: $\lceil N/4 \rceil + 2$ (at max)
Generalized “Express Lanes”

- Add more “express lanes” by doubling the distance

- Every \( (2^i) \)th node has a pointer to \( 2^i \) ahead
  - 50% nodes belong to 2 pointer lists,
  - 25% nodes belong to 3 pointer lists,
  - 12.5% nodes belong to 4 pointer lists, ...
Generalized Costs

- Similar characteristics to binary search tree

- Additional memory usage: $N$

- Search time: $O(\log N)$
  - $\log_2 N$ (levels) $\times$ 2 (comparisons/level)
Probabilistic Data Structure

- Insert & delete break the skip pointer patterns
  - building it again requires $O(N)$ time
- Instead of uniform distances, keep the ratios
  - 50% nodes 2 pointers,
  - 25% nodes 3 pointers,
  - 12.5% nodes 4 pointers, ...
- Probabilistically the same structure
  - no need of uniform patterns
  - but still operates in $\log N$ time with a high probability
Random Patterns

- **Uniform patterns**

- **Random patterns**
Fraction $p$

- Fraction $p$ of the nodes with level $i$ pointers also have level $i+1$ pointers
  - $p = 1/2$ (original discussion)
  - $p$ can be 1/4 for performance and memory usage

- Maximum number of pointers at level 1: $\log_{1/p} N$
  - Most implementation fixes the max number to 20
  - Number of levels: $\log_{1/p} N$
  - Search time: $O(\log N) = (\log_{1/p} N) \times 1/p$
Random Level

- Size of node
  - Number of lists a node belongs to

- Probabilistically determine the size of node $v$
  - Probability of $|v| = 1 : (1 - p)$
  - Probability of $|v| = 2 : p(1 - p)$
  - Probability of $|v| = 3 : p^2(1 - p)$
  - ...

```python
randomLevel()
lvl := 1
-- random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
  lvl := lvl + 1
return lvl
```
Insert

- Insert 17 (for example)
  - search position where 17 can be inserted
  - keep the update candidates from forward pointers
  - update pointers after determine the size with randomLevel()
Delete

- Similar to “insert”
  - search skip-list and find the element
  - keep the update candidates from forward pointers
  - delete the element and bypass the forward pointers

- Distribution of node size
  - size of node is randomly given
  - overall distribution will not change much
Worst Case

- If all nodes become the same size, it’s a normal list
- Insert & delete will change the distribution of node size, but the size to be inserted or deleted is also random
- Very low probability to make the list to be a normal list

Very low probability: search time becomes 3 times $\log N$
Memory Usage

- **List**
  - 1 extra space per node: $N$

- **Binary search tree**
  - 2 extra space per node: $2N$

- **Skip list**
  - Average size of node: $1/(1-p)$
  - $1/(1-p)$ extra spaces per node: $N/(1-p)$
  - Avg size: $(1-p) + 2p(1-p) + 3p^2(1-p) \ldots = 1/(1-p)$
Memory Usage (cont’d)

- $p$ determines the total amount of memory usage
- smaller $p$ results in less memory usage
- still, search in $\log N$ time even with a small memory
Skip List Usage

- Useful in parallel computing
  - concurrent insert in different parts
  - no need of global rebalancing
- Resource discovery in ad-hoc wireless network
  - Skip Graphs - robust to the loss of any single node
- Sometimes, skip list performs worse than B-tree
  - due to memory locality, and
  - more space requirement (from fragmentation)
Skip List Adoption

- **Redis (BSD)**
  - open source persistent key/value store for Posix systems

- **Qmap (GPL, LGPL)**
  - template class of Qt which provides a dictionary type

- **Skipdb (BSD)**
  - open source database format using ordered key/value pairs

- **ConcurrentSkipListSet, ConcurrentSkipListMap in Java 1.6**
  - concurrent version of Set and Map using skip list