Topics

Red-Black tree
  – Insert
  – Delete

B-Tree
  – Motivation
  – Search
  – Insert
  – Delete

Based on:
[Whitney: CS660 Combinatorial Algorithms, San Diego State University, 1996]
[Frederic Maire: An Introduction to Btrees, Queensland University of Technology, 1998]
Red-Black tree

Approximately balanced BST

\[ h_{RB} \leq 2 \times h_{BST} \] (height \( \leq 2 \) x height of a balanced tree)

Additional bit for COLOR = \{red \mid black\}

nil (non-existent child) = pointer to nil node

![Diagram of Red-Black tree with node labels and arrows indicating direction from leaf to inner node.](image)
Red-Black tree

A binary search tree is a red-black tree if:

1. Every node is either red or black.
2. Every leaf (nil) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.
5. Root is black.

Black-height $bh(x)$ of a node $x$ is the number of black nodes on any path from $x$ to a leaf, not counting $x$.
Black-height $bh(x)$ of a node $x$ is the number of black nodes on any path from $x$ to a leaf, not counting $x$. 

black height $bh(x)$

black height $bh(T) = 3$

height $h(T) = 6$
Binary Search Tree -> RB Tree

black height \( bh(T) = 4 \)

\( h(T) = 4 \)
Binary Search Tree -> RB Tree

black height $bh(T) = 3$

height $h(T) = 4$
Binary Search Tree -> RB Tree

black height $bh(T) = 3$

$h(T) = 4$
Binary Search Tree -> RB Tree

black height bh(T) = 2
h(T) = 4
Red-Black tree

Black-height $bh(x)$ of a node $x$
- is the number of black nodes on any path from $x$ to a leaf, not counting $x$
- is equal for all paths from $x$ to a leaf
- For given $h$ is $bh(x)$ in the range from $h/2$ to $h$
  - if $\frac{1}{2}$ of nodes red $\Rightarrow bh(x) \approx \frac{1}{2} h(x)$, $h(x) \approx 2 \lg(n+1)$
  - if all nodes black $\Rightarrow bh(x) = h(x) = \lg(n+1)$

Height $h(x)$ of a RB-tree rooted in node $x$
- is at maximum twice of the optimal height of a balanced tree
- $h \leq 2\lg(n+1)$ $\Rightarrow h \in \Theta(\lg(n))$
RB-tree height proof \[\text{[Cormen, p.264]}\]

A red-black tree with \(n\) internal nodes has height \(h\) at most \(2\log(n+1)\)

Proof 1. Show that subtree starting at \(x\) contains at least \(2^{bh(x)} - 1\) internal nodes.

By induction on height of \(x\):

I. If \(x\) is a leaf, then \(bh(x) = 0\), \(2^{bh(x)} - 1 = 0\) internal nodes \hspace{1cm} //… nil node

II. Consider \(x\) with height \(h\) and two children (with height at most \(h - 1\))

– \(x\)'s children black-height is either \(bh(x) - 1\) or \(bh(x)\) \hspace{1cm} // \(x\) is black or red

– Ind. hypothesis: \(x\)'s children subtree has at least \(2^{bh(x)} - 1\) internal nodes

– So subtree rooted at \(x\) contains at least \((2^{bh(x)} - 1) + (2^{bh(x)} - 1) + 1 = 2^{bh(x)} - 1\) internal nodes => proved

Proof 2. Let \(h = \text{height of the tree rooted at } x\)

– min \(\frac{1}{2}\) nodes are black on any path to leaf \hspace{1cm} => \(bh(x) \geq h / 2\)

– Thus, \(n \geq 2^{h/2} - 1 \iff n + 1 \geq 2^{h/2} \iff \log(n+1) \geq h / 2\)

– \(h \leq 2\log(n+1)\)
RB-tree Search

Search is performed as in simple BST, node colors do not influence the search.

Search in R-B tree with N nodes takes
1. In general -- at most $2\times\log_2(N+1)$ key comparisons.
2. In best case when keys are generated randomly and uniformly -- cca $1.002\times\log_2(N)$ key comparisons,
   very close to the theoretical minimum.
Inserting in Red-Black Tree

Color new node $x$ Red
Insert it as in the standard BST

If parent $p$ is Black, stop. Tree is a Red-Black tree.
If parent $p$ is Red (3+3 cases)...

resp.

While $x$ is not root and parent is Red
  if $x$’s uncle is Red then case 1  // propagate red up
  else  { if $x$ is Right child then case 2  // double rotation
    case 3 }

Color root Black
Inserting in Red-Black Tree

If parent is Black, stop. Tree is a Red-Black tree.
Inserting in Red-Black Tree

- x’s parent is Red
- x's uncle y is Red
- x is a Left child

Case 1a

Loop: x = x.p.p

bh(x) increased by one

x is node of interest

x's uncle is Red
Inserting in Red-Black Tree

- x’s parent is Red
- x's uncle y is Red
- x is a Right child

Loop: x = x.p.p

Case 1b

Recolor

bh(x) increased by one
Inserting in Red-Black Tree

x’s parent is Red

x's uncle y is Black

x is a Right child

Case 2

Lrot(x.p)

transform to Case 3

x is a Right child

x's uncle is Black
Inserting in Red-Black Tree

x’s parent is Red
x's uncle y is Black
x is a Left child

Terminal case, tree is a Red-Black tree

Case 3

x is a Left child
x's uncle is Black
Inserting in Red-Black Tree

Cases Right from the grandparent are symmetric
\begin{algorithm}
\caption{RB-INSERT($T, x$)}
1 \textbf{Tree-Insert}($T, x$)
2 \hspace{1em} \texttt{color}[x] \leftarrow \texttt{RED}
3 \textbf{while} $x \neq \text{root}[T]$ and \texttt{color}[p[x]] = \texttt{RED}
4 \hspace{1em} \textbf{do if} $p[x] = \text{left}[p[p[x]]]$
5 \hspace{2em} \textbf{then} $y \leftarrow \text{right}[p[p[x]]]$ \hspace{1em} \text{Red uncle $y$ -> recolor up}
6 \hspace*{2.5em} \textbf{if} \texttt{color}[y] = \texttt{RED}
7 \hspace*{3.5em} \textbf{then} \texttt{color}[p[x]] \leftarrow \texttt{BLACK} \hspace{1em} \triangleright \text{Case 1}
8 \hspace*{3.5em} \hspace{1em} \texttt{color}[y] \leftarrow \texttt{BLACK} \hspace{1em} \triangleright \text{Case 1}
9 \hspace*{3.5em} \texttt{color}[p[p[x]]] \leftarrow \texttt{RED} \hspace{1em} \triangleright \text{Case 1}
10 \hspace*{3.5em} \hspace{1em} x \leftarrow p[p[x]] \hspace{1em} \triangleright \text{Case 1}
11 \hspace{1em} \textbf{else if} $x = \text{right}[p[x]]$
12 \hspace{2em} \textbf{then} $x \leftarrow p[x]$ \hspace{1em} \triangleright \text{Case 2}
13 \hspace{2.5em} \texttt{LEFT-ROTATE}($T, x$) \hspace{1em} \triangleright \text{Case 2}
14 \hspace{1em} \texttt{color}[p[x]] \leftarrow \texttt{BLACK} \hspace{1em} \triangleright \text{Case 3}
15 \hspace{1em} \texttt{color}[p[p[x]]] \leftarrow \texttt{RED} \hspace{1em} \triangleright \text{Case 3}
16 \hspace{1em} \texttt{RIGHT-ROTATE}($T, p[p[x]]$) \hspace{1em} \triangleright \text{Case 3}
17 \hspace{1em} \textbf{else} (same as \textbf{then} clause
18 \hspace{2em} \text{with “right” and “left” exchanged})
19 \hspace{1em} \texttt{color}[\text{root}[T]] \leftarrow \texttt{BLACK}
\end{algorithm}

\texttt{p}[x] = \text{parent of } x
\texttt{left}[x] = \text{left child of } x
\texttt{y} = \text{uncle of } x

\text{[Cormen90]}
Inserting in Red-Black Tree

Insertion in $\Theta(\log(n))$ time
Requires at most two rotations
Deleting in Red-Black Tree

Find node to delete
Delete node as in a regular BST
Node y to be physically deleted will have at most one child x!!!

If we delete a Red node, tree still is a Red-Black tree, stop
Assume we delete a black node

Let x be the child of deleted (black) node y
If x is red, color it black and stop

while(x is not root) AND ( x is black)
    move x with virtual black mark through the tree
    (If x is black, mark it virtually double black A)
//note that the whole x 's subtree lost 1 unit of black height
while(\(x\) is not root) AND ( \(x\) is black) {
  // move \(x\) with virtual black mark \(A\) through the tree
  // just recolor or rotate other subtree up (decrease bh in R subtree)
  if (sibling is red)
    -> Case 1: Rotate right subtree up, color sibling black, and continue in left subtree with the new sibling
  if (sibling is black with both black children)
    -> Case 2: Color sibling red and go up
  else // black sibling with one or two red children
    if(red left child) -> Case 3: rotate to surface
    Case 4: Rotate right subtree up
}
Deleting in R-B Tree - Case 1

- \( x \) is the child of the physically deleted black node => double black
- \( x \)'s sibling \( w \) is red
  (\( x \)'s parent must be black)

- \( x \) stays at the same black height
  [Possibly transforms to case 2a and terminates – depends on 3,4]
Deleting in R-B Tree - Case 2a

- x’s sibling w is black
- x’s parent is red
- x’s sibling left child is black
- x’s sibling right child is black

Case 2a

Terminal case, tree is Red-Black tree

Note that A's subtree had less by 1 black height than D's subtree

STOP
Deleting in R-B Tree - Case 2b

- x’s sibling w is black
- x’s parent is black
- x’s sibling left child is black
- x’s sibling right child is black

Decreases x black height by one

Note that A’s subtree had less by 1 black height than D’s subtree

continue with new x
Deleting in R-B Tree - Case 3

- x’s sibling w is black
- x’s parent is either
- x’s sibling left child is red // impossible to color w red
- x’s sibling right child is black

Case 3

Transform to case 4
- x stays at same black height
Deleting in R-B Tree - Case 4

- x’s sibling w is black
- x’s parent is either
- x’s sibling left child is either
- x’s sibling right child is red

// impossible to color w red

Case 4

Terminal case, tree is Red-Black tree
(D inherits the color of B)
Deleting in Red-Black Tree

Notation similar to AVL

\( z = \text{logically removed} \)
\( y = \text{physically removed} \)
\( x = y \text{'s only child} \)

\[
\text{RB-DELETE}(T, z)
\]

1. \( \text{if } \text{left}[z] = \text{nil}[T] \text{ or } \text{right}[z] = \text{nil}[T] \)
2. \( \text{then } y \leftarrow z \)
3. \( \text{else } y \leftarrow \text{Tree-Successor}(z) \)
4. \( \text{if } \text{left}[y] \neq \text{nil}[T] \)
5. \( \text{then } x \leftarrow \text{left}[y] \)
6. \( \text{else } x \leftarrow \text{right}[y] \)
7. \( p[x] \leftarrow p[y] \)
8. \( \text{if } p[y] = \text{nil}[T] \)
9. \( \text{then } \text{root}[T] \leftarrow x \)
10. \( \text{else if } y = \text{left}[p[y]] \)
11. \( \text{then } \text{left}[p[y]] \leftarrow x \)
12. \( \text{else } \text{right}[p[y]] \leftarrow x \)
13. \( \text{if } y \neq z \)
14. \( \text{then } \text{key}[z] \leftarrow \text{key}[y] \)
15. \( \quad \triangleright \text{If } y \text{ has other fields, copy them, too.} \)
16. \( \text{if } \text{color}[y] = \text{BLACK} \)
17. \( \text{then } \text{RB-DELETE-FIXUP}(T, x) \)
18. \( \text{return } y \)

[Cormen90]
RB-DELETE-FIXUP(T, x)

1. while \( x \neq \text{root}[T] \) and \( \text{color}[x] = \text{BLACK} \)
2. do if \( x = \text{left}[p[x]] \)
3. then \( w \leftarrow \text{right}[p[x]] \)
4. 
5. if \( \text{color}[w] = \text{RED} \)
6. then \( \text{color}[w] \leftarrow \text{BLACK} \) \( \triangleright \) Case 1
7. \( \text{color}[p[x]] \leftarrow \text{RED} \) \( \triangleright \) Case 1
8. \( \text{LEFT-ROTATE}(T, p[x]) \) \( \triangleright \) Case 1
9. \( w \leftarrow \text{right}[p[x]] \) \( \triangleright \) Case 1
10. 
11. if \( \text{color}[^{\text{left}}][w] = \text{BLACK} \) and \( \text{color}[^{\text{right}}][w] = \text{BLACK} \)
12. then \( \text{color}[w] \leftarrow \text{RED} \) \( \triangleright \) Case 2
13. \( x \leftarrow p[x] \) \( \triangleright \) Case 2
14. 
15. else if \( \text{color}[^{\text{right}}][w] = \text{BLACK} \)
16. then \( \text{color}[^{\text{left}}][w] \leftarrow \text{BLACK} \) \( \triangleright \) Case 3
17. \( \text{color}[w] \leftarrow \text{RED} \) \( \triangleright \) Case 3
18. \( \text{RIGHT-ROTATE}(T, w) \) \( \triangleright \) Case 3
19. \( w \leftarrow \text{right}[p[x]] \) \( \triangleright \) Case 3
20. 
21. \( \text{color}[w] \leftarrow \text{color}[p[x]] \) \( \triangleright \) Case 4
22. \( \text{color}[p[x]] \leftarrow \text{BLACK} \) \( \triangleright \) Case 4
23. \( \text{color}[^{\text{right}}][w] \leftarrow \text{BLACK} \) \( \triangleright \) Case 4
24. \( \text{LEFT-ROTATE}(T, p[x]) \) \( \triangleright \) Case 4
25. \( x \leftarrow \text{root}[T] \) \( \triangleright \) Case 4

\( x \) = \textit{child} of removed node
\( p[x] \) = parent of \( x \)
\( w \) = sibling of \( x \)

[CMen90]
Deleting in R-B Tree

Delete time is $\Theta(\log(n))$
At most three rotations are done
Which BS tree is the best? [Pfaff 2004]

It is data dependent

• For random sequences
  => use *unsorted tree*, no waste time for rebalancing

• For mostly random ordering with occasional runs of sorted order
  => use *red-black trees*

• For insertions often in a sorted order and
  – later accesses tend to be random => AVL trees
  – later accesses are sequential or clustered => splay trees
    • self adjusting trees,
    • update each search by moving searched element to the root
B-tree as BST on disk
B-tree

Order 5 (5-ary tree)
Min degree $t = 3$

Based on [Cormen] and [Maire]
B-tree

1. Motivation
2. Multiway search tree
3. B-tree
4. Search
5. Insert
6. Delete
Motivation

- Large data do not fit into operational memory -> disk
- Time for disk access is limited by HW
  (Disk access = Disk-Read, Disk-Write)
- Disk access is MUCH slower compared to instruction
  – 1 disk access ~ 13 000 000 instructions!!!!
  – Number of disk accesses dominates the computational time
B-tree

Motivation

Disk access = Disk-Read, Disk-Write
  – Disk divided into blocks
    (512, 2048, 4096, 8192 bytes)
  – Whole block transferred

  – Design a *multiway search tree*
  – Each node fits to one disk block
B-tree

Multiway search tree

= a generalization of Binary search tree \((m=2)\)

Each node has at most \(m\) children \((m>2)\)

Internal node with \(n\) keys has \(n+1\) successors, \(n < m\) (except root)

Leaf nodes with no successors

Tree is ordered

Keys in nodes separates the ranges in subtrees
**B-tree**

Multiway search tree – internal node

Keys in internal node separate the ranges of keys in subtrees

\[ k_1 < k_2 < \ldots < k_5 \]
B-tree

Multiway search tree – leaf node

Leaves have no subtrees and do not use pointers

Leaves have no pointers to subtrees

\[ k_1 < k_2 < \ldots < k_5 \]
A B-tree of order $m$ is an $m$-way search tree, such that

- All leaves have the same height (B-tree is balanced)
- All internal nodes are constrained to have
  - at least $m/2$ non-empty children and (precisely later)
  - at most $m$ non-empty children
- The root can have 0 or between 2 to $m$ children
  - 0 - leaf
  - $m$ - a full node
Different authors use different names

- **Order $m$ B-tree**
  - Maximal number of children
  - Maximal number of keys (No. of children - 1)
  - Minimal number of keys

- **Minimum degree $t$**
  - Minimal number of children [Cormen]
B-tree – problems with notation

Relation between minimal and maximal number of children also differs.
For minimal number $t$ of children
Maximal number $m$ of children is

- $m = 2t - 1$ simple B-tree, multiphase update strategy
- $m = 2t$ optimized B-tree, singlephase update strategy
B-tree of order $m=1000$ of height 2 contains
1 001 001 nodes (1+1000 + 1 000 000)
999 999 999 keys ~ one billion keys (1 miliarda klíčů)
B-tree node fields

\( n \) … number of keys \( k_i \) stored in the node \( n < m \).

Node with \( n = m-1 \) is a **full-node**

\( k_i \) … \( n \) keys, stored in non-decreasing order

\[ k_1 \leq k_2 \leq \ldots \leq k_n \]

**leaf** … boolean value, true for leaf, false for internal node

\( c_i \) … \( n+1=m \) pointers to successors (undefined for leaves)

Keys \( k_i \) separate the keys in subtree:

For \( \text{keys}_i \) in the subtree with root \( k_i \) holds

\[ \text{keys}_1 \leq k_1 \leq \text{keys}_2 \leq k_2 \leq \ldots \leq k_n \leq \text{keys}_{n+1} \]
B-tree

B-tree algorithms

• Search
• Insert
• Delete
B-tree search

Similar to BST tree search
Keys in nodes sequentially or binary search

Input: pointer to tree root and a key $k$
Output: an ordered pair $(y, i)$, node $y$ and index $i$
     such that $y.k[i] = k$
     or NIL, if $k$ not found
Search 17: 17 not found => return NIL

Search 18: 18 found => return (x, 3)
B-tree search

B-treeSearch(x,k)

i ← 1

while i ≤ x.n and k > x.k[i]  // sequential search
  do i ← i+1

if i ≤ x.n and k = x.k[i]
  return (x, i)  // pair: node & index

if x.leaf
  then return NIL

else
  Disk-Read(x.c[i])  // tree traversal
  return B-treeSearch(x.c[i],k)
B-tree search

B-treeSearch complexity

Number of disk pages read is

\[ O(h) = O(\log_m n) \]

Where \( h \) is tree *height* and

- \( m \) is the tree order
- \( n \) is number of tree nodes

Since num. of keys \( x.n < m \), the while loop takes \( O(m) \) and

total time is \( O(m \log_m n) \)

Using tree order \( m \)
B-tree search complexity

Using minimum degree $t$

Number of disk pages read is

$O(h) = O(\log_t n)$

Where $h$ is tree height and

$t$ is the minimum degree of B-tree

$n$ is number of tree nodes

Since num. of keys $x.n < 2t$, the while loop takes $O(t)$

and

total time is $O(t \log_t n)$
B-tree update strategies

Two principal strategies

1. Multiphase strategy
   “solve the problem, when appears”  \[ m=2t-1 \text{ children} \]

2. Single phase strategy [Cormen]
   “avoid the future problems”  \[ m =2t \text{ children} \]

Actions:
- Split full nodes
- Merge nodes with less than minimum entries
B-tree insert - 1. Multiphase strategy

Insert to a **non-full** node

Insert 17

```
13
9 12
14 16 18
```

```
13
9 12
14 16 17 18
```
B-tree insert - 1. Multiphase strategy

Insert to a full node

Insert 25

13

9 12

14 16 17 18

25

median

Node split

Propagate median up

1. Multiphase strategy
“solve the problem, when appears”
B-tree insert - 1. Multiphase strategy

Insert \((x, T)\) - pseudocode

Find the leaf for \(x\)
If not full, insert \(x\) and stop
while (current_node full) \(\text{(node overflow)}\)
  find median (in keys in the node after insertion of \(x\))
  split node into two
  promote median up as new \(x\)
  current_node = parent of current_node or new root
Insert \(x\) and stop

x...key, T...tree

Top down phase

Bottom-up phase
B-tree insert - 2. Singlephase strategy

**Principle: “avoid the future problems”**

- Split the full node with $2t-1$ keys when enter
- It creates space for future medians from the children
- No need to go bottom-up

**Splitting of**
- Root $\Rightarrow$ tree grows by one
- Inner node or leaf $\Rightarrow$ parent gets median key

Top down phase only
B-tree insert - 2. Singlephase strategy

Insert to a non-full node

Insert B

\[ m = 2t = 6 \text{ children} \]

\[ m-1 \text{ keys} = \text{odd max number} \]
B-tree insert - 2. Singlephase strategy

Splitting a passed full node and insert to a not full node

Insert Q

1 new node
B-tree insert - 2. Singlephase strategy

Splitting a passed **full root** and insert to a **not full** node

Insert L

Split GMPTX
Tree grows by 1

Insert L to JK

2 new nodes
B-tree insert - 2. Singlephase strategy

Insert F

Split ABCDE

Insert F to DE
B-tree insert - 2. Single phase strategy

Insert \((x, T)\) - pseudocode

While searching the leaf \(x\)

if (node full)
    find median (in keys in the full node only)
    split node into two
    insert median to parent (there is space)

Insert \(x\) and stop

Top down phase only

\(x\) ...key, \(T\)... tree
B-tree delete

Delete (x, btree) - principles

• Search for value to delete
• Entry is in leaf
  is simple to delete. Do it. Corrections of number of elements later...
• Entry is in Inner node
  – It serves as separator for two subtrees
  – swap it with predecessor(x) or successor(x)
  – and delete in leaf

Leaf in detail

if leaf had more than minimum number of entries
  delete x from the leaf and STOP
else
  redistribute the values to correct and delete x in leaf
  (may move the problem up to the parent,
  problem stops by root, as it has no minimum number of entries)
B-tree delete

Node has less than minimum entries

- Look to siblings left and right
- If one of them has more than minimum entries
  - Take some values from it
  - Find new median in the sequence:
    (sibling values – separator- node values)
  - Make new median a separator (store in parent)
- Both siblings are on minimum
  - Collapse node – separator – sibling to one node
  - Remove separator from parent
  - Go up to parent and correct
Delete (x, btree) - pseudocode

if (x to be removed is not in a leaf)
    swap it with successor(x)

currentNode = leaf

while (currentNode underflow)
    try to redistribute entries from an immediate sibling into currentNode via its parent
    if (impossible) then merge currentNode with a sibling and one entry from the parent

currentNode = parent of currentNode

Multipass strategy only
Maximum height of B-tree

\[ h \leq \log_{\left\lfloor \frac{m}{2} \right\rfloor} \left( \frac{n+1}{2} \right) \]

Gives the upper bound to number of disk accesses

See [Cormen] for details
References


Red Black Tree

[Whitney]: CS660 Combinatorial Algorithms, San Diego State University, 1996], RedBlack, B-trees
http://www.eli.sdsu.edu/courses/fall96/cs660/notes/redBlack/redBlack.html#RT

http://en.wikipedia.org/wiki/B-tree

[Jones] Jeremy Jones: B-Tree animation - java applet

Splay tree


Tree comparison