Principles of B-trees

Nodes and binary search

A node $u$ has
size $size(u)$,
keys $k_1, \ldots, k_{size(u)-1}$
children $c_1, \ldots, c_{size(u)}$.

Binary search property: for $i = 1, \ldots, size(u) - 1$
keys in the subtree of $c_i$ are smaller than $k_i$
keys in the subtree of $c_{i+1}$ are larger than $k_i$

Size limitation, parameter $t$

For root $r$, $2 \leq size(r) \leq 2t$
For every other node $u$, $t \leq size(r) \leq 2t$

Variant: upper bound can be $2t - 1$.

Shape limitation, or balance

All leaves have the same depth, i.e. the same distance from the root.
Nice properties:

★ The number of levels if there are $n$ keys: about $\log_t n = \frac{1}{\ln t} \ln n$
   (about: can be smaller, or larger by 1)

☆ The time needed to perform binary search: about $\log_2 t$ steps per level,
   hence, about $\log_2 n$ steps (but we may add 1 for each level, so a bit more).

✩ Insertion and deletion require to visit and restructure at most 2 nodes on each level.

☆ With very small $t$, this gives $O(\log n)$ time for insertion and deletion. This
   is used for 2-3-4 trees ($t = 2$), 2-3 trees ($t = 2$, variation of the rules), red-
   black trees (groups of 1-2-3 nodes obey the rules of 2-3-4 trees).

★ In disk memory (or solid state disk etc.) we use large $t$, so each node occu-
   pies up to a full page (upper bound), and at least half a page (lower bound).
**Insertion principles.**

1. During insertion, a node can increase its size by 1. A node with size above the upper bound is unstable, and splits into two parts, as equal in size as possible. The pointer to such a node is replaced with a trio: separating value and two pointers.

2. The parent of a split child increases in size: it has a trio where one pointer was before. If this makes it unstable, it splits.

3. If the root splits, the trio that replaces the root pointer becomes the root node. Pointer to the trio becomes the new root pointer.

**Remark.** Unstable nodes are good for the design and analysis, but they are not implemented. Basically, we describe how to avoid using unstable nodes.

**Deletion principles.**

1. When we delete in a node, a sibling is a designated helper.

2. During deletion, the size can go down by 1. In the extreme case, the size drops to 1, and the number of keys to 0. This basically means that there is nothing there, and the pointer to its only child actually belongs to the grand-parent of the child, or, in the case of the root. the only child of the root is the new root.

3. If the node becomes unstable, we first try to get a child from the designated helper. This changes the key that separates those two siblings.

4. If getting the child is impossible, the two siblings have sizes $t$ and $t - 1$, hence $2t - 1$ children, and they are legally fused. The separating value (and one pointer) is removed from the parent.

**Extra principles for red-black trees**

1. Groups of 1-2-3 binary tree nodes obey the rules of 2-3-4 tree.

2. If a pointer in node $u$ points to a node in the same group, it is red. Otherwise it is black.

3. A group of 3 nodes must have Λ shape: two red pointer in the root node of the group.
B-trees and red-black trees: examples

Inserting $a$, $b$, $c$, $d$ to an initially empty set.

Worth to notice:

- after inserting $c$ to the red-black tree we obtain a group of three nodes of illegal shape, a **rotation** corrects this shape;
- after inserting $d$ to 2-3-4 tree the root node becomes unstable and splits, which creates a new root;
- in red-black tree the root group is similarly split, and this split is obtained by changing the colors of the pointers of the root node: when they are red, the root is in the same group with the rest of the node, when they are black, the root is a separate group, so the left and right subtrees are separate groups as well.
Inserting $e, f, g$ to the set obtained before.

Worth to notice:

after inserting $f$ we split a node in 2-3-4 tree and a group in red-black tree; this brings the separating key, $d$, to the parent node or the parent group. The latter means that the pointer to the node of $d$ becomes red.
Inserting $h, i$ to the set obtained before.
Inserting $j$ to the set obtained before.
Deleting a from the set obtained before, in 2-3-4 tree.

Empty node is unstable, the sibling has no spare children, so they fuse into one node:

Again we have an unstable node with one child only, its sibling has a spare child, so we transfer a child:
Deleting a from the set obtained before, in red-black tree. A group of nodes that is too small is empty, its place is indicated by doubly-black pointer, which we indicate on the diagram with a black rectangle. When this group fuses with its sibling group, its parent group is reduced to zero nodes and is now indicated by the doubly black pointer.

Now this group receives extra child, e, and extra (i.e. the only!) separating key, d, while the helping group drops in size to one node, h.
**Treaps — randomness gives balance without rotations**

Binary search trees built by insertions alone, without deletions, in random order have very good depth on the average. However, the insertions do not have to come in random order, and besides, we have deletions which are easy to implement but hard to analyze: how the average depth changes if you perform both insertions and deletions.

Luckily, a very simple trick allows to enforce the shape as if the the tree was built by insertions alone, in random order. When we insert a new element we give it a random priority from some sufficiently large range. We will assume that this priority is a random real number from the range $[0, 1]$. The structures that form the tree now have 4 fields: \{key, pr, left, right\}. So when we insert a new element $x$ we find $p\{x, p, NULL, NULL\}$. Then we connect this structure to the tree.

The rules of that tree are the following

**Binary search order of keys**

- if $s$ is a descendent of $t->left$ then $s->key < t->key$
- if $s$ is a descendent of $t->right$ then $s->key > t->key$

**Heap order of priorities**

- if $s$ is the parent of $t$ then $s->pr \leq t->pr$

The heap order enforces the shape as if the keys were inserted in the same order as their priorities, the least priority is on the root, and in a subtree, with content determined by the selection of the root and other ancestors, the smallest priority has uniformly random “owner” and this owner is at the root.

We will see that the code for enforcing that shape is very simple.
**Insertion in treaps — splitting**

In some applications an ordered dictionary/tree can be split into two according to some key value \( x \): one new dictionary will have keys smaller than \( x \), the other will have the larger keys. Split is also useful during insertion.

We want to insert a new key \( x \) with priority \( p \) into tree \( t \).

\[
\begin{align*}
s &= \{x, p, ?, ?\}; & // \text{magic allocates a new structure} \\
T &= & & // \text{locate the subtree to modify} \\
\text{while} \ (t = \ast T, t \neq NULL \&\& t->pr < p) \\
& \quad \text{if} \ (t->key < x) \\
& \quad \quad \ast T = t->left; \\
& \quad \text{else} \\
& \quad \quad \ast T = t->right; \\
& \quad \text{if} \ (t == NULL) \{ & // \text{insertion at the bottom} \\
& \quad \quad \ast T = s; \\
& \quad \quad s->left = s->right = NULL; \\
& \quad \quad \text{return;}
\}
\end{align*}
\]

// replace node \( \ast T \) with \( s \), make children of \( s \) by splitting

// old \( \ast T \) into \( \ast L, \ast R \), with keys smaller/larger than \( x \)

\[
\begin{align*}
L &= & & (s->left); \\
R &= & & (s->right); \\
\ast T &= & & s; \\
\text{while} \ (t != NULL) \\
& \quad \text{if} \ (t->key < x) \\
& \quad \quad \ast L = t, \\
& \quad \quad L &= & & (t->left), \\
& \quad \quad t &= & & t->left; \\
& \quad \text{else} \\
& \quad \quad \ast R = t, \\
& \quad \quad R &= & & (t->right), \\
& \quad \quad t &= & & t->right; \\
\ast L &= & & \ast R = NULL;
\end{align*}
\]
Example how this works:
Deletion in treaps — melding

In some applications two ordered dictionaries/trees such that all keys from the first dictionary precede keys from the second can be combined into one. This operation is called meld.

We want to delete key $x$. We locate the structure with $x$ and then we replace the pointer to that structure with the pointer to the meld of its children.

```c
void remove(x, T) {
    while (t = *T) { // this breaks when t == NULL
        if (t->key < x))
            *T = &(t->right);
        else if (t->key > x))
            *T = &(t->left);
        else {
            *T = meld(t->left, t->right);
            free t;
            return;
        }
    }
}
```

Meld is easier to implement recursively:

```c
tree meld(s, t) // assume keys in x smaller than in t
{
    if (s == NULL)
        return t;
    else if (t == NULL)
        return s;
    else if (s->pr > t->pr)
        return s->right = meld(s->right, t), s;
    else
        return t->left = meld(s, t->left), t;
}
```
Priority queue and heaps

Ordered dictionary ordinarily uses pointers for every key it stores and uses trees that are not “perfectly balanced”. We can do better if we use it in a restricted way. In a typical application, we store objects with priorities and in some applications, the priorities can change over time.

Set Operations:

Initialize(Q), Insert(x,p,Q), DeleteMin(Q)

Other updates:

IncreasePriority(i,p,Q), DecreasePriority(i,p,Q).

In the last operation, we decrease the priority of an object at position i. We can know this position if we use cross-referencing.

We implement priority queues using heaps. Heap is a tree in which parent node has smaller (or equal) priority to the node of the parent. This is called heap order or partial order. Because we do not have other limitations, we are quite flexible how we place the objects in the tree.

The simplest heap with m objects is an array sector (H, 1, m + 1) where position 1 is the root, and children of position i are: 2i and 2i + 1. Conversely, parent of i is i/2 (rounded down).

Example

Example of a heap (H[i] has number i underneath):
Restoring heap property

We will define all operations in a heap using two restore operations. Assume that \( <H, 0, m> \) has heap property and that we want to change entry \( H[i] \) to \( x \). To restore the heap property we need to rearrange the entries. We have two cases.

The easier case is when \( x < H[i] \). We will call this task \( \text{rad}(H, i, x) \), for restore after decreasing. We can define \( \text{rad}(H, i, x) \) recursively.

\[
\text{rai}(H, i, x) = \begin{cases} 
\text{if } (i == 1 \text{ or } (p = i/2, x \geq H[p])) \{ \\
H[i] = x; \\
\text{return;}
\} \\
H[i] = H[p]; \\
\text{rai}(H, p, x);
\end{cases}
\]

This formulation is recursive so we can have an inductive proof of correctness. Because the recursive call is the last statement in this function, we can eliminate this call using tail recursion principle.

\[
\text{while } (i > 1 \&\& (p = i/2, x < H[p])) \\
H[i] = H[p], \\
i = p; \\
H[i] = x;
\]

The worst case running time is \( c \log_2 i \).
Example: \((H, 1, 13)\) forms a heap shown before and we execute \(\text{rad}(H, 9, 2)\). Location to be changed by \(\text{rad}\) will be colored blue.
The more complicated case is when $x > H[i]$. We will call this task $\text{rai}(H, i, m, x)$, for restore after increasing; here $m$ is the number of keys stored in the heap. We can define $\text{rai}(H, i, m, x)$ recursively.

$\text{rai}(H, i, m, x)$
\begin{verbatim}
{ g = 2*i; // start computing the good child of i
  if (g > m) { // no children \rightarrow finish
    H[i] = x;
    return;
  }
  if (g+1 \leq m && H[g] > H[g+1])
    g++; // good child has the smaller priority
  if (x \leq H[g]) {
    H[i] = x;
    return;
  }
  H[i] = H[g];
  rai(H, g, m, x);
}
\end{verbatim}

Again, we can eliminate the tail recursion:

while ($g = 2*i, g < m$) {
  if ($g+1 \leq m && H[g] > H[g+1]$)
    g++;
  if ($x \leq H[g]$)
    break;
  H[i] = H[g];
  i = g;
}

$H[i] = x$;

The worst case running time is $c \log_2 m/i$. 
Example: \((H, 1, 13)\) we execute \(\text{rai}(H, 0, 12, 39)\). Location to be changed by \(\text{rai}\) are blue, the good child is yellow.
Heapify, impose the heap property

We can impose the heap property by pretending to restore it.

Version 1: pretend that all entries are $\infty$ and that you decrease them to their actual values. If you do it in order $A[0], A[1], \ldots$ then before you process $A[i]$ the fragment $< A, i, m >$ remains unchanged.

$$\text{for } (i = 2; i \leq m; i++)$$
$$\text{rad}(A, i, A[i]);$$

The worst case running time is

$$c \sum_{i=1}^{m-1} \log_2 i \approx cm(\log_2 m - \log_2 e) = \Theta(m \log m).$$

Version 2: pretend that all entries are $-\infty$ and that you increase them to their actual values. If you do it in order $A[m-1], A[m-2], \ldots$ then before you process $A[i]$ the fragment $< A, 0, i >$ remains unchanged.

$$\text{for } (i = m/2; i > 0; i--)$$
$$\text{rai}(A, i, m, A[i]);$$

The worst case running time is

$$c \sum_{i=1}^{m-1} \log_2 m/i \approx cm \log_2 m - cm(\log_2 m - \log_2 e) = cm \log_2 e = \Theta(m).$$

Version 2 is much better!
Heap sort

We want to sort \((A, 0, n)\). We create heap \((H, 1, n + 1)\) and then we remove the minimum from the heap and place it at the beginning of the resulting sequence. At any stage, \((H, 1, m + 1) = (A, 0, m)\) contains the heap and \((A, m, n)\) stores the resulting sequence.

\[ H = A-1; \]
\[ \text{for } (i = n/2; i > 0; i--) \]
\[ \text{rad}(H,i,n,H[i]); \]
\[ \text{for } (m = n; m > 1; m--) \]
\[ \text{temp} = H[m], \]
\[ H[m] = H[1], \]
\[ \text{rai}(H,1,m-1,temp); \]

This sorts in the reverse order. To sort in increasing order we need to modify the heap so it has maximum at the root rather than the minimum.
**Priority queue**

Abstract data type:

possible states: sets of elements of some comparable type;

operations:

MAKEEMPTY, initialize empty set,
INSERT,
DELETE_MIN, remove and return the minimum.

Implementation with bounded capacity $n$, using array $H[n]$ and $m$:
MAKEEMPTY()
{ $m = 0$; }

INSERT($x$);
{ $m++$;
 rad($H,m,m,x$)
 }

DELETE_MIN()
{ $x = H[1]$;
 m--;
 rai($H,1,m,H[m+1]$);
 return $x$;
 }