2-3 Tree

Definition  A 2-3 tree is a search tree that

- may have 2-nodes and 3-nodes
- height-balanced (all leaves are on the same level)

A 2-3 tree is constructed by successive insertions of keys given, with a new key always inserted into a leaf of the tree. If the leaf is a 3-node, it’s split into two with the middle key promoted to the parent.
2-3 tree construction – an example

Construct a 2-3 tree the list 9, 5, 8, 3, 2, 4, 7

```
9
  |   |
5, 9
  |   |   |   |
5, 8, 9
  |   |   |   |
8
  |   |   |   |
2, 3, 5
  |   |   |   |
9

3, 8
  |   |   |   |
2
  |   |   |
4, 5, 7
  |   |   |
9

3, 8
  |   |   |   |
3, 5, 8
  |   |   |   |
2
  |   |   |
4, 7
  |   |   |
9
```

```
8
  |   |   |   |
5
  |   |   |
9

3, 5, 8
  |   |   |   |
2
  |   |   |
4
  |   |   |
7
  |   |   |
9

3
  |   |   |   |
2
  |   |   |
4
  |   |   |
7
  |   |   |
9
```
Analysis of 2-3 trees

- \( \log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1 \)

- Search, insertion, and deletion are in \( \Theta(\log n) \)

- The idea of 2-3 tree can be generalized by allowing more keys per node
  - 2-3-4 trees
  - B-trees
What is the “best” binary tree?
Heaps and Heapsort

**Definition**  A heap is a binary tree with keys at its nodes (one key per node) such that:

- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing
- The key at each node is $\geq$ keys at its children
Illustration of the heap's definition

Illustration of the heap’s definition

Note: Heap’s elements are ordered top down (along any path down from its root), but they are not ordered left to right.
Some Important Properties of a Heap

- Given \( n \), there exists a unique binary tree with \( n \) nodes that is essentially complete, with \( h = \lfloor \log_2 n \rfloor \)

- The root contains the largest key

- The subtree rooted at any node of a heap is also a heap

- A heap can be represented as an array
Heap’s Array Representation

Store heap’s elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order

Example:

- Left child of node $j$ is at $2j$
- Right child of node $j$ is at $2j+1$
- Parent of node $j$ is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations
Heap Construction (bottom-up)

Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn’t satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node
Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8
Construct a heap for the list 2, 9, 7, 6, 5, 8
Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8
Pseudopodia of bottom-up heap construction

**Algorithm** HeapBottomUp(H[1..n])
// Constructs a heap from the elements of a given array
// by the bottom-up algorithm
// Input: An array H[1..n] of orderable items
// Output: A heap H[1..n]
for i ← [n/2] downto 1 do
    k ← i;  v ← H[k]
    heap ← false
    while not heap and 2 \* k ≤ n do
        j ← 2 \* k
        if j < n  // there are two children
            if H[j] < H[j + 1]  j ← j + 1
            if v ≥ H[j]
                heap ← true
            else H[k] ← H[j];  k ← j
        H[k] ← v
Heapsort

Stage 1: Construct a heap for a given list of \( n \) keys

Stage 2: Repeat operation of root removal \( n-1 \) times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds
Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

<table>
<thead>
<tr>
<th>Stage 1 (heap construction)</th>
<th>Stage 2 (root/max removal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2  9  7  6  5  8</td>
<td>9  6  8  2  5  7</td>
</tr>
<tr>
<td>2  9  8  6  5  7</td>
<td>7  6  8  2  5</td>
</tr>
<tr>
<td>2  9  8  6  5  7</td>
<td>8  6  7  2  5</td>
</tr>
<tr>
<td>9  2  8  6  5  7</td>
<td>5  6  7  2</td>
</tr>
<tr>
<td>9  6  8  2  5  7</td>
<td>7  6  5  2</td>
</tr>
</tbody>
</table>

Analysis of Heapsort

Stage 1: Build heap for a given list of $n$ keys

worst-case

$$C(n) = \sum_{i=0}^{h-1} 2(h-i) 2^i = 2(n - \log_2(n + 1)) \in \Theta(n)$$

Both worst-case and average-case efficiency: $\Theta(n \log n)$

In-place: yes

Stability: no (e.g., 1 1)
Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them.
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied.

Example: Insert key 10

Efficiency: $O(\log n)$
Horner’s Rule For Polynomial Evaluation

Given a polynomial of degree $n$

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

and a specific value of $x$, find the value of $p$ at that point.

Two brute-force algorithms:

1. $p \leftarrow 0$
2. $p \leftarrow a_0$; $\text{power} \leftarrow 1$
3. For $i \leftarrow n$ downto $0$
   4. $\text{power} \leftarrow 1$
   5. For $j \leftarrow 1$ to $i$
      6. $\text{power} \leftarrow \text{power} \ast x$
      7. $p \leftarrow p + a_i \ast \text{power}$
   8. $\text{power} \leftarrow \text{power} \ast x$
9. $p \leftarrow p + a_i \ast \text{power}$
10. Return $p$

11. $p \leftarrow a_0$; $\text{power} \leftarrow 1$
12. For $i \leftarrow 1$ to $n$
   13. $\text{power} \leftarrow \text{power} \ast x$
   14. $p \leftarrow p + a_i \ast \text{power}$
15. Return $p$
Horner’s Rule

Example: \( p(x) = 2x^4 - x^3 + 3x^2 + x - 5 = \)
\[
= x(2x^3 - x^2 + 3x + 1) - 5 = \\
= x(x(2x^2 - x + 3) + 1) - 5 = \\
= x(x(x(2x - 1) + 3) + 1) - 5
\]

Substitution into the last formula leads to a faster algorithm

Same sequence of computations are obtained by simply arranging the coefficient in a table and proceeding as follows:

coefficients 2 -1 3 1 -5

\[x=3\]
Horner’s Rule pseudocode

ALGORITHM Horner($P[0..n]$, $x$)

// Evaluates a polynomial at a given point by Horner’s rule
// Input: An array $P[0..n]$ of coefficients of a polynomial of degree $n$
//        (stored from the lowest to the highest) and a number $x$
// Output: The value of the polynomial at $x$

$p ← P[n]$

for $i ← n − 1$ downto 0 do

$p ← x \times p + P[i]$

return $p$

Efficiency of Horner’s Rule: # multiplications = # additions = $n$

Synthetic division of $p(x)$ by $(x-x_0)$
Example: Let $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$. Find $p(x):(x-3)$
Computing $a^n$ (revisited)

**Left-to-right binary exponentiation**

Initialize product accumulator by 1.

Scan $n$'s binary expansion from left to right and do the following:

If the current binary digit is 0, square the accumulator (S);
if the binary digit is 1, square the accumulator and multiply it by $a$ (SM).

Example: Compute $a^{13}$. Here, $n = 13 = 1101_2$

binary rep. of 13: \[1 \quad 1 \quad 0 \quad 1\]

<table>
<thead>
<tr>
<th>SM</th>
<th>SM</th>
<th>S</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

accumulator: \[1 \quad 1^2 \cdot a = a \quad a^2 \cdot a = a^3 \quad (a^3)^2 = a^6 \quad (a^6)^2 \cdot a = a^{13}\]

(computed left-to-right)

Efficiency: $b \leq M(n) \leq 2b$ where $b = \lfloor \log_2 n \rfloor + 1$
Computing $a^n$ (cont.)

**Right-to-left binary exponentiation**

Scan $n$’s binary expansion from right to left and compute $a^n$ as the product of terms $a^{2^i}$ corresponding to 1’s in this expansion.

**Example** Compute $a^{13}$ by the right-to-left binary exponentiation. Here, $n = 13 = 1101_2$.

\[
\begin{align*}
1 & & 1 & & 0 & & 1 & \\
& a^8 & a^4 & a^2 & a & : & a^{2^i} \text{ terms} \\
& a^8 & * & a^4 & * & a & : & \text{ product}
\end{align*}
\]

(computed right-to-left)

Efficiency: same as that of left-to-right binary exponentiation
Problem Reduction

This variation of transform-and-conquer solves a problem by transforming it into a different problem for which an algorithm is already available.

To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.
Examples of Solving Problems by Reduction

- computing $\text{lcm}(m, n)$ via computing $\text{gcd}(m, n)$
- counting number of paths of length $n$ in a graph by raising the graph’s adjacency matrix to the $n$-th power
- transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- linear programming
- reduction to graph problems (e.g., solving puzzles via state-space graphs)