Binary Search Trees

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

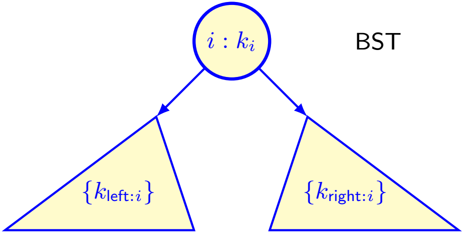
1. Properties of Binary Search Trees
2. Basic BST operations
3. The worst-case time complexity of BST operations
4. The average-case time complexity of BST operations
5. Self-balancing binary and multiway search trees
6. Self-balancing BSTs: AVL trees
7. Self-balancing BSTs: Red-black trees
8. Balanced B-trees for external search

Binary Search Tree: Left-Right Ordering of Keys

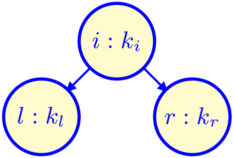
Left-to-right numerical ordering in a BST: for every node i,

- the values of all the keys $k_{\text{left}:i}$ in the left subtree are smaller than the key k_i in i and
- the values of all the keys $k_{\text{right}:i}$ in the right subtree are larger than the key k_i in i: $\{k_{\text{left}:i}\} \ni l < k_i < r \in \{k_{\text{right}:i}\}$

Compare to the **bottom-up** ordering in a *heap* where the key k_i of every parent node i is greater than or equal to the keys k_l and k_r in the left and right child node l and r, respectively: $k_i \geq k_l$ and $k_i \geq k_r$.

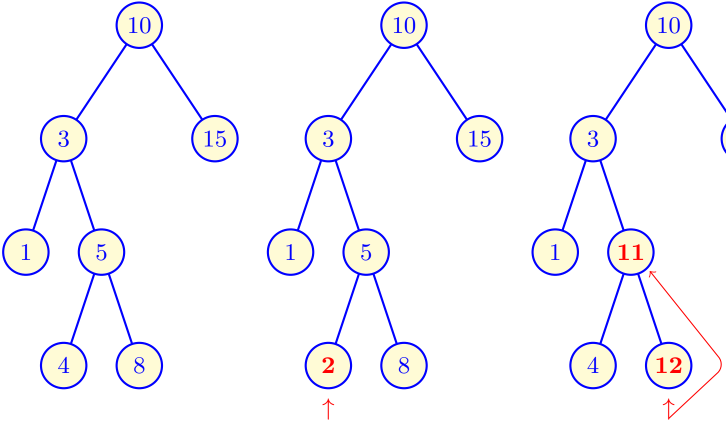


BST



Heap

Binary Search Tree: Left-Right Ordering of Keys



BST

Non-BST:
Key "2" cannot be in the right subtree of key "3".

Non-BST:
Keys "11" and "12" cannot be in the left subtree of key "10".

Basic BST Operations

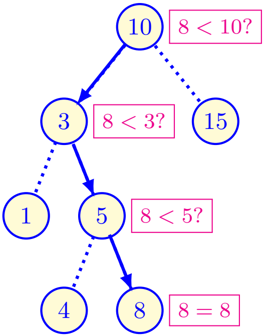
BST is an explicit *data structure* implementing the table ADT.

- BST are more complex than heaps: any node may be removed, not only a root or leaves.
- The only practical constraint: no duplicate keys (attach them all to a single node).

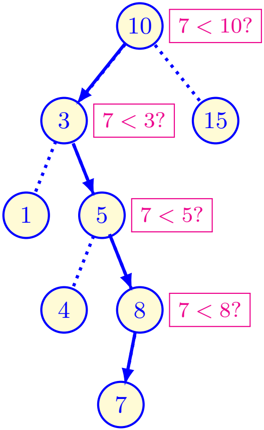
Basic operations:

- **find** a given search key or detect that it is absent in the BST.
- **insert** a node with a given key to the BST if it is not found.
- **findMin**: find the minimum key.
- **findMax**: find the maximum key.
- **remove** a node with a given key and restore the BST if necessary.

BST Operations Find / Insert a Node



Found node



Inserted node

find: a successful binary search.

insert: creating a new node at the point where an unsuccessful search stops.

BST Operations: FindMin / FindMax

Extremely simple: starting at the root, branch repeatedly left (**findMin**) or right (**findMax**) as long as a corresponding child exists.

- The root of the tree plays a role of the pivot in `quicksort` and `quickselect`.
- As in `quicksort`, the recursive traversal of the tree can sort the items:
  1. First visit the left subtree;
  2. Then visit the root, and
  3. Then visit the right subtree.

$O(\log n)$ average-case and $O(n)$ worst-case running time for find, insert, findMin, and findMax operations, as well as for selecting a single item (just as in `quickselect`).

BST Operation: Remove a Node

The most complex because the tree may be disconnected.

- Reattachment must retain the ordering condition.
- Reattachment should not needlessly increase the tree height.

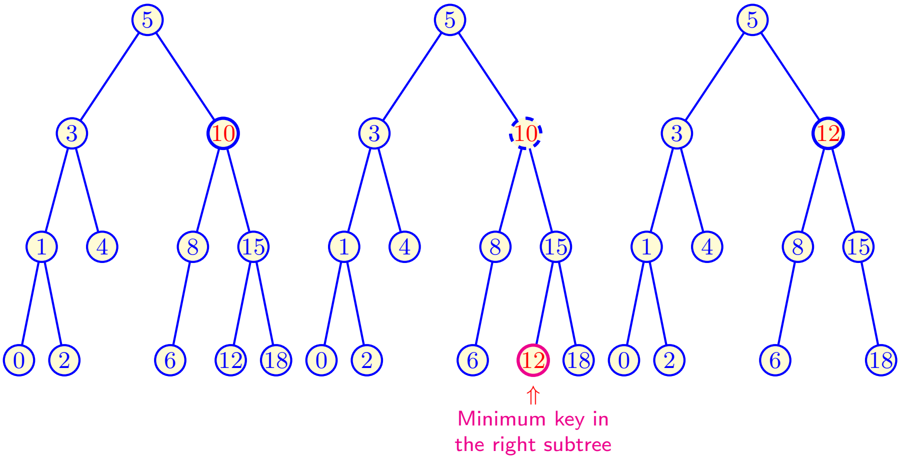
Standard method of removing a node i with c children:

c	Action
0	Simply remove the leaf i.
1	Remove the node i after linking its child to its parent node.
2	Swap the node i with the node j having the smallest key k_j in the right subtree of the node i. After swapping, remove the node i (as now it has at most one right child).

In spite of its asymmetry, this method cannot be really improved.

BST Operation: Remove a Node

Remove 10 ⇒ Replace 10 (swap with 12 and delete)



Analysing BST: The Worst-case Time Complexity

Lemma 3.11: The search, retrieval, update, insert, and remove operations in a BST all take time in $\mathrm{O}(h)$ in the worst case, where h is the height of the tree.

Proof: The running time $T(n)$ of these operations is proportional to the number of nodes ν visited.

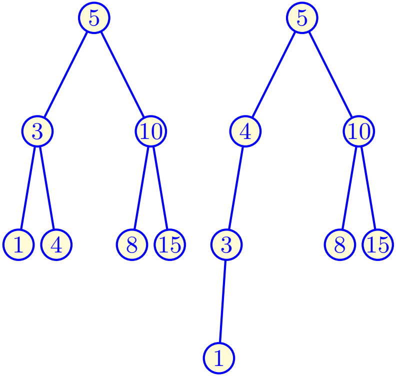
- Find / insert: $\nu = 1 + \langle$the depth of the node$\rangle$.
- Remove: $\langle$the depth $+$ at most the height of the node$\rangle$.
- In each case $T(n) = \mathrm{O}(h)$. □

For a well-balanced BST, $T(n) \in \mathrm{O}(\log n)$ (logarithmic time).

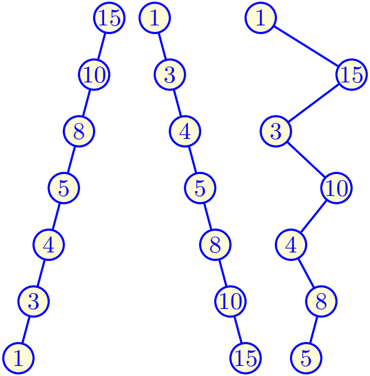
In the worst case $T(n) \in \Theta(n)$ (linear time) because insertions and deletions may heavily destroy the balance.

Analysing BST: The Worst-case Time Complexity

BSTs of height $h \approx \log n$



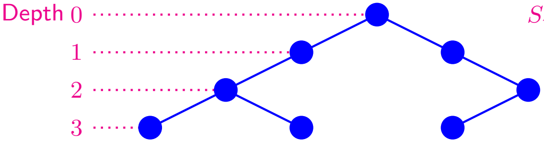
BSTs of height $h \approx n$



Analysing BST: The Average-case Time Complexity

More balanced trees are more frequent than unbalanced ones.

Definition 3.12: The *total internal path length*, $S_\tau(n)$, of a binary tree τ is the sum of the depths of all its nodes.



$$S_\tau(8) = 0 + \\ + 1 + 1 \\ + 2 + 2 \\ + 3 + 3 + 3 = \mathbf{15}$$

- Average complexity of a successful search in τ: the average node depth, $\frac{1}{n}S_\tau(n)$, e.g. $\frac{1}{8}S_\tau(8) = \frac{15}{8} = 1.875$ in this example.
- Average-case complexity of searching:
  - Averaging $S_\tau(n)$ for all the trees of size n, i.e. for all possible $n!$ insertion orders, occurring with equal probability, $\frac{1}{n!}$.

The $\Theta(\log n)$ Average-case BST Operations

Let $S(n)$ be the average of the total internal path length, $S_\tau(n)$, over all BST τ created from an empty tree by sequences of n random insertions, each sequence considered as equiprobable.

Lemma 3.13: The expected time for successful and unsuccessful search (update, retrieval, insertion, and deletion) in such BST is $\Theta(\log n)$.

Proof: It should be proven that $S(n) \in \Theta(n \log n)$.

- Obviously, $S(1) = 0$.
- Any n-node tree, $n > 1$, contains a left subtree with i nodes, a root at height 0, and a right subtree with $n - i - 1$ nodes; $0 \leq i \leq n - 1$.
- For a fixed i, $S(n) = (n - 1) + S(i) + S(n - i - 1)$, as the root adds 1 to the path length of each other node.

The $\Theta(\log n)$ Average-case BST Operations

Proof of Lemma 3.13 (continued):

- After summing these recurrences for $0 \leq i \leq n-1$ and averaging, just the same recurrence as for the average-case `quicksort` analysis is obtained:
$$S(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} S(i)$$
- Therefore, $S(n) \in \Theta(n \log n)$, and the expected depth of a node is $\frac{1}{n}S(n) \in \Theta(\log n)$.
- Thus, the average-case search, update, retrieval and insertion time is in $\Theta(\log n)$.
- It is possible to prove (but in a more complicate way) that the average-case deletion time is also in $\Theta(\log n)$. □

The BST allow for a special **balancing**, which prevents the tree height from growing too much, i.e. avoids the worst cases with linear time complexity $\Theta(n)$.

Self-balanced Search Trees

Balancing ensures that the total internal path lengths of the trees are close to the optimal value of $n \log n$.

- The average-case and the worst-case complexity of operations is $\mathrm{O}(\log n)$ due to the resulting balanced structure.
- But the insertion and removal operations take longer time on the average than for the standard binary search trees.

Balanced BST:

- AVL trees (1962: G. M. Adelson-Velskii and E. M. Landis).
- Red-black trees (1972: R. Bayer) – *"symmetric binary B-trees"*; the present name and definition: 1978; L. Guibas and R. Sedgewick.
- AA-trees (1993: A. Anderson).

Balanced multiway search trees:

- B-trees (1972: R. Bayer and E. McCreight).

Self-balancing BSTs: AVL Trees

Complete binary trees have a too rigid balance condition to be maintained when new nodes are inserted.

Definition 3.14: An AVL tree is a BST with the following additional balance property:

- for any node in the tree, the height of the left and right subtrees can differ by at most 1.

The height of an empty subtree is -1.

Advantages of the AVL balance property:

- Guaranteed height $\Theta(\log n)$ for an AVL tree.
- Less restrictive than requiring the tree to be complete.
- Efficient ways for restoring the balance if necessary.

Self-balancing BSTs: AVL Trees

Lemma 3.15: The height of an AVL tree with n nodes is $\Theta(\log n)$.

Proof: Due to the possibly different heights of subtrees, an AVL tree of height h may contain fewer than $2^{h+1} - 1$ nodes of the complete tree.

- Let S_h be the size of the smallest AVL tree of height h.
- $S_0 = 1$ (the root only) and $S_1 = 2$ (the root and one child).
- The smallest AVL tree of height h has the smallest subtrees of height $h - 1$ and $h - 2$ by the balance property, so that

$$S_h = S_{h-1} + S_{h-2} + 1 = F_{h+3} - 1 \Leftrightarrow$$

i	1	2	3	4	5	6	7	...
h			0	1	2	3	4	...
F_i	1	1	2	3	5	8	13	...
S_h			1	2	4	7	12	...

where F_i is the i^{th} Fibonacci number (recall Lecture 6).

Self-balancing BSTs: AVL Trees (*Proof of Lemma 3.15* – cont.)

That $S_h = F_{h+3} - 1$ is easily proven by induction:

- **Base case:** $S_0 = F_3 - 1 = 1$ and $S_1 = F_4 - 1 = 2$.
- **Hypothesis:** Let $S_i = F_{i+3} - 1$ and $S_{i-1} = F_{i+2} - 1$.
- **Inductive step:** Then
  $S_{i+1} = S_i + S_{i-1} - 1 = \underbrace{F_{i+3} - 1}_{S_i} + \underbrace{F_{i+2} - 1}_{S_{i-1}} + 1 = F_{i+4} - 1$

Therefore, for each AVL tree of height h and with n nodes:

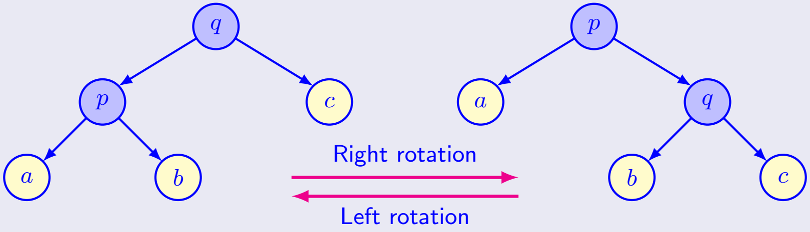
$$n \geq S_h \approx \frac{\varphi^{h+3}}{\sqrt{5}} - 1 \text{ where } \varphi \approx 1.618,$$

so that its height $h \leq 1.44 \lg(n+1) - 1.33$. □

- The worst-case height is at most 44% more than the minimum height for binary trees.
- The average-case height of an AVL tree is provably close to $\lg n$.

Self-balancing BSTs: AVL Trees

Rotation to restore the balance after BST insertions and deletions:



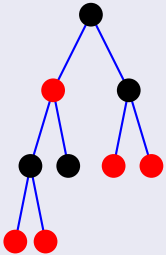
If there is a subtree of large height below the node a, the right rotation will decrease the overall tree height.

- All self-balancing binary search trees use the idea of rotation.
- Rotations are mutually inverse and change the tree only locally.
- Balancing of AVL trees requires extra memory and heavy computations.
- More relaxed efficient BSTs, r.g., red-black trees, are used more often in practice.

Self-balancing BSTs: Red-black Trees

Definition 3.17: A **red-black** tree is a BST such that

- Every node is coloured either **red** or **black**.
- Every non-leaf node has two children.
- The root is **black**.
- All children of a **red** node must be **black**.
- Every path from the root to a leaf must contain the same number of **black** nodes.



Theorem 3.18: If every path from the root to a leaf contains b black nodes, then the tree contains at least $2^b - 1$ black nodes.

Self-balaning BSTs: Red-black Trees

Proof of Theorem 3.18:

- **Base case:** Holds for $b = 1$ (either the black root only or the black root and one or two red children).
- **Hypothesis:** Let it hold for all red-black trees with b black nodes in every path.
- **Inductive step:** A tree with $b + 1$ black nodes in every path and two black children of the root contains two subtrees with b black nodes just under the root and has in total at least $1 + 2 \cdot (2^b - 1) = 2^{b+1} - 1$ black nodes.
- If the root has a red child, the latter has only black children, so that the total number of the black nodes can become even larger. □

Self-balancing BSTs: Red-black and AA Trees

Searching in a red-black tree is logarithmic, $\mathrm{O}(\log n)$.

- Each path cannot contain two consecutive red nodes and increase more than twice after all the red nodes are inserted.
- Therefore, the height of a red-black tree is at most $2\lceil\lg n\rceil$.

No precise average-case analysis (only empirical findings or properties of red-black trees with n random keys):

- The average case: $\approx \lg n$ comparisons per search.
- The worst case: $< 2\lg n + 2$ comparisons per search.
- $\mathrm{O}(1)$ rotations and $\mathrm{O}(\log n)$ colour changes to restore the tree after inserting or deleting a single node.

AA-trees: the red-black trees where the left child may not be red – are even more efficient if node deletions are frequent.

Balanced B-trees

The "Big-Oh" analysis is invalid if the assumed equal time complexity of elementary operations does not hold.

- External ordered databases on magnetic or optical disks.
  - One disk access – hundreds of thousands of computer instructions.
  - The number of accesses dominates running time.
- Even logarithmic worst-case complexity of red-black or AA-trees is unacceptable.
  - Each search should involve a very small number of disk accesses.
  - Binary tree search (with an optimal height $\lg n$) cannot solve the problem.

Height of an optimal m-ary search tree (m-way branching):

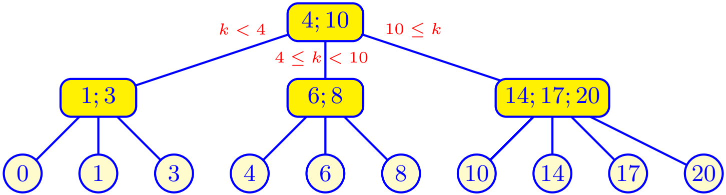
$$\approx \log_m n, \text{ i.e. } \approx \frac{\lg n}{\lg m}$$

Balanced B-trees

Height of the optimal m-ary search tree with n nodes:

n	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}	10^{12}
$\lceil \log_2 n \rceil$	17	20	24	27	30	33	36	39
$\lceil \log_{10} n \rceil$	5	6	7	8	9	10	11	12
$\lceil \log_{100} n \rceil$	3	3	4	4	5	5	6	6
$\lceil \log_{1000} n \rceil$	2	2	3	3	3	4	4	4

Multiway search tree of order $m = 4$:



Data records are associated only with leaves (most of definitions).

Balanced B-trees

A **B-tree** of order m is an m-ary search tree such that:

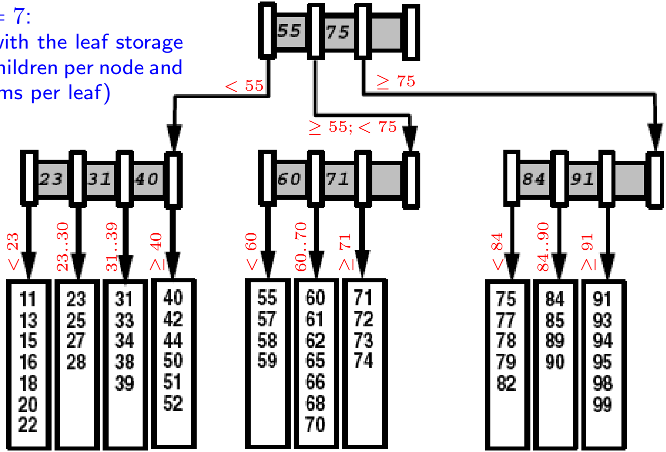
1. The root either is a leaf, or has $\mu \in \{2, \ldots, m\}$ children.
2. There are $\mu \in \{\lceil \frac{m}{2} \rceil, \ldots, m\}$ children of each non-leaf node, except possibly the root.
3. $\mu - 1$ keys, $(\theta_i : i = 1, \ldots, \mu - 1)$, guide the search in each non-leaf node with μ children, θ_i being the smallest key in subtree $i + 1$.
4. All leaves at the same depth.
5. Data items are in leaves, each leaf storing $\lambda \in \{\lceil \frac{l}{2} \rceil, \ldots, l\}$ items, for some l.

- Conditions 1–3: to define the memory space for each node.
- Conditions 4–5: to form a well-balanced tree.

Balanced B-trees

B-trees are usually named by their **branching limits** $\left\lceil \frac{m}{2} \right\rceil - m$: e.g., 2–3 trees with $m = 3$ or 2–4 trees with $m = 4$.

$m = 4$; $l = 7$:
2–4 B-tree with the leaf storage size 7 (2..4 children per node and 4..7 data items per leaf)



Balanced B-trees

Because the nodes are at least half full, a B-tree with $m \geq 8$ cannot be a simple binary or ternary tree.

◦ Simple **data insertion** if the corresponding leaf is not full.

◦ Otherwise, splitting a full leaf into two leaves, both having the minimum number of data items, and updating the parent node.

- If necessary, the splitting propagates up until finding a parent that need not be split or reaching the root.
- Only in the extremely rare case of splitting the root, the tree height increases, and a new root with two children (halves of the previous root) is created.

Data insertion, deletion, and retrieval in the worst case: about $\left\lceil \log_{\frac{m}{2}} n \right\rceil$ disk accesses.

- This number is practically constant if m is sufficiently big.