

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

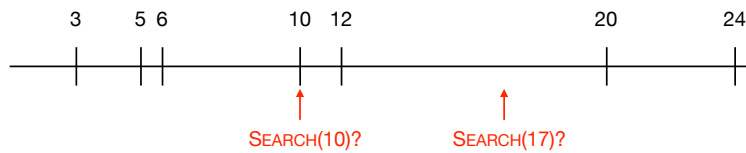
Philip Bille

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

Dynamic Ordered Sets

- **Dynamic Ordered Sets.** Maintain dynamic set S supporting the following operations. Each element x has key $x.key$ and satellite data $x.data$.
 - **SEARCH(k):** return element x such that $x.key = k$ if it exists. Otherwise return null.
 - **INSERT(x):** add x to S (assume $x.key$ is not already in S).
 - **DELETE(x):** remove x from S .
- We want to maintain elements **ordered** by the keys. Allows efficient support for many other important operations and other features.

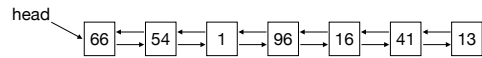


Dynamic Ordered Sets

- **Applications.**
 - Dictionaries.
 - Indexes.
 - Filesystem.
 - Databases.
 -
- **Challenge.** How can we solve problem with current techniques?

Dynamic Ordered Sets

- **Solution 1: linked list.** Maintain S in a doubly-linked list.



- SEARCH(k): linear search for largest key $\leq k$.
- INSERT(x): insert x in the front of list.
- DELETE(x): remove x from list.

- **Time.**

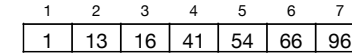
- SEARCH in $O(n)$ time ($n = |S|$).
- INSERT and DELETE in $O(1)$ time.

- **Space.**

- $O(n)$.

Dynamic Ordered Sets

- **Solution 2: sorted array.** Maintain S in an sorted array according to keys.



- SEARCH(k): binary search for k .
- INSERT(x): find index using SEARCH(x .key). Build new array of size $+1$ with x inserted.
- DELETE(x): build new array of size -1 with element with key k removed.

- **Time.**

- SEARCH in $O(\log n)$ time.
- INSERT and DELETE in $O(n)$ time.

- **Space.**

- $O(n)$.

Nearest Neighbor

Data structure	SEARCH	INSERT	DELETE	Space
linked list	$O(n)$	$O(1)$	$O(1)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$

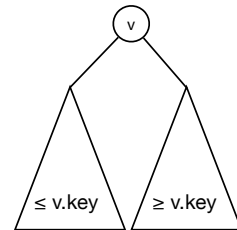
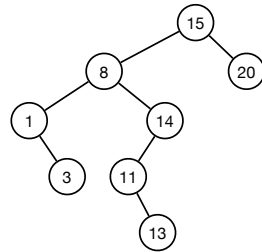
- **Challenge.** Can we do significantly better?

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

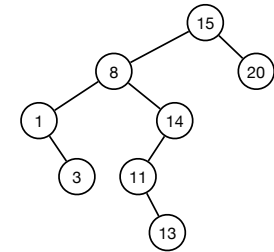
Binary Search Trees

- **Binary tree.**
 - Rooted tree
 - Each internal node has a **left child** and/or a **right child**.
- **Binary search tree.**
 - Binary tree in symmetric order.
- **Symmetric order.** For each vertex v :
 - all vertices in left subtree are $< v.key$.
 - all vertices in right subtree are $> v.key$.



Binary Search Trees

- Symmetric order ~ **inorder traversal** outputs the keys in sorted order.
- **Inorder traversal.**
 - Visit left subtree recursively.
 - Visit vertex.
 - Visit right subtree recursively.
- **Preorder traversal.**
 - Visit vertex.
 - Visit left subtree recursively.
 - Visit right subtree recursively.
- **Postorder traversal.**
 - Visit left subtree recursively.
 - Visit right subtree recursively.
 - Visit vertex.



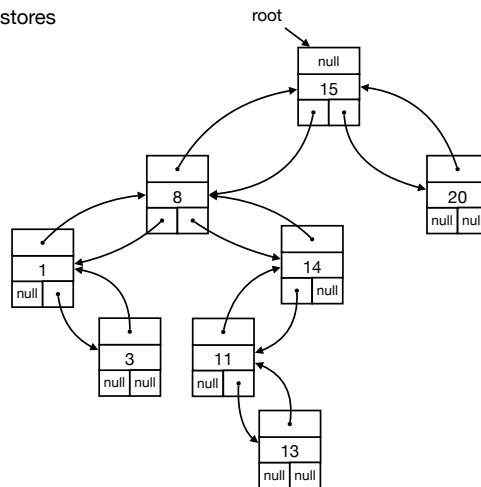
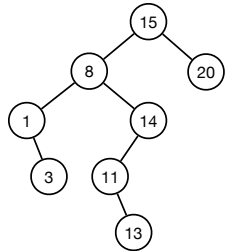
Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

Postorder: 3, 1, 13, 11, 14, 8, 20, 15

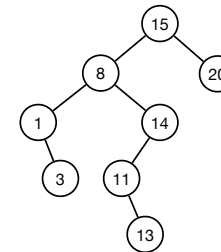
Binary Search Trees

- **Representation.** Each node x stores
 - $x.key$
 - $x.left$
 - $x.right$
 - $x.parent$
 - $(x.data)$
- **Space.** $O(n)$



Binary Search Trees

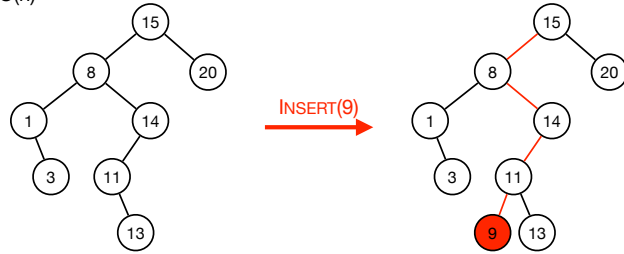
- **SEARCH(k):** traverse tree top-down.
 - Compare key k against key in node.
 - If equal return element. If less go left. If greater go right.
 - If we reach bottom, return null.
- **Time.** $O(h)$



Binary Search Trees

- INSERT(x): traverse tree top-down and compare keys.
 - search for x.
 - add x at leaf.

• Time: $O(h)$



Binary Search Trees

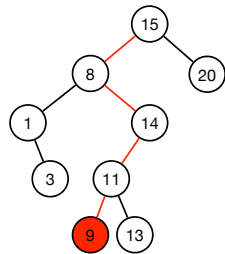
- INSERT(x): traverse tree top-down and compare keys.
 - if less go left; if greater go right; if equal, return node.
 - if null, insert x.

• Exercise. Insert following sequence in binary search tree: 6, 14, 3, 8, 12, 9, 34, 1, 7

Binary Search Trees

```

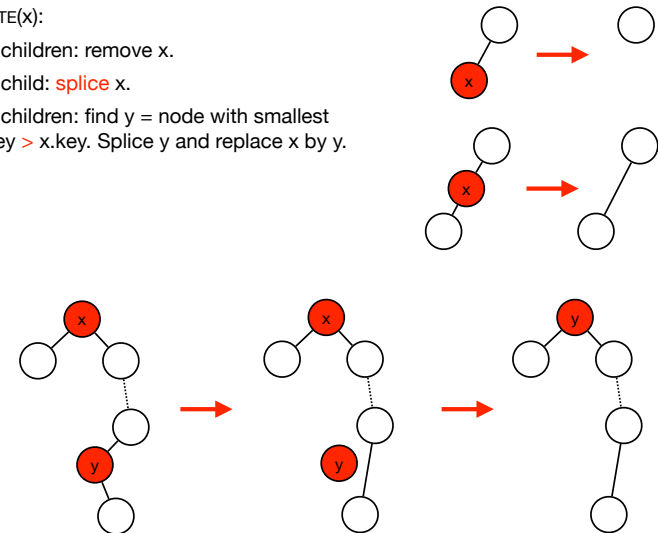
INSERT(x,v)
  if (v == null) return x
  if (x.key ≤ v.key)
    v.left = INSERT(x, v.left)
  if (x.key > v.key)
    v.right = INSERT(x, v.right)
    
```



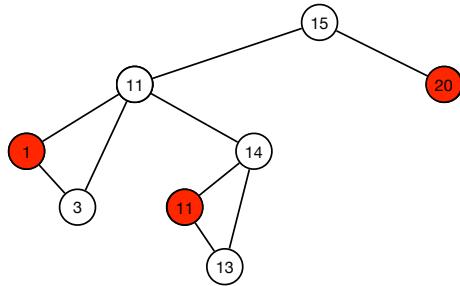
• Time: $O(h)$

Binary Search Trees

- DELETE(x):
 - 0 children: remove x.
 - 1 child: splice x.
 - 2 children: find y = node with smallest key $> x.key$. Splice y and replace x by y.



DELETE 20 1 8



Binary Search Trees

- DELETE(x):
 - 0 children: remove x.
 - 1 child: splice x.
 - 2 children: find y = node with smallest key > x.key. Splice y and replace x by y.
- Time. $O(h)$

Dynamic Ordered Sets

Data structure	SEARCH	INSERT	DELETE	Space
linked list	$O(n)$	$O(1)$	$O(1)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
binary search tree	$O(h)$	$O(h)$	$O(h)$	$O(n)$

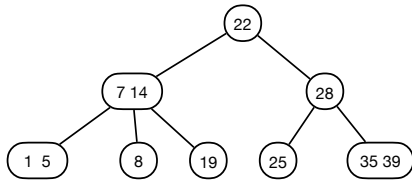
- Height. Depends on sequence of operations.
 - $h = \Omega(n)$ worst-case and $h = \Theta(\log n)$ on average.
- Challenge. Can we maintain height at $O(\log n)$ worst-case?

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

Balanced Search Trees

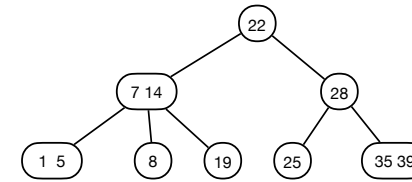
- **2-3 Tree.**
 - Rooted tree.
 - Each internal node has 2 or 3 children.
 - **2-node:** 2 children and 1 key
 - **3-node:** 3 children and 2 keys.
 - **Symmetric order.**
 - Inorder traversal outputs the keys in sorted order.
 - **Perfect balance.**
 - Every path from root to a leaf has the same length
- \Rightarrow height of tree is $\Theta(\log n)$



Balanced Search Trees

- **SEARCH(k):** traverse tree top-down.
 - Compare key k against keys in node.
 - If equal return element. Otherwise, recurse in child with interval containing k and recurse.
 - If we reach bottom, return null.

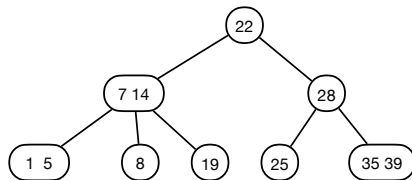
- **Time.** $O(\log n)$



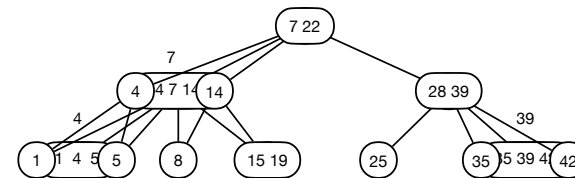
Balanced Search Trees

- **INSERT(x):**
 - Search for x .
 - Add x at leaf.
 - If too large, move middle key to parent. Repeat if necessary.

- **Time.** $O(\log n)$



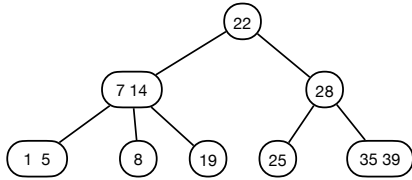
INSERT 15 42 4



Balanced Search Trees

- DELETE(x):
 - Search for x.
 - If x is not a leaf, find node with smallest key $> x$.key, swap with x, and delete it.
 - If too small, take from parent. Repeat if necessary.

- Time. $O(\log n)$



Dynamic Ordered Sets

Data structure	SEARCH	INSERT	DELETE	Space
linked list	$O(n)$	$O(1)$	$O(1)$	$O(n)$
sorted array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
binary search tree	$O(h)$	$O(h)$	$O(h)$	$O(n)$
2-3 tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

- Can we do better?
 - Many variants of balanced search trees supporting many different operations.
 - Many efficient practical solutions.
 - Optimal time bounds for comparison-based data structures.
 - Even better bounds possible with more advanced techniques.

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees