(Binary) Search Trees

Reminder: Dictionary ADT

**Operations:**

- **Initialize** - the data structure
  - **Search** - For an element with a given key
  - **Insert** - a new element
  - **Delete** - a specified element
- **Predecessor** - return element with closest smaller key
- **Successor** - return element with closest larger key
  - **Join** - combine two dictionaries to make a larger one
- **PrintSorted** - print the dictionary in sorted order

**Also useful:**

- **Minimum** - return the element with smallest key
- **Maximum** - return the element with largest key
Sequential Search (unsorted array)

Sequential-Search\( (A, k) \)
1 \( for \ i \leftarrow 1 \) to \( n \)
2 \( if \ A[i] = k \)
3 \( return \ i \)
4 \( return \ NIL \)

Running Time: \( O(n) \)
\[ n = \text{length}[A] \]

Sequential-Insert\( (A, k) \)
1 \( n \leftarrow n + 1 \)
2 \( A[n] \leftarrow k \)
3 \( return \ n \)

Running Time: \( O(1) \)
**Binary Search** (sorted array)

**Binary-Search**(A, k)

1. left ← 1 ; right ← length[A]
2. repeat
3. m ← (left+right)/2
4. if k < A[m]
5.   right ← m-1
6. else left ← m+1
7. until k = A[m] or left > right
8. if k = A[m]
9.   return m
10. else return NIL

**Running Time:** $O(\log n)$
Binary Search Comparison Tree

Search(M)
Binary Search Trees

Binary Search Tree Property:
For all nodes $x \in T$:

\[ y \in T_L(x) \Rightarrow \text{key}[y] \leq \text{key}[x] \]
\[ y \in T_R(x) \Rightarrow \text{key}[y] \geq \text{key}[x] \]
Binary Search Tree - example
Tree Walks
Tree Walks (reminder)

**InOrder-Tree-Walk(x)**
1 if x ≠ NIL
2 InOrder-Tree-Walk(left[x])
3 print key[x]
4 InOrder-Tree-Walk(right[x])

**PreOrder-Tree-Walk(x)**
1 if x ≠ NIL
2 print key[x]
3 PreOrder-Tree-Walk(left[x])
4 PreOrder-Tree-Walk(right[x])

**PostOrder-Tree-Walk(x)**
1 if x ≠ NIL
2 PostOrder-Tree-Walk(left[x])
3 PostOrder-Tree-Walk(right[x])
4 print key[x]

Running Time: $O(n)$
InOrder Walk

**InOrder-Tree-Walk**(x)

1. if x ≠ NIL
2. InOrder-Tree-Walk(left[x])
3. print key[x]
4. InOrder-Tree-Walk(right[x])

An InOrder traversal of a BST prints the elements in sorted order, in $O(n)$ time.
BST: Minimum/Maximum

**Tree-Minimum**(x)

1. while left[x] ≠ NIL
2. x ← left[x]
3. return x

**Tree-Maximum**(x)

1. while right[x] ≠ NIL
2. x ← right[x]
3. return x

**Running Time:** $O(h)$
BST: Successor

**Tree-Successor(x)**

1. If right[x] ≠ NIL
2. Return Tree-Minimum(right[x])
3. y ← p[x]
4. While y ≠ NIL and x = right[y]
5. x ← y
6. y ← p[y]
7. Return y

Running Time: O(h)
BST: Search

Tree-Search(x, k)

1 if x = NIL or k = key[x]
2 return x
3 if k < key[x]
4 return Tree-Search(left[x], k)
5 else
6 return Tree-Search(right[x], k)

Running Time: O(h)
BST: Insert

Tree-Insert(T, z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4      y ← x
5      if key[z] < key[x]
6          x ← left[x]
7      else x ← right[x]
8  endwhile
9  p[z] ← y
10  if y = NIL
11     root[T] ← z
12  else if key[z] < key[y]
13     left[y] ← z
14  else right[y] ← z

Running Time: O(h)
Worst Case depth

Diagram showing two trees with labels A, B, C, X, Y, Z.
Average Case

**Theorem:**
If we insert $n$ random elements into an initially empty Binary Search Tree, then the average node depth is $O(\log n)$

**Assume:**
- Trees are formed by insertions only
- All input permutations are equally likely
\( P(n) = \) average \# nodes on a root-to-node path

**Assume:**
Tree formed by inserting \( n \) random elements into an initially empty tree

\( P(0) = 0 \)
\( P(1) = 1 \)
\[ \frac{i}{n} (P(i) + 1) + \frac{n - i - 1}{n} (P(n - i - 1) + 1) + \frac{1}{n} \]
\[
\frac{i}{n} (P(i) + 1) + \frac{n - i - 1}{n} (P(n - i - 1) + 1) + \frac{1}{n}
\]

Average the sum over all \(i\)

\[
P(n) = 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} (iP(i) + (n - i - 1)P(n - i - 1))
\]

\[
P(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} ip(i) \quad \text{for } n \geq 2
\]

We can show by induction on \(n\):

\[
P(n) \leq 1 + 4 \log n = O(\log n)
\]
Conclusion:
If we insert $n$ random elements into an initially empty Binary Search Tree, then the average depth of a node is $O(\log n)$.
BST: Delete

Tree-Delete(T, z)
1  if left[z] = NIL or right[z] = NIL
2    y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5    x ← left[y]
6  else x ← right y
7  if x ≠ NIL
8    p[x] ← p[y]
9  if p[y] = NIL
10   root[T] ← x
11  else if y = left[p[y]]
12    left[p[y]] ← x
13    else right[p[y]] ← x
14  if y ≠ z
15   key[z] ← key[y]
16  return y

Running Time: O(h)
Deletion: leaf
Deletion: single child

Diagram of a binary search tree with nodes 15, 5, 3, 10, 12, 13, 6, 7, 16, 20, 18, 23, showing the process of deleting a node with a single child.
Deletion: two children
Balanced Trees

\[ x \leq T_L(x) \leq T_R(x) \]

\[ |T_L(x)| \approx |T_R(x)| \]

\[ \implies \text{all leaves have similar depth - } O(\log n) \]
Top-Down 2-3-4 Trees

- **2-nodes** – as in a BST – hold 1 key and two children
- **3-nodes** hold 2 keys and have 3 links to children
- **4-nodes** hold 3 keys and have 4 links to children

```
A
  +-------------------+
  |                  |
  x                  y
  +-------------------+
  |                  |
  B                  C

a ∈ A ⇒ a < x
b ∈ B ⇒ x ≤ b < y
c ∈ C ⇒ y ≤ c
```

```
A
  +-------------------+
  |                  |
  x                  y
  +-------------------+
  |                  |
  B                  C
  +-------------------+
  |                  |
  y                  z
  +-------------------+
  |                  |
  B                  D

a ∈ A ⇒ a < x
b ∈ B ⇒ x ≤ b < y
c ∈ C ⇒ y ≤ c < z
d ∈ D ⇒ z ≤ d
```
Example
Insert

- Do an unsuccessful search
- if search terminated at a 2-node, turn it into a 3-node
- if search terminated in a 3-node, turn it into a 4-node
- if search terminates at a 4-node,
  - split 4-node into one 2-node and one 3-node
  - pass one key back up to its parent
Insert - Splitting 4-nodes
Properties of $n$ node 2-3-4 trees

Property 1:
Worst-case search takes $O(\log n)$ time

Property 2:
- Worst-case for Insert makes $O(\log n)$ splits
- Average-case Insert seems to be $< 1$ split
Red-Black Trees Representation