Topic 4: Search Trees

Read: Chpt. 4 & 10, Weiss

Let S be a set of records with keys that can be linearly ordered. In general,
Record ↔ Instance of a class
Field ↔ Member variable
Key ↔ Form of identification

Typical Operations:
Static operations:
findMinKey, findMaxKey, findKey, …
Dynamic operations:
insertItemKey, deleteMinKey, deleteMaxKey, deleteItemKey, changeKey, …

Possible Approach:
Linear ADT such as sortedList.

Better Approach:
Nonlinear ADTs such as Binary search tree, 2-3 tree.

Designing Nonlinear ADT:
Always focusing on:
• Topological/Structural Property
• Relational Property
Search Tree:
Tree based data structure supporting frequent find operations.

Simplest Search Tree:
Binary search tree.

Defn: A binary search tree is a binary tree $H$ such that the key (priority) of any node $x$ is greater than the priority of all its left descendants and is smaller than or equal to the priority of all its right descendants (BST property).

BST:
Structural Property: A binary tree.
Relational Property: Satisfying BST property.

BST Property:

Remark: Duplicate elements are allowed in BST.
Observations:
1. BST structure models and generalizes binary search.
2. BST may not be a balanced binary tree.
3. BST can be a skew tree.
4. Leftmost descendant of root = item with min priority.
5. Rightmost descendant of root = item with max priority.
6. Inorder traversal = Sorted order.

Examples: BST using integer keys.

```
          7
         / \
        4   9
       / \
      6   8
     /   \
    5     
```

Example: BST using characters keys.

```
Emily
/   \
Allen   John
/     \
Alex    Fred
        /   \
        Emm   Hillar
```
Modeling Binary Search using a BST:

Consider binary searching a sorted array A:

\[
\begin{array}{cccc}
  & i & m-1 & m & m+1 & j \\
A: & & \ldots & y & \ldots & \\
  & x:y & \\
\end{array}
\]

Binary Search Algorithm:

\[
m \leftarrow (i+j)/2;
\]

if \( x = A[m] \)

then return \( m \)

else if \( x < A[m] \)

then search\((i,m-1,x)\)

else search\((m+1,j,x)\)

endif;

endif;

Observation:

In general, one can perform a 2-ary search by comparing \( x \) with any element \( y \) with index \( k \) in \( A \):

\[
\begin{array}{cccc}
  & i & k-1 & k & k+1 & j \\
A: & & \ldots & y & \ldots & \\
  & x:y & \\
\end{array}
\]
Generalized 2-ary Search Algorithm:

\[ m \leftarrow k; \]
\[ \text{if } x = A[m] \]
\[ \quad \text{then return } m \]
\[ \text{else if } x < A[m] \]
\[ \quad \text{then search}(i, m-1, x) \]
\[ \quad \text{else search}(m+1, j, x) \]
\[ \text{endif;} \]
\[ \text{endif;} \]

Binary Search Tree and Generalized 2-ary Search:

\[ A[k] = y \]

\[
\begin{array}{c}
\text{x < y} \\
\text{x ≥ y}
\end{array}
\]

\[
\begin{array}{c}
A[i],\ldots,A[k-1] \\
A[k+1],\ldots,A[j]
\end{array}
\]
Implementing BST:
1. *Array implementation:*
   Infeasible!
   Why not?

2. *Pointer-based Implementation:*
   **TreeNode:**

   ![TreeNode Diagram]

   **Example:**

   ![Example Tree Diagram]
BST Operations:
1. Search Operation:
   Think of binary search!

Example: Consider search(T,5).

![BST Tree Diagram]

Algorithm:
search(in binTree: BST, in searchKey: KeyType)
{
    if (binTree = NULL) // empty BST
        return not found;
    else if (binTree→key = searchKey) // key found
        return found;
    else if (binTree→key > searchKey) // search L-tree
        return search(binTree→lchild, searchKey)
    else // search R-tree
        return search(binTree→rchild, searchKey);
} // end search
2. **Insert Operation:**

   Find position for insertion using search. When position found (pointer = NULL), create new TreeNode and insert.

**Algorithm:**

```c
InsertItem(inout treePtr: TreeNodePtr, 
           in newItem: TreeItemType)
{
    if (treePtr = NULL) // empty BST
        create new TreeNode and insert;
    else if (newItem.getKey() < treePtr->item.getKey())
        insertItem(treePtr->lchild, newItem);
    else insertItem(treePtr->rchild, newItem);
} // end search
```
Example: Insert items with keys 7, 3, 1, 8, 13, 15, 6, 9, 10, in the given order, into an initially empty BST.
3. **Delete Operation:**
   (a) Consider first a `deleteMin(T)` operation. Observe that the min element x must be the leftmost descendant of the root and, x must have 0 or 1 child. Hence, we can simply replace x with its right child (may be empty) in the BST.

**Before:**

```plaintext
    T
   /|
  x  y
  /|
 z z
```

**After:**

```plaintext
    T
   /|
  z z
  /|
 y y
```
(b) Consider the general \textit{delete}(T,k) operation. Let N be the node with key k.

\textbf{Three cases:}
\begin{enumerate}
\item N has no child: Remove N.
\item N has exactly one child: Replace N with its only child.
\item N has two children: Replace N with the min priority item of its right subtree (using deleteMin operation).
\end{enumerate}

\textbf{Remark:} We can also use deleteMax operation to support the implementation of the general delete operation. For consistency, we will only use deleteMin operation in delete operations.

\textbf{Example:} Delete 3, 7, 8 from the following BST T.
delete(T,3):

delete(T,7):

delete(T,8):
**Complexity Analysis:**

For above BST operations, observe that $T(n) = O(h)$, where $h$ is the height of a given BST. Hence,

<table>
<thead>
<tr>
<th></th>
<th>findMin</th>
<th>findMax</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_w(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T_a(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>deleteMin</th>
<th>deleteMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_w(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T_a(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

**Remark:** Despite of the $T_w(n) = O(n)$ worst-case performance of all standard BST operations, BST remains an attractive search tree structure due to its simplicity, in both concept and implementation, and good average-case performance.

**One Remaining Problem:**

How do we construct a BST for a given set of records $S$?

Using insert operations, we can build a BST with

$$T_w(n) = 1 + 2 + 3 + \ldots + (n-1)$$

$$= n(n-1)/2$$

$$= O(n^2).$$
**Saving and Restoring a BST in a File:**

**Q:** How do we save a BST in a (sequential) file so that it can be restored later if needed?

**A:** Use preorder traversal.

Original BST can be restored by inserting those records in the preorder traversal sequence into an initially empty BST.

**Example:** The BST with the preorder traversal sequence 7, 3, 1, 6, 8, 13, 9, 10, 15.

![BST Diagram]

**Q:** What if we would like to balance the BST?
Building a Balanced BST:

Observe that for any given set of records $S$ with distinct keys, the representation of $S$ using BST is not unique. However, once the topology of a binary tree structure is given, the representation of $S$ using BST with the given structure must be unique. Hence, we can restructure a given BST $T$ by first traversing $T$ in inorder and then build a (complete) BST for $T$ using its inorder traversal.

Example: Given the above BST $T$ representing the set $S = \{7, 3, 1, 6, 8, 13, 9, 10, 15\}$.
Inorder traversal of $T$: $1, 3, 6, 7, 8, 9, 10, 13, 15$.

A complete BST $T$ for $S$:

![Diagram of a BST](image)

HW. Compare the implementations of the above buildTree operations.
To Sort, Or Not To Sort:
Given a set of n records S with keys \( x_1, x_2, \ldots, x_n \), \( x_1 \leq x_2 \leq \ldots \leq x_n \).

For a given key x, let \( \Pr(x_i = x) = p_i \), \( 1 \leq i \leq n \), and \( \sum_{i=1}^{n} p_i = 1 \).

Q: If m searches are to be performed, \( m \gg n \), what kind of DS should be used to store this set of records such that the average search time for x is minimized?

**Approach 1:**
Sort the records into non-decreasing order according to their keys \( x_i \) and then store the sorted records in an array A. Apply binary search to A to search for x.

\[
T_a(n) = m \cdot O(\log n).
\]

**Remark:** This approach does not use any of the given information on \( p_i \).

**Approach 2:**
Sort the records into non-increasing order according to their probabilities \( p_i \) and then store the sorted records in an array A. Apply sequential search to A to search for x.

\[
T_a(n) = m \cdot \sum_{i=1}^{n} i \cdot p_i.
\]

**Remark:** This approach does not use any of the given information on \( x_i \).
**Approach 3:**
Store \( \{x_1, x_2, \ldots, x_n\} \) in a BST \( T \). Apply search\((T,x)\) to \( T \).
\[
T_a(n) = m \cdot O(\lg n),
\]

**Remark:** Which structure of a BST should we use? Also, this approach does not use any of the given information on \( p_i \).

**Approach 4:**
Use the structure of an optimal BST to minimize the average search time for \( x \).

Recall that the representation of \( S \) using BST is not unique. Let’s try to construct a BST for \( S \) that will minimize \( T_a(n) \). This is known as the **Optimal BST Problem**.

Let \( T \) be any BST representing \( S \) with \( n \) objects.
\[
T_a(n) = \sum_{i=1}^{n} p_i \cdot [d(x_i) + 1],
\]

where \( d(x_i) \) is the depth of \( x_i \).

Observe that in any BST for \( \{x_i, x_{i+1}, \ldots, x_j\} \), one of these elements, say \( x_k \), must be the root of the BST.
Q: What are the elements forming the left (right) subtree of $x_k$?

Observe that $x_k < \text{every element in } \{x_i, x_{i+1}, \ldots, x_{k-1}\}$ and $x_k \geq \text{every element in } \{x_{k+1}, x_{k+2}, \ldots, x_j\}$. Hence, once an element $x_k$ is chosen as the root of a binary search tree (subtree), the left subtree as well as the right subtree of $x_k$ is automatically fixed.

A greedy approach:
Use the element with the highest probability as the root for each tree (subtree)!

Q: Is it optimal?
Example:
Consider $<x_1, x_2, x_3, x_4>$ with $p_1 = 0.1, p_2 = 0.2, p_3 = 0.3, p_4 = 0.4$.

$$T_a(n) = 1 \times 0.4 + 2 \times 0.3 + 3 \times 0.2 + 4 \times 0.1 = 2.0$$

Optimal BST:

$$T_a(n) = 1 \times 0.3 + 2 \times 0.2 + 2 \times 0.4 + 3 \times 0.1 = 1.8$$
Computing Opt BST:
Let $c_{i,j} =$ the min average cost in searching for $x$ in an optimal BST formed by $\{x_i, x_{i+1}, \ldots, x_j\}$.

Observe that the two subtrees formed by $\{x_i, x_{i+1}, \ldots, x_{k-1}\}$ and $\{x_{k+1}, x_{k+2}, \ldots, x_j\}$ must also be an optimal BST. Hence,

$$c_{i,j} = \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j} + 1 \ast p_k + \sum_{l=i}^{k-1} p_l + \sum_{l=k+1}^{j} p_l\}.$$  

Or,

$$c_{i,j} = \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j} + \sum_{l=i}^{j} p_l\}.$$  

Observe that, for all $i$, $c_{i,i} = p_i, c_{i+1,i} = 0$.

To solve the optimal BST problem, we need to compute $c_{i,j}$ and to re-construct the optimal BST by keeping track of the root $x_k$ used in each subtree.

**Approach:**
To compute $c_{1,n}$, do
Step 1: Compute $c_{i,i}$ for all $i$.
Step 2: Compute $c_{i,j}$ in increasing difference of $(j-i)$.

**Q:** How do we recover the structure of the opt BST?
Define \( t_{i,j} = k \) iff \( x_k \) is the root of an optimal BST formed by \( \{x_i, x_{i+1}, \ldots, x_k, x_{k+1}, \ldots, x_j\} \).

**Dynamic Programming Algorithm:**
for \( i = 1 \) to \( n \) do // initialization
\[
    c_{i,j} = p_i; \\
    t_{i,i} = i;
\]
endfor;

for \( m = 1 \) to \( n-1 \) do // compute \( c_{i,j} \) in increasing \( m \)
    for \( i = 1 \) to \( n-m \) do
        \( j = i + m; \)
        \( \text{sum} = 0; \) // computing \( \text{sum}(p_i, \ldots, p_j) \)
        for \( l = i \) to \( j \) do
            \( \text{sum} = \text{sum} + p_l; \)
        endfor;
        \( c_{i,j} = \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j}\} + \text{sum} \)
        \( t_{i,j} = k; \)
    endfor;
endfor;

**Complexity Analysis:**
\( T(n) = O(n^3), \)
\( S(n) = O(n^2). \)
**Example:** Given \( \{x_1, x_2, x_3, x_4\} \) with \( p_1 = 0.1 \), \( p_2 = 0.2 \), \( p_3 = 0.3 \), \( p_4 = 0.4 \).

\[
c_{1,1} = 0.1, c_{2,2} = 0.2, c_{3,3} = 0.3, c_{4,4} = 0.4.
\]

\[
c_{1,2} = \min\{c_{1,0} + c_{2,2}, c_{1,1} + c_{3,2}\} + \sum_{l=1}^{2} p_l = \min\{0.2, 0.1\} + 0.3 = 0.4, t_{1,2} = 2.
\]

\[
c_{2,3} = \min\{c_{2,1} + c_{3,3}, c_{2,2} + c_{4,3}\} + \sum_{l=2}^{3} p_l = \min\{0.3, 0.2\} + 0.5 = 0.7, t_{2,3} = 3.
\]

\[
c_{3,4} = \min\{c_{3,2} + c_{4,4}, c_{3,3} + c_{5,4}\} + \sum_{l=3}^{4} p_l = \min\{0.4, 0.3\} + 0.7 = 1.0, t_{3,4} = 4.
\]

\[
c_{1,3} = \min\{c_{1,0} + c_{2,3}, c_{1,1} + c_{3,3}, c_{1,2} + c_{4,3}\} + \sum_{l=1}^{3} p_l = \min\{0.7, 0.4, 0.4\} + 0.6 = 1.0, t_{1,3} = 2.
\]

\[
c_{2,4} = \min\{c_{2,1} + c_{3,4}, c_{2,2} + c_{4,4}, c_{2,3} + c_{5,4}\} + \sum_{l=2}^{4} p_l = \min\{1.0, 0.6, 0.7\} + 0.9 = 1.5, t_{2,4} = 3.
\]

\[
c_{1,4} = \min\{c_{1,0} + c_{2,4}, c_{1,1} + c_{3,4}, c_{1,2} + c_{4,4}, c_{1,3} + c_{5,4}\} + \sum_{l=1}^{4} p_l = \min\{1.5, 1.1, 0.8, 1.0\} + 1.0 = 1.8, t_{1,4} = 3.
\]

**Constructing Optimal BST:**

\[
\begin{align*}
t_{1,4} &= 3 \\
t_{1,2} &= 2 \\
t_{4,4} &= 4 \\
t_{1,1} &= 1
\end{align*}
\]
Recall that BST is a very attractive and useful data structure but it can be highly unbalanced, resulting in worst-case $O(n)$ complexity!

**Q:** Can we design a balanced search tree data structure such that insert, find, deletemin, deletemax, and delete operations all have $T_w(n) = O(\log n)$?

**Balanced Tree Structures:**
Non-binary tree: Less complicated (2-3 tree)
Binary tree structure: Very complicated (AVL tree)

**Q:** What is a 2-3 tree?

Recall that a BST can be used to model a 2-ary search.

**Q:** Why just consider 2-ary search? Should one consider k-ary search, $k > 2$?

Consider 3-ary Search:

<table>
<thead>
<tr>
<th>s</th>
<th>j</th>
<th>k</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>y</td>
<td>...</td>
<td>z</td>
</tr>
</tbody>
</table>

A:

- $x < y$: search($x$, s, j-1);
- $y < x < z$: search($x$, j+1, k-1);
- $x > z$: search($x$, k+1, t);
Observe that we must know y and z in order to perform a 3-ary search! Hence, in order to model a 3-ary search, we must store the information for both y and z. A 2-3 tree is a tree that can be used to model a 3-ary search. In general, a 2-3-4-…-m tree can be constructed in a similar fashion to model a (m-1)-ary search.

**Basic 2-3 tree:**

![2-3 Tree Diagram](image)

**Nodes in 2-3 Tree:**
1. Interior Node: Holding information to facilitate searching.
2. Leaf Node: Holding actual data record.
Characteristics of 2-3 Tree T:
1. There are two types of nodes in T: 
   *Leaf nodes* and *non-leaf (interior) nodes*.
2. All data elements are stored in the leaf nodes and they must be ordered from left (minimum) to right (maximum).
3. All leaf nodes must have the same depth.
4. Each interior node can either be a 2-node with exactly two subtrees or a 3-node with exactly three subtrees.
5. If an interior node is a 2-node, it will hold the minimum key of its second subtree. If an interior node is a 3-node, it will hold the minimum key of both its second and third subtrees.
6. An empty tree and a tree containing a single data element in a leaf node are 2-3 trees.

Example:
Node Structure:

<table>
<thead>
<tr>
<th>minSecond</th>
<th>parent</th>
<th>minThird</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>key</td>
<td>tag</td>
</tr>
<tr>
<td>first</td>
<td>second</td>
<td>third</td>
</tr>
</tbody>
</table>

- tag = 0  ⇒  interior node
- tag = 1  ⇒  leaf node

Example:
**Typical 2-3 Tree Operations:**

Find, Insert, FindMin, FindMax, DeleteMin, DeleteMax, Delete.

Consider $\text{search}(x, T)$.

\[
\begin{align*}
\text{if } (T \rightarrow \text{tag} == 1) & \quad // \text{leaf node found} \\
& \quad \text{then return } (x == T \rightarrow \text{key}) \\
& \quad \text{else temp} = T; \\
& \quad \quad \text{if } (x < T \rightarrow \text{minSecond}) \\
& \quad \quad \quad \text{then return } \text{search}(x, T \rightarrow \text{first}) \\
& \quad \quad \quad \text{else if } (T \rightarrow \text{minThird} != -1 \text{ and } \\
& \quad \quad \quad \quad \quad x \geq T \rightarrow \text{minThird}) \\
& \quad \quad \quad \quad \quad \quad \text{then } \text{search}(x, T \rightarrow \text{third}) \\
& \quad \quad \quad \quad \quad \quad \text{else } \text{search}(x, T \rightarrow \text{second}) \\
& \quad \quad \quad \quad \quad \quad \quad \text{endif;} \\
& \quad \quad \text{endif;} \\
& \quad \text{endif;}
\end{align*}
\]

**Complexity:**

Search operation depends on height of 2-3 tree. Since a 2-3 tree with $n$ data objects has height $h$, $\log_2 n \leq h \leq \log_3 n$, $\text{search}(x, T)$ has complexity $T_w(n) = O(\lg n)$. 
Consider \texttt{insert}(x,T):

\textbf{Case Analysis:}

0: Create a new leaf node with \( x \).
1. If \( T = \text{NULL} \), return \( T \) with one node.
2. If \( T \) has one node \( y \), create a new interior node with children \( x \) and \( y \).
3. In general, find parent \( N \) of \( x \) for insertion.
   (a) If \( N \) is a 2-node, insert \( x \) and adjust \( N \).
   (b) If \( N \) is a 3-node, split \( N \) into two interior nodes (2-nodes) \( N_1 \) and \( N_2 \) with \( x \) inserted. Adjust \( N_1 \) and \( N_2 \).
      (i) If \( N \) was the root of \( T \), create a new interior node, which becomes the new root of \( T \), having children \( N_1 \) and \( N_2 \).
      (ii) If \( N \) was not the root of \( T \), \( N \) must have a parent \( p(N) \). Attach \( N_1 \) to \( p(N) \) as a child and then insert \( N_2 \) to \( p(N) \) as before.

\textbf{Complexity:}

\[ T_w(n) = O(\log n). \]
Consider **delete(x,T):**

**Case Analysis:**
1. If T = NULL, return error.
2. If T has one node, T becomes NULL if x is found.
3. In general, find parent N of x and delete x from N.
   (a) If N is a 3-node, delete x and done.
   (b) If N is a 2-node, delete x and N becomes a “1-node”.
      (i) If N was the root of T, destroy the interior node N and T becomes a 2-3 tree with just one leaf node.
      (ii) If N was not the root of T, N must have a parent p(N) and N must have an immediate sibling N*. If N* is a 3-node, then N can “adopt” a new child from N*. If N* is a 2-node, then N will give its only child to N* for adoption and N will now become childless! Delete N from p(N) as before.

**Complexity:**
\( T_w(n) = O(\log n) \).

Consider **buildTree** using insert operations:
\( T_w(n) = O(n\log n) \).