EECS560 Data Structures Kong

Topic 4: Basic Search Trees and their Implementations

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, AVL tree, splay tree, etc.

Designing Nonlinear ADT:
Always focusing on

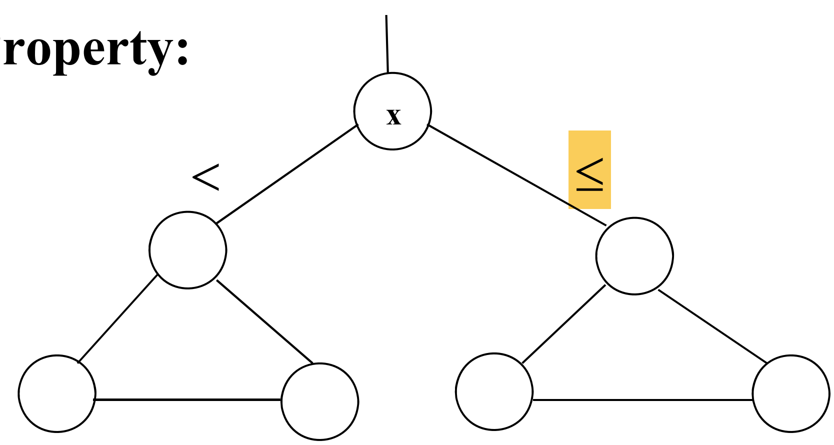
- Topological/Structural Property
- Relational Property

Search Tree:
Tree based data structure supporting frequent find/search operations.

Simplest Search Tree:
Binary search tree (BST).

Defn: A ***binary search tree*** is a binary tree T satisfying the following ***BST property***: the key (priority) of any node x in T 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:

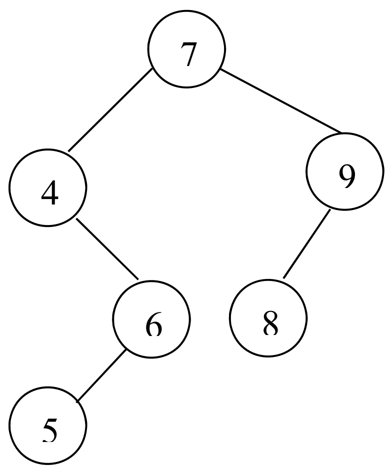


Remark: Duplicate elements are allowed in BST.

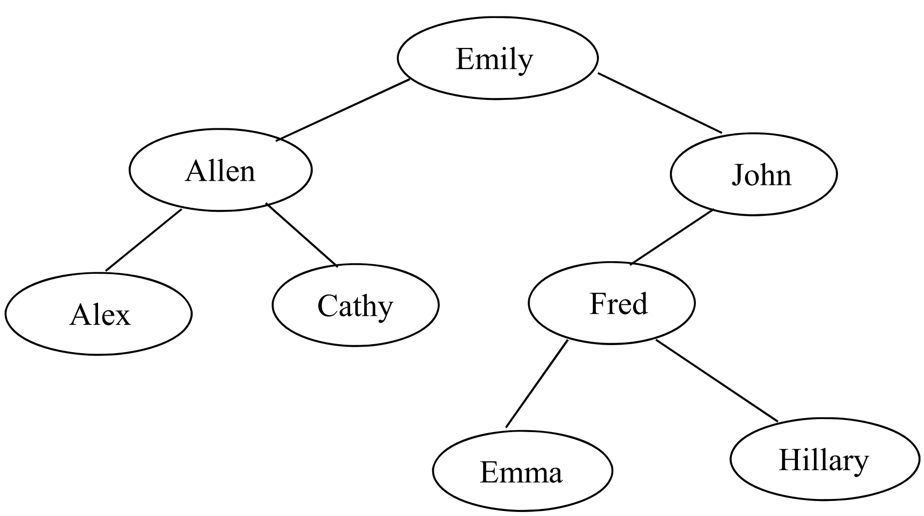
Observations:

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.



Example: BST using characters keys.



Modeling Binary Search using a BST:

Consider searching a sorted array A of distinct elements using binary search:

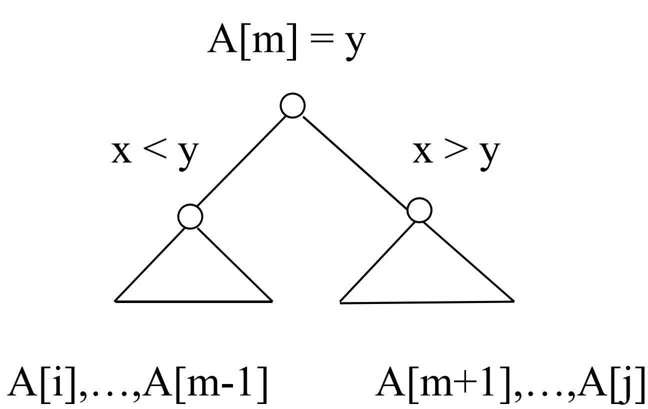
	i		m-1	m	m+1		j
A:		…		**y**		…	

x:y

Binary Search Algorithm:

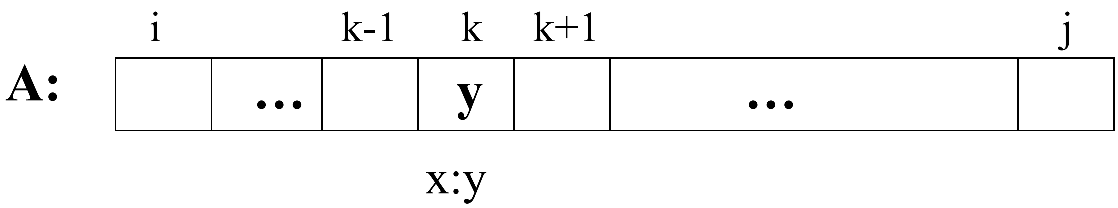
```
m ← (i+j)/2;
if x = A[m]
   then return m
   else if x < A[m]
            then search(i,m-1,x)
            else  search(m+1,j,x)
         endif;
endif;
```

Modeling Binary Search:



Generalization of Binary Search:

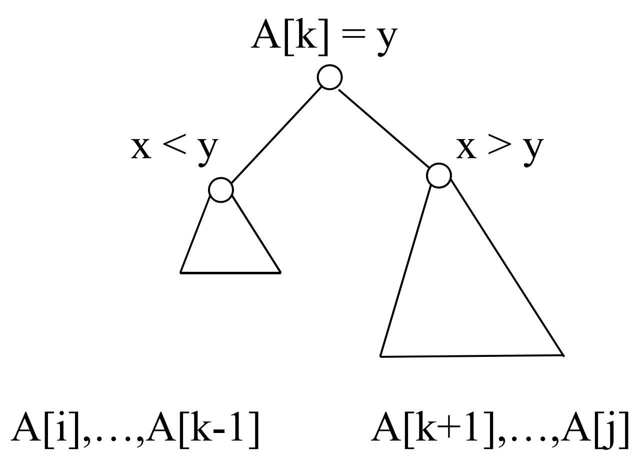
In general, one can perform a 2-ary search by comparing x with ***any*** element y with index k in A.



Generalized 2-ary Search Algorithm:

```
m ← k;
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;
```

Modeling Generalized 2-ary Search:



Implementing BST:

1. ***Array implementation***:

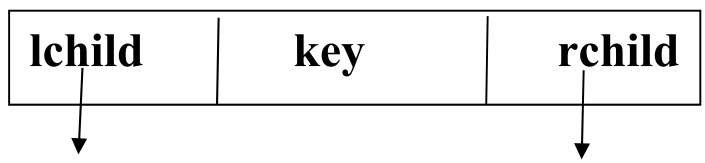
   Infeasible!

   Why not?

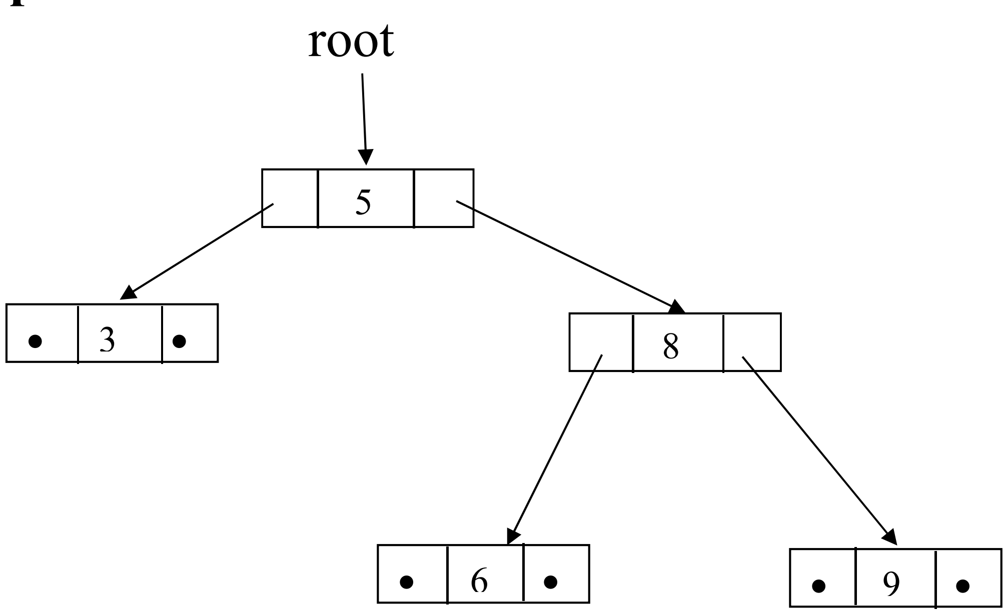
   BST will be skew tree, array node is followed by height 2^(h+1)+1

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

TreeNode:



Example:

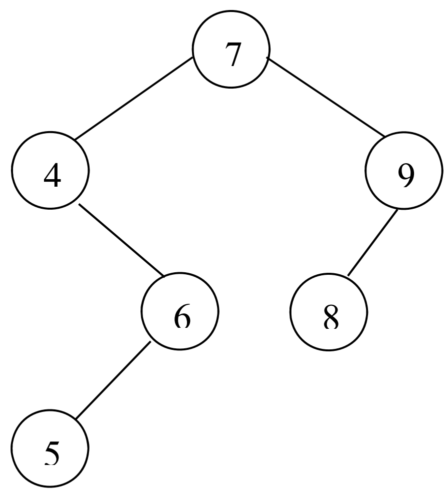


BST Operations:

1. ***Search Operation***:

   Think of binary search!

Example: Consider search(T,5).



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
```

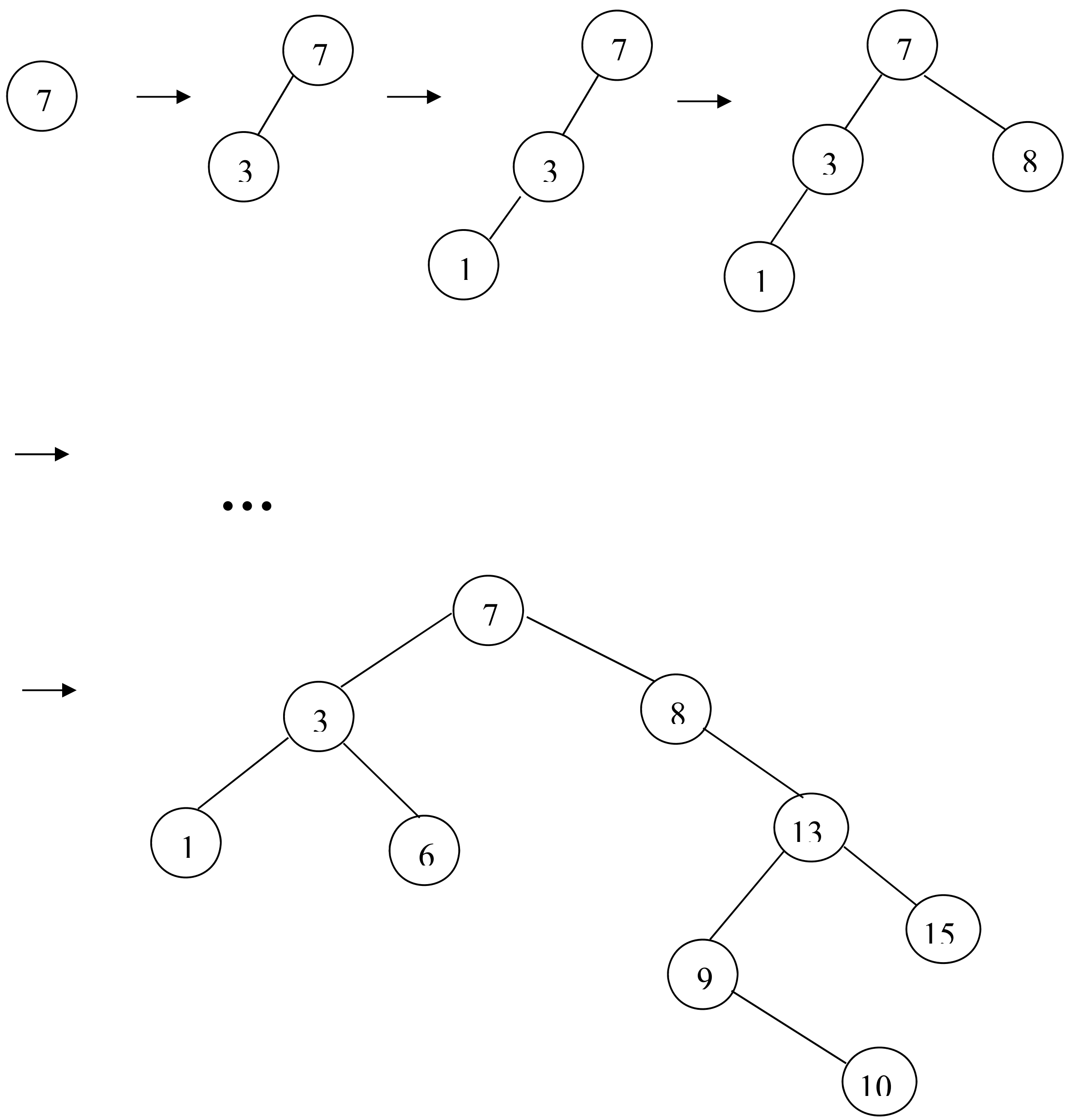
2. ***Insert Operation***:

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

Algorithm: 使用指针的引用;BinaryNode* &curr;

```
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
```

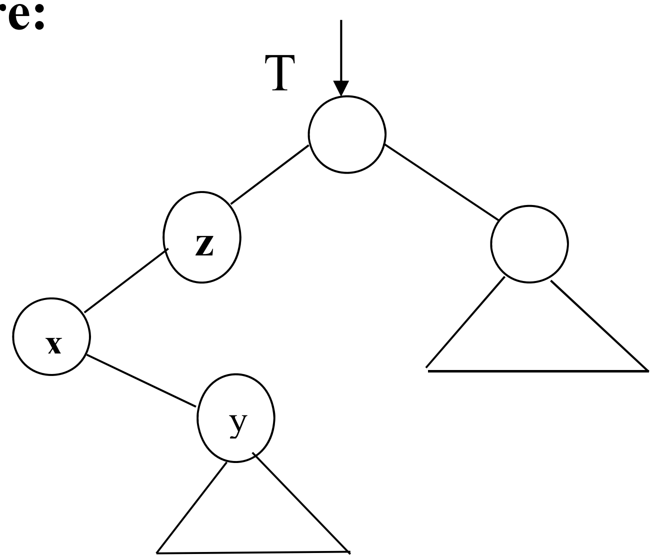
Example: Insert items with keys 7, 3, 1, 8, 13, 15, 6, 9, 10, in the given order, into an initially empty BST.



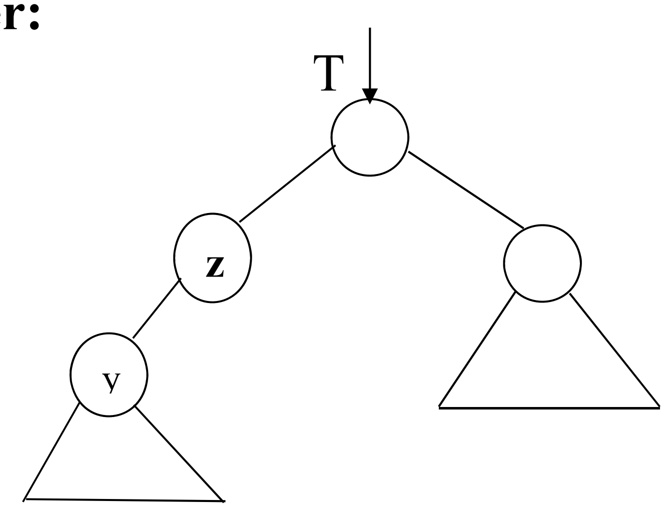
3. ***Delete Operation***:

(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:



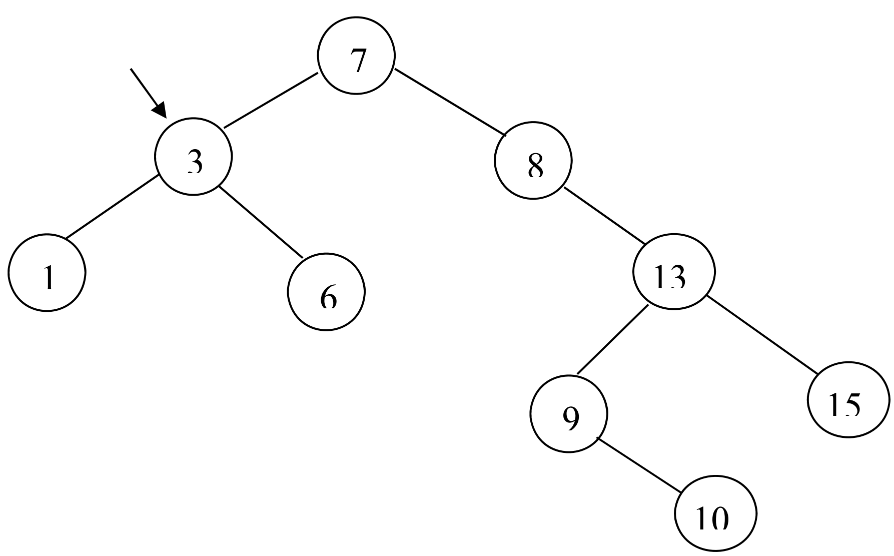
After:



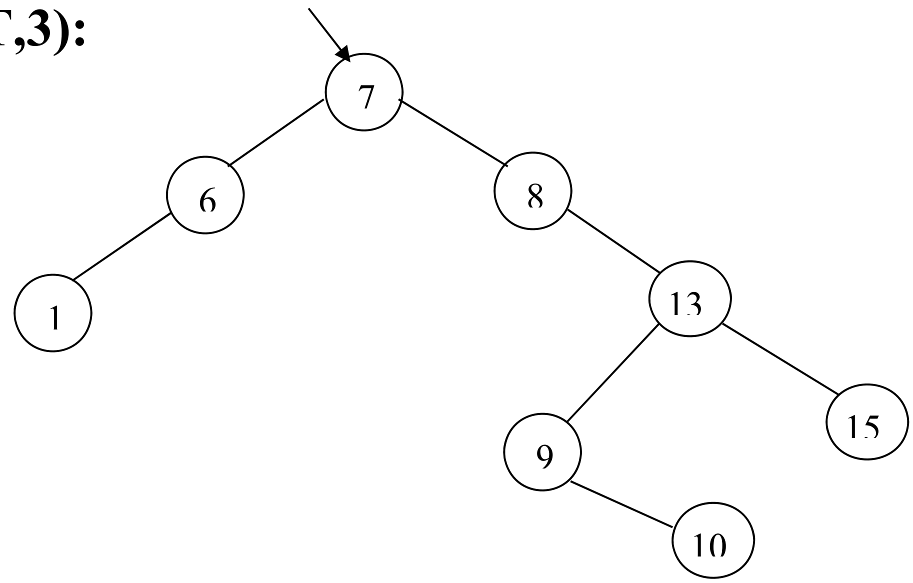
(b) Consider the general ***delete(T,k)*** operation.
Let N be the node with key k.
Three cases:
1. N has no child: Remove N.
2. N has exactly one child: Replace N with its only child.
3. N has two children: Replace N with the min priority item of its right subtree (using deleteMin operation).

Warning: Do not use deleteMax operation on the left subtree for the general delete operation. If there are duplicate elements in a BST, using deleteMax operation on the left subtree for the general delete operation may result in incorrect BST.

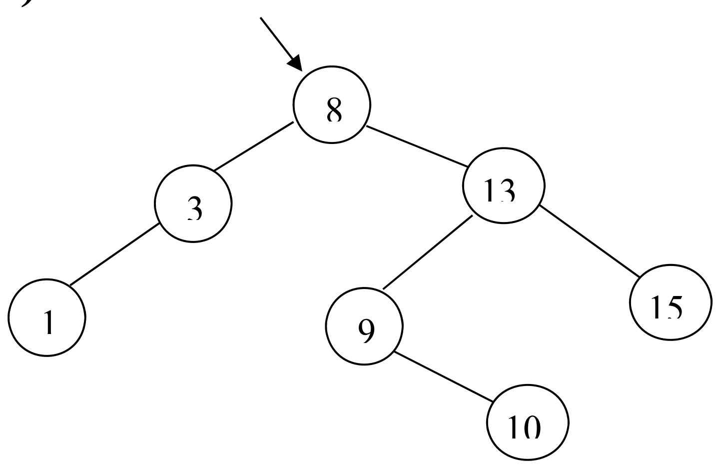
Example: Delete 3, 7, 8 from the following BST T.



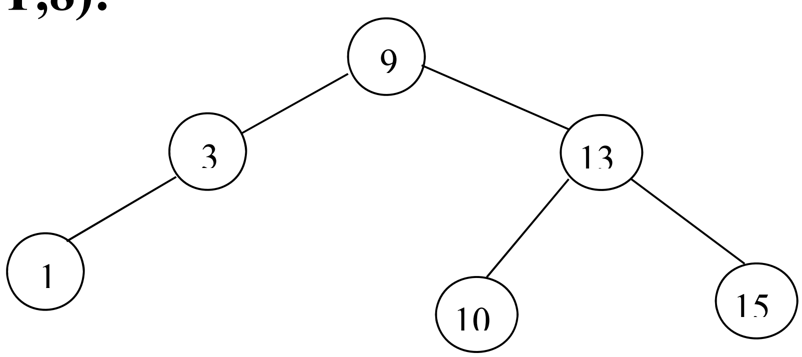
delete(T,3):



delete(T,7):



delete(T,8):



Advantage and disadvanages

Complexity Analysis:

For above BST operations, observe that $T(n) = O(h)$, where h is the height of a given BST. Hence, for the worst-case performance of all standard BST operations, $T_w(n) = O(n)$. However, BST remains an attractive search tree structure due to its simplicity and good average-case performance.

	findMin	**findMax**	**search**	**insert**	**delete**
$\mathbf{T_w(n)}$	O(n)	O(n)	O(n)	O(n)	O(n)
$\mathbf{T_a(n)}$	O(lgn)	O(lgn)	O(lgn)	O(lgn)	O(lgn)

	deleteMin	**deleteMax**
$\mathbf{T_w(n)}$	O(n)	O(n)
$\mathbf{T_a(n)}$	O(lgn)	O(lgn)

Q: How do we construct an initial BST for a given set of records S?

A: One can always build the initial data structure by inserting the elements in a given set S into an initially empty structure.

Using insert operations, we can build a BST with

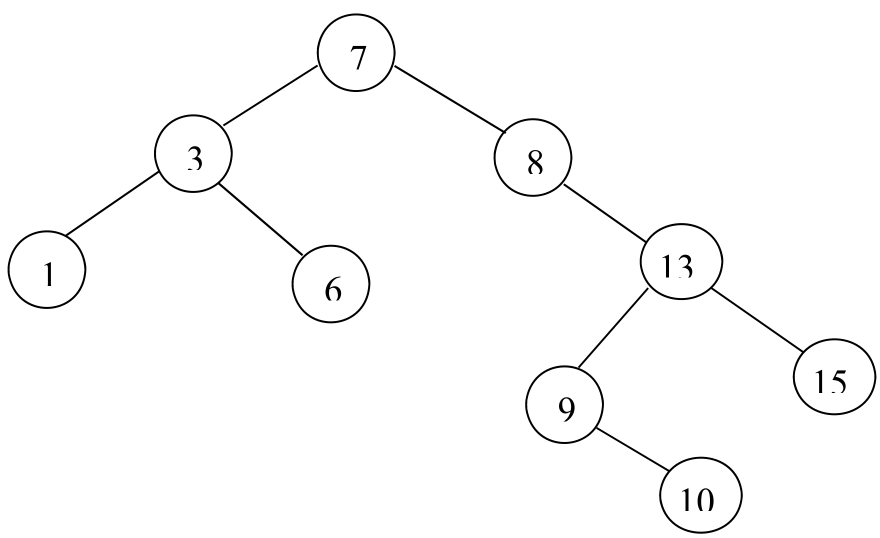
$$
\begin{aligned}
T_w(n) &= 1 + 2 + 3 + \ldots + (n-1) \\
&= n(n-1)/2 \\
&= O(n^2).
\end{aligned}
$$

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: Using preorder traversal.
Original BST can be restored by inserting those records, from left to right, 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.



HW: Explain how you can use the postorder traversal of a BST to reconstruct the original BST.

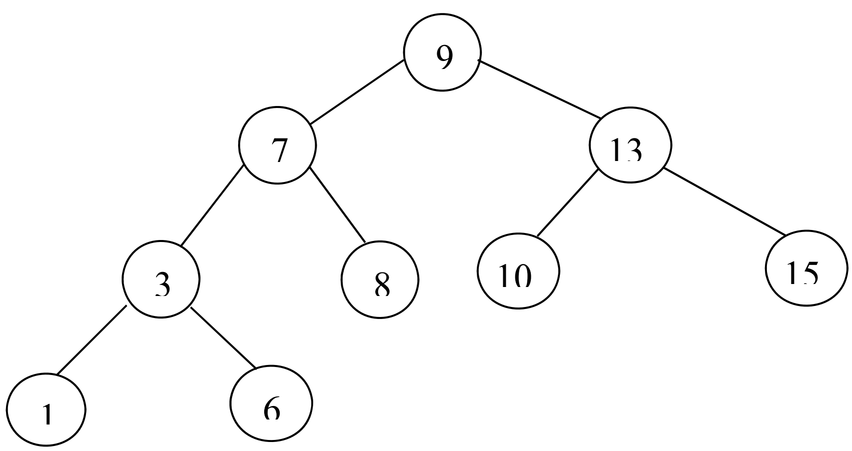
Q: What if we would like to balance the BST?

Building a Balanced BST:

Observe that for any given set of records S, the representation of S using BST is not unique. However, if all the keys are distinct, 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.
左中右

A complete BST T for S:



To Sort, Or Not To Sort:

Given a set of n records S with keys $x_1, x_2, \ldots, x_n$, $x_1 \le x_2 \le \ldots \le x_n$.

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

Q: If m searches are to be performed, m >> 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.

$$T_a(n) = O(\lg n).$$

Q: Is this the best way to organize the records in S so as to minimize the average search time?

Remark: This approach does not make use of 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.

$$T_a(n) = \sum_{i=1}^{n} i * p_i.$$

Q: Is this the best way to organize the records in S so as to minimize the average search time?

Remark: This approach does not make use of any of the given information on x_i.

Approach 3:

Store $\{x_1, x_2, \ldots, x_n\}$ in a BST T.

Apply BST operation search(T,x) to T.

$$T_a(n) = O(\lg n),$$

Remark: Which BST should we use? Also, this approach still does not make use of 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.

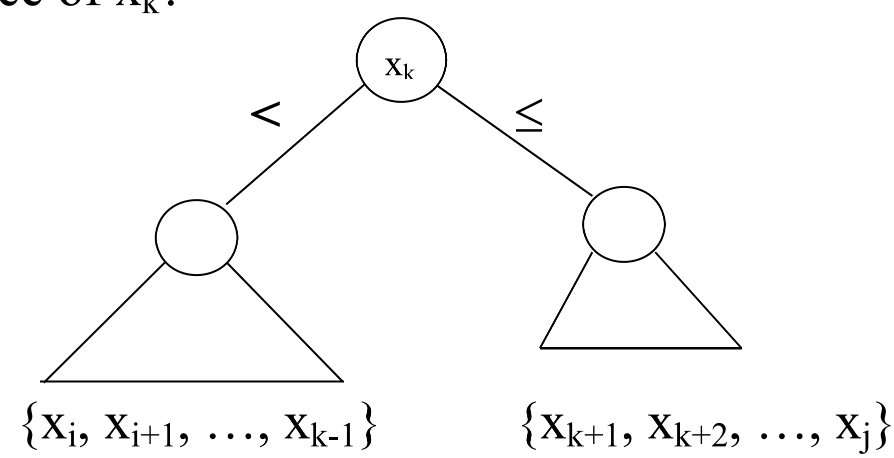
Optimal BST Problem:

Let T be any BST representing S with n objects.

$$T_a(n) = \sum_{i=1}^{n} p_i * [d(x_i) + 1], \text{ where } d(x_i) \text{ is the depth of } x_i.$$

Observe that in constructing a 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 $x_k >$ every element in $\{x_i, x_{i+1}, \ldots, x_{k-1}\}$ and $x_k \leq$ 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)!

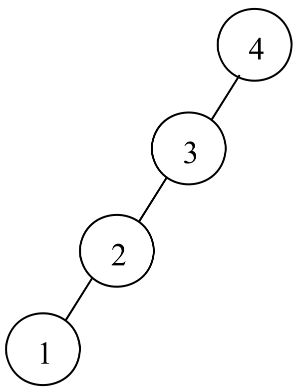
Q: Is it optimal?

it is not optimal, disadvantage!!!

Example: A greedy BST.

Consider <x_1, x_2, x_3, x_4> with $p_1 = 0.1, p_2 = 0.2, p_3 = 0.3, p_4 = 0.4.$

Level: depth



1

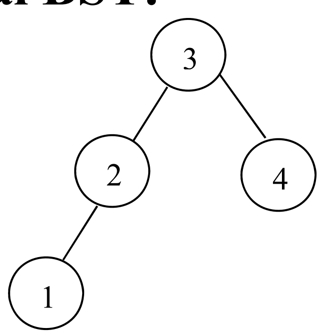
2

3

4

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

An Optimal BST:



$T_a(n) = 1*0.3 + 2*0.2 + 2*0.4 + 3*0.1 = 1.8.$

Computing Optimal 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,

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

Or,

Important here, left min: i~k-1; right min: k+1~j

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

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;

    sum = 0; // computing sum($p_i,\ldots,p_j$)

    for l = i to j do

      sum = sum + p_l;

    endfor;

    $c_{i,j} = \min_{i \le k \le j}\{c_{i,k-1} + c_{k+1,j}\} + sum$

    $t_{i,j} = k;$

  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 $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, $p_4 = 0.4$.

Ci+1,i = 0; C1,i-1 = 0

step 1 $c_{1,1} = 0.1, c_{2,2} = 0.2, c_{3,3} = 0.3, c_{4,4} = 0.4.$

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

min{C1,0+..,C1,1+…}, stop C1,2

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

right side means increase 1

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

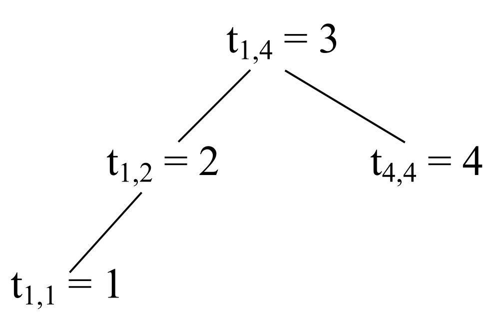
$$c_{1,3} = \min\{c_{1,0} + c_{2,3}, c_{1,1} + c_{3,3}, c_{1,2} + c_{4,3}\} + \sum_{l=1}^{3} p_l = \min\{0.7, 0.4, 0.4\} + 0.6 = 1.0, t_{1,3} = 2.$$

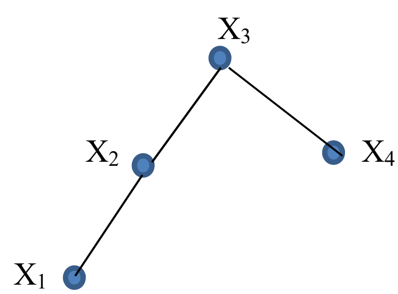
$$c_{2,4} = \min\{c_{2,1} + c_{3,4}, c_{2,2} + c_{4,4}, c_{2,3} + c_{5,4}\} + \sum_{l=2}^{4} p_l = \min\{1.0, 0.6, 0.7\} + 0.9 = 1.5, t_{2,4} = 3.$$

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

min{C1,0+..,C1,1+…,C1,2+…,C1,3+…} stop C1,4

Constructing Optimal BST:





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 all standard search tree operations all have $T_w(n) = O(\lg n)$?

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? Can we generalize it to k-ary search, k > 2?

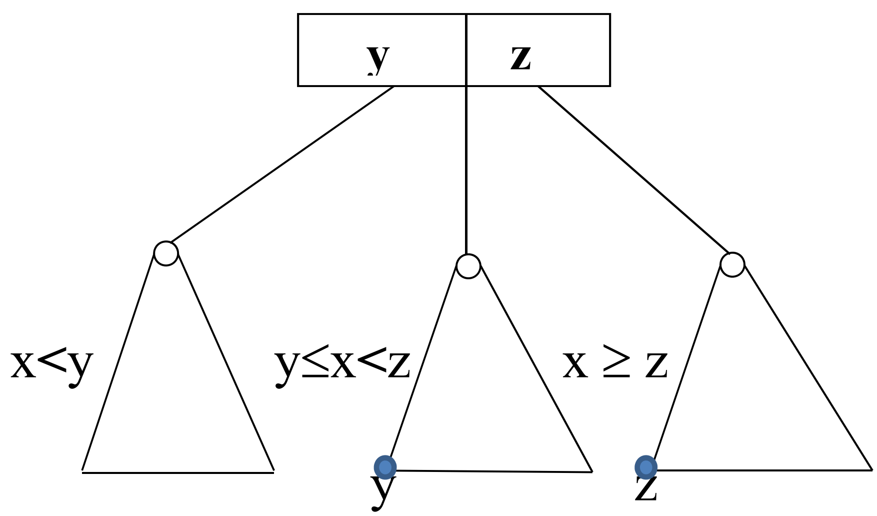
Consider 3-ary Search:

	s		j			k		t
A:		**…**	**y**	**…**	**…**	**z**	**…**	
			x:y			x:z		

$x < y$: 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! 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:



Nodes in 2-3 Tree:

1. Interior Node: Holding information to facilitate searching.
2. Leaf Node: Holding actual data record.

minSecond = y = info on min priority among all records from second subtree.

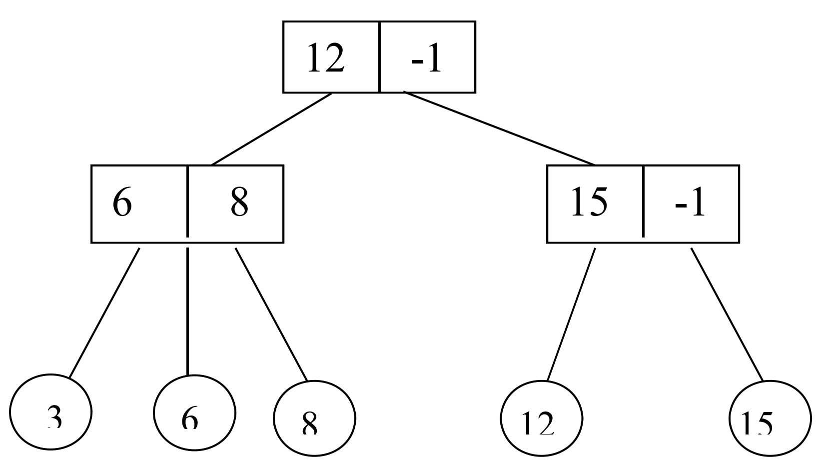
minThird = z = info on min priority among all records from third subtree if exists.

Characteristics of 2-3 Tree T:

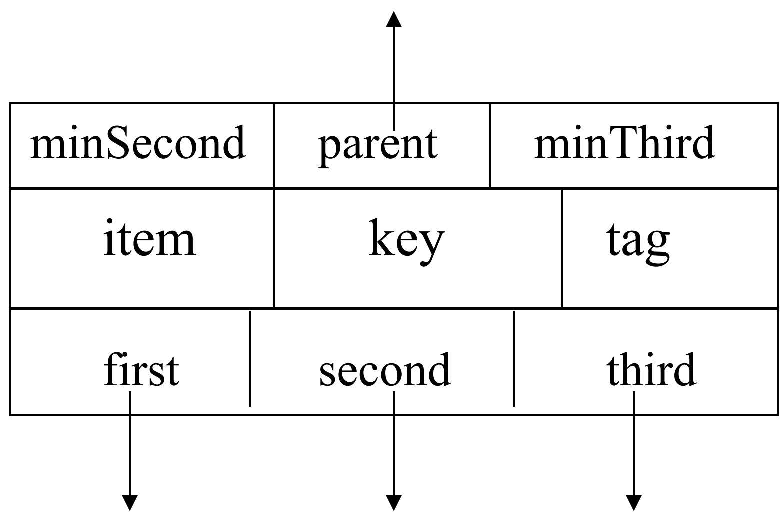
1. There are two types of nodes in T:
   Leaf nodes and *non-leaf (interior) nodes*.
2. All data elements are stored in the leaf nodes and they must be ordered from left (minimum) to right (maximum).
3. All leaf nodes must have the same depth.
4. Each interior node can either be a *2-node* with exactly 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:

Tag =0 interior node

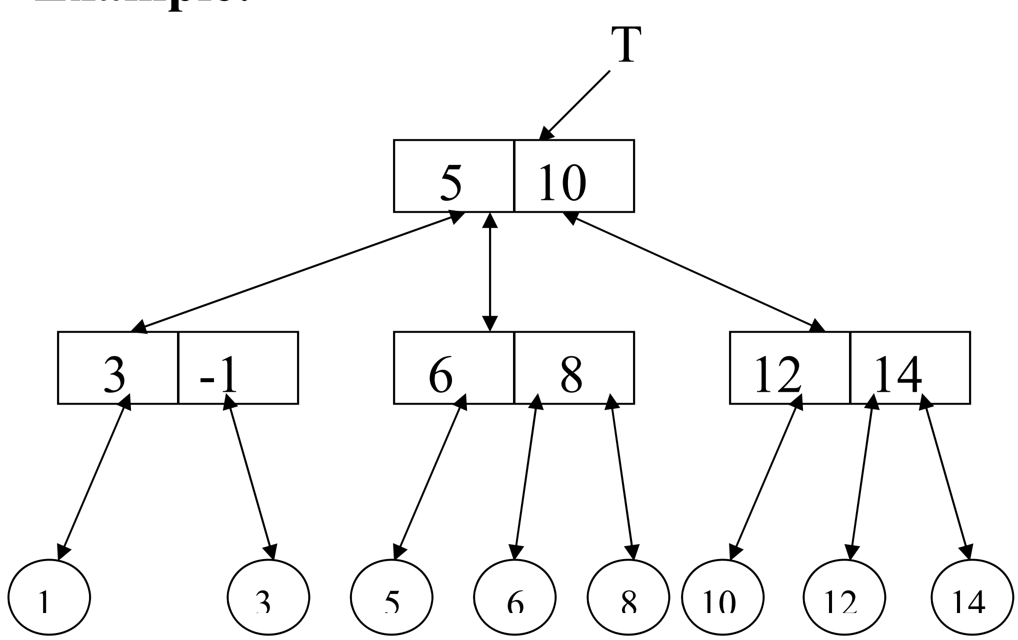


Node Structure:



tag = 0 $\Rightarrow$ interior node
tag = 1 $\Rightarrow$ leaf node

Example:



Typical 2-3 Tree Operations:
Find, Insert, FindMin, FindMax, DeleteMin, DeleteMax, Delete.

Consider **search(x,T)**.

```
if (T→tag == 1)                  // leaf node found
   then  return (x == T→key)
   else   temp = T;
          if (x < T→minSecond)
             then  return search(x,T→first)
             else  if (T→minThird != -1 and
                              x >= T→minThird)
                     then  search(x,T→third)
                     else  search(x,T→second)
                   endif;
         endif;
endif;
```

Complexity:
Search operation depends on height of 2-3tree. Since a 2-3 tree with n data objects has height h, $\log_2 n \le h \le \log_3 n$, search(x,T) has complexity $T_w(n) = O(\lg n)$.

Consider **insert(x,T):**

Case Analysis:

0: Create a new leaf node with x.
1. If T = 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) N1 and N2 with x inserted. Adjust N1 and N2.
      (i) If N was the root of T, create a new interior node, which becomes the new root of T, having children N1 and N2.
      (ii) If N was not the root of T, N must have a parent p(N). Attach N1 to p(N) as a child and then insert N2 to p(N) as before.

Complexity:

$T_w(n) = O(\lg n)$.

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:

$T_w(n) = O(\lg n)$.

BuildTree using insert operations:

$T_w(n) = O(n\lg n)$.

9/8/18