Binary Search Trees

A *binary search tree* (BST) is a properly ordered binary tree, which means the nodes are associated with elements of a set such that the following BST property is satisfied:

For each node say *x*,

- every element associated with any node in the left subtree of node *x* is less than the element associated with *x*, and
- every element associated with any node in the right subtree of *x* is greater than the element associated with *x*.

Figure 1 shows a binary search tree. Notice that this tree is obtained by inserting the values 25, 3, 14, 18, 9, 26, 10, 45, 84, 89, 2 in that order, starting from an empty tree.

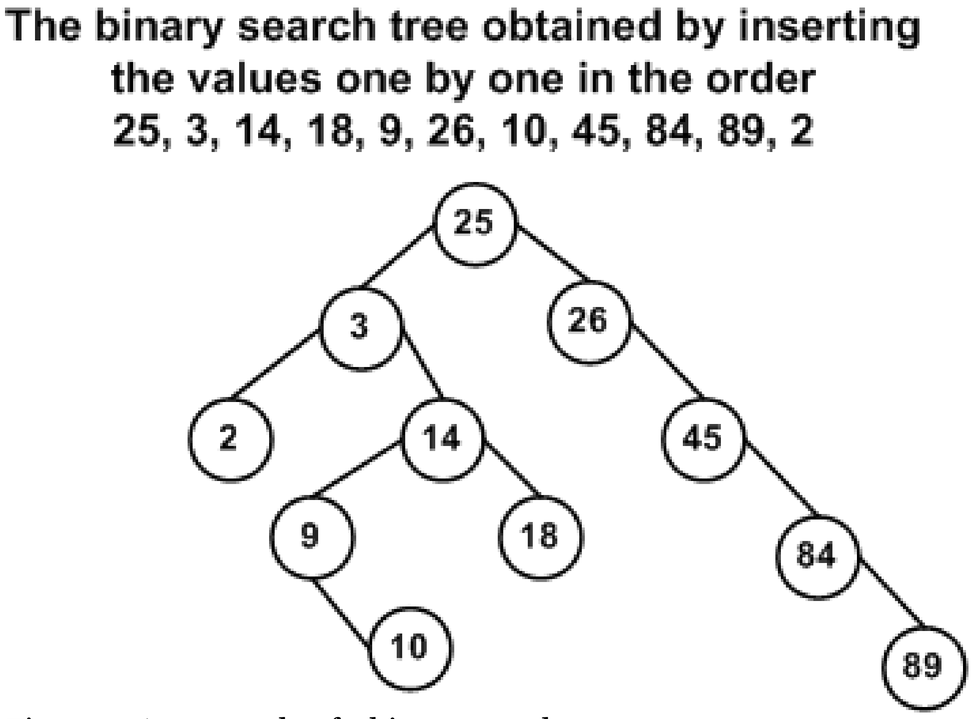

The binary search tree obtained by inserting the values one by one in the order 25, 3, 14, 18, 9, 26, 10, 45, 84, 89, 2
25
3
26
2
14
45
9
18
84
10
89

Figure 1: An example of a binary search tree

It is interesting to note that the inorder traversal of a binary search tree yields a sorted (in increasing order) sequence of the elements. This is an immediate consequence of the BST property. Thus a BST can be used to sort a given set of elements. To sort a given set, we first create a BST with these elements and then do an inorder traversal of the BST.

The following observations are also immediate from the BST property:

- The largest element in a BST can be found starting from the root node and repeatedly traversing along the right child until a node with no right child is found. (We can call the element associated with such a node, the *rightmost element* of the BST).

- The smallest element in a BST can be found by starting from the root node and repeatedly traversing along the left child until a node with no left child is found. (We can call the element associated with such a node, the *leftmost element* of the BST).

- A **search** for a given element in a BST can be accomplished as follows. Start with the root and keep moving left or right using the BST property. If the element we are seeking is present, this search procedure will lead us to the element. If the element is not present, we end up in a null link.

Insertion in a BST
Inserting a given element, say x, into a BST is a straight forward operation. We first search for the element x. If x is present, there is nothing to do. If x is not present, then our search procedure ends in a null link. It is at this position of this null link that x has to be inserted.

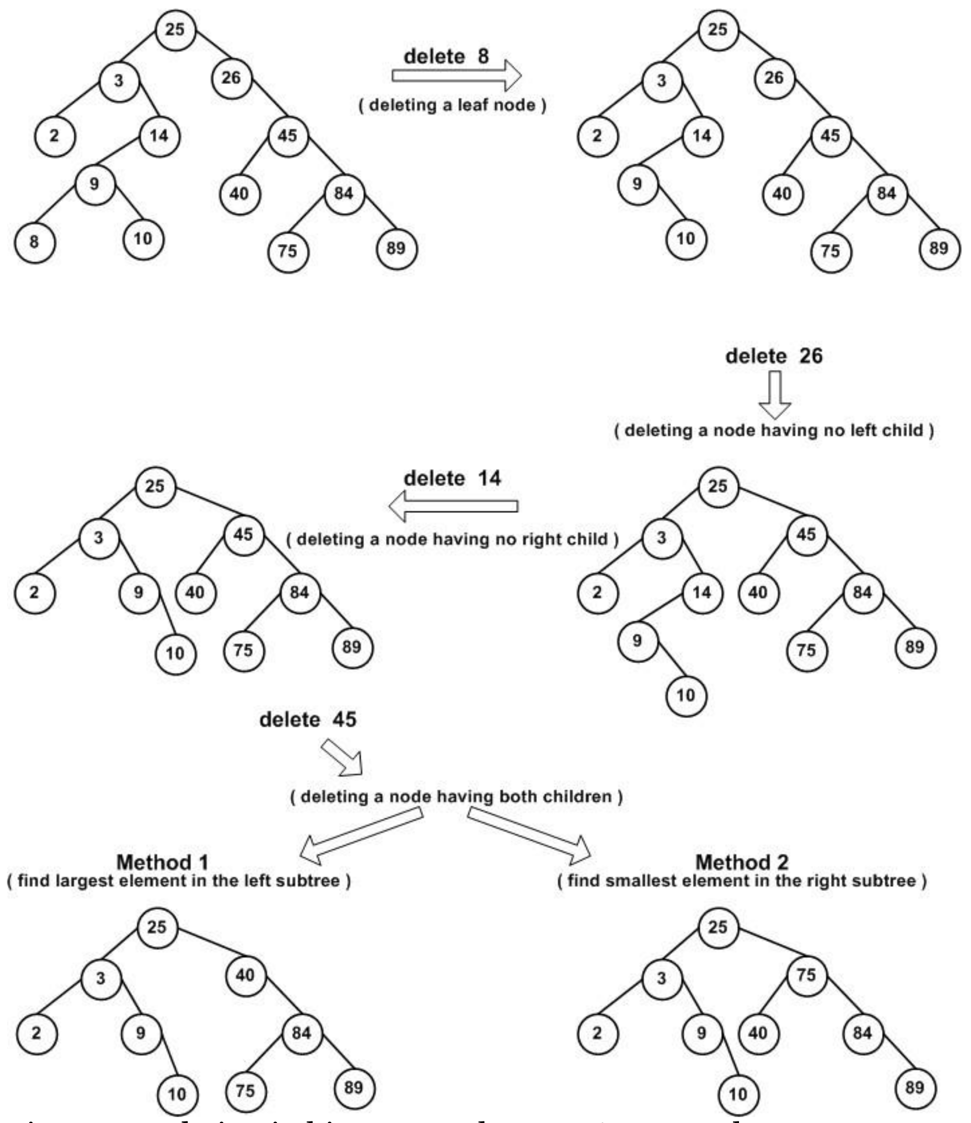
Deletion in a BST
Suppose that x is the specified element to be deleted from the BST. Let X be the node associated with the element x in the BST. We have three cases:

- **Case 1:** Node X is a leaf. In this case we simply delete the node X.

- **Case 2:** Node X has at most one child. Let C be the child node of node X. We have two subcases:
    - Suppose node X is the root node, then delete node X and make node C the root of the tree.
    - Suppose node X is not the root node and let P be the parent node of node X. We delete node X, and make node C the left or right child of node P, depending upon whether node X was the left or right child of P, respectively.

- **Case 3:** Node X has both, the left and right children. In this case we can use any one of the following two methods:
    - *Method 1:* Delete the node associated with the largest element, say w, among all descendants of the left child of X. Associate the element value w to node X. Note that in this case, the selected node for deletion will not have a right child; so this node can be deleted using Case 1 or 2, depending upon whether the node is a leaf or has only the left child.
    - *Method 2:* Delete the node associated with the smallest element, say w, among all descendants of the right child of X. Associate the element value w to node X. Note that in this case, the selected node for deletion will not have a left child; so this node can be deleted using Case 1 or 2, depending upon whether the node is a leaf or has only the right child.

Figure 2 illustrates all these cases of deletion in BSTs.



delete 8
(deleting a leaf node)
delete 26
(deleting a node having no left child)
delete 14
(deleting a node having no right child)
delete 45
(deleting a node having both children)
Method 1
(find largest element in the left subtree)
Method 2
(find smallest element in the right subtree)

Figure 2: Deletion in binary search trees: An example

Performance of Insert and Delete in a BST
If we repeatedly insert a sorted sequence of values to form a BST, we obtain a completely skewed BST. The height of such a tree is $n - 1$ if the tree has n nodes. Thus, the worst case complexity of searching or inserting an element into a BST having n nodes is n iterations.

However, the average case complexity of Search, Insert, and Delete operations in a BST is approximately *log n* iterations, where n is the number of nodes in the tree.

Dictionaries

A *dictionary* is a set of elements that supports the following operations:

- insert a given element into the set
- delete a specified element from the set
- search for a given element in the set

Dictionaries have several applications in the real world, and there are many ways of implementing them; such as hash tables, binary search trees, and height balanced search trees.

Hash Tables

A binary search tree supports operations such as inserting a new element, deleting a specified element, searching for a given element, and finding the maximum value (or equivalently a minimum value) element. In this section, we discuss the hash table, which supports some of the operations allowed by binary search trees, namely inserting a new element, deleting a specified element, and searching for a given element.

Implementation of hash tables is frequently called hashing. Hashing is an extremely effective way of implementing dictionaries. In the average case, the search, insert, and delete operations can be accomplished in constant time. In the worst case these operations take *n* iterations. By careful design, we can make the probability that more than constant time is required to be arbitrarily small.

A hash table data structure can be viewed as fixed number (say B) of buckets arranged in a sequence such as {0, 1, . . . , B − 1}. These buckets contain the elements of the set. In general, a search is performed on some part of the element. This is called the key. For instance, an element could consist of a name, phone number, salary, and other data items. We may want to conduct our search using element's name as the key. Each key is mapped into some number in the range 0 to B − 1 and placed in the appropriate bucket. The mapping is called a hash function, which ideally should ensure that any two distinct keys get different buckets. Since there are a fixed number of buckets and in general, the set of elements (keys) are large, we need a hash function that distributes the keys evenly among the buckets.

There are two ways of implementing the hash tables.

- Open or External
- Closed or Internal

Open Hashing

Let:

- U be the set of keys:
  - integers

    - character strings
    - complex bit patterns
- Γ be the set of hash values (also called the buckets or bins). Let $\Gamma = \{0, 1, \ldots, B - 1\}$ where $B > 0$ is a positive integer.

A hash function $h : U \rightarrow \Gamma$ associates buckets (hash values) to keys. Two main issues:

1. Collisions
   If x1 and x2 are two different keys, it is possible that h(x1) = h(x2). This is called a collision. Collision resolution is the most important issue in hash table implementations.
2. Hash Functions
   Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue. Choosing a good hash function is outside the scope of this course.

Collision Resolution by Chaining

- Put all the elements that hash to the same value in a linked list. See Figure 1.

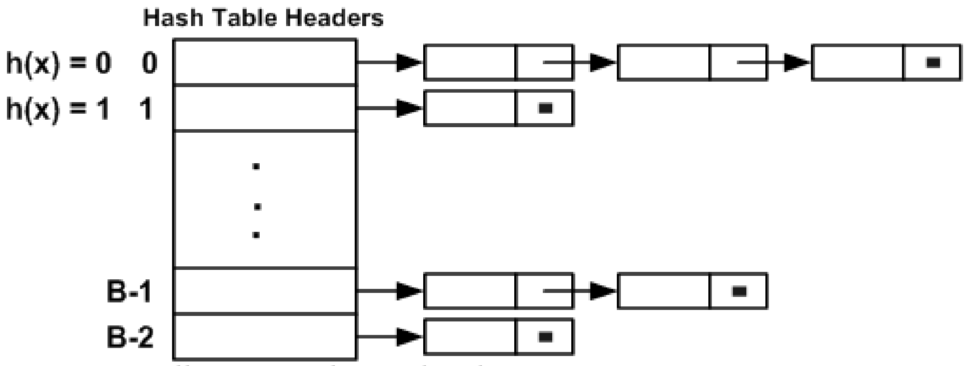


Figure 1: Collision resolution by chaining

Suppose we use the hash function h(x) = x%7 and insert the keys 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 into an open hash table. The resulting table is shown in Figure 2.

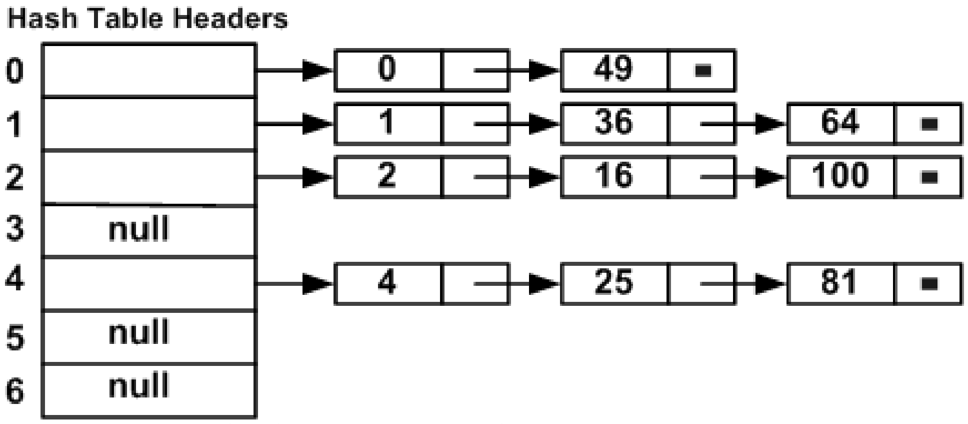


Figure 2: Open hashing: An example

Worst case complexity of Search, Insert, and Delete operations is when we have 100% collisions, resulting in *n* iterations, where n is the number of keys in the hash table.

Closed Hashing

- All elements are stored in the hash table itself
- Collisions are handled by generating a sequence of *rehash* values.

$$h : \underbrace{U}_{\text{universe of primary keys}} \times \underbrace{\{0,1,2,...\}}_{\text{probe number}} \rightarrow \{0,1,2,...,B-1\}$$

- Given a key x, it has a hash value h(x,0) and a set of rehash values

  h(x, 1), h(x,2), . . . , h(x, B-1)

- We require that for every key x, the probe sequence

  < h(x,0), h(x, 1), h(x,2), . . . , h(x, B-1)>

  be a permutation of <0, 1, ..., B-1>.

  This ensures that every hash table position is eventually considered as a slot for storing a record with a key value x.

Search (x)

- Search will continue until you find the element x (successful search) or an empty slot (unsuccessful search).

Delete (x)

- No delete if the search is unsuccessful.
- If the search is successful, then put the label DELETED (different from an empty slot).

Insert (x)

- No need to insert if the search is successful.
- If the search is unsuccessful, insert at the first position with a DELETED tag.

Rehashing Methods

Denote h(x, 0) by simply h(x).

1. Linear probing
   $h(x, i) = (h(x) + i) \% B$
2. Quadratic probing
   $h(x, i) = (h(x) + C_1 i + C_2 i_2) \% B$
   where C_1 and C_2 are constants

An Example of Linear Probing:

Assume linear probing with the following hashing and rehashing functions:

$$\begin{aligned} h(x,0) &= x\%7 \\ h(x,i) &= (h(x,0)+i)\%7 \end{aligned}$$

Start with an empty table.

Insert (20)
Insert (30)
Insert (9)
Insert (45)
Insert (14)

0	14
1	empty
2	30
3	9
4	45
5	empty
6	20

Search (35)
Delete (9)

0	14
1	empty
2	30
3	deleted
4	45
5	empty
6	20

Search (45)
Search (52)
Search (9)
Insert (45)
Insert (10)

0	14
1	empty
2	30
3	10
4	45
5	empty
6	20

Delete (45)
Insert (16)

0	14
1	empty
2	30
3	10
4	16
5	empty
6	20

Hashing Functions

What is a good hash function?

- The one that distributes the keys evenly (uniformly) among the buckets. That is, each bucket is equally likely to be occupied.
- A common approach is to derive a hash value in a way that is expected to be independent of any patterns that might exist in the data.

- The division method computes the hash value as the remainder when the key is divided by a prime number. In general, this method gives okay results.

A key is mapped into one of m slots using the function

h(k) = k % m

This method requires only a single division, hence it is fast. The integer m should not be a power of 2, since if $m = 2^p$, then h(k) is just the p lowest order bits of k. In general, good values for m are prime numbers.