Binary Search Trees

Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

6 Nov, 2017

Outline

A motivating example

Google may want to store all taken email ids, and quickly tell a new user whether her choice of email id is available or not. There are around 500 million user ids.

- Need to support both quick additions as well as membership queries.

Naive algorithm: Store entries in an array. What is the running time of the queries “add” and “is-member”?

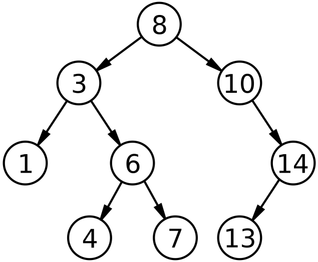
Term frequency in a document [Kernighan and Ritchie]

Given a text document, find the set of words that occur in the document and the count of the number of times each word occurs.

> *Now is the time for all good men to come to the aid of their party ...*

Using an array to store words as they are encountered in the text, takes time quadratic in the size of (number of words in) the doucment.

Binary Search Trees

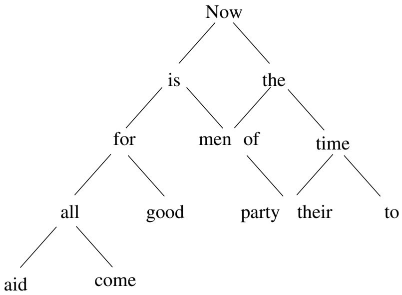


Can support *add* and *is-member* in $O(\log n)$ worst-case time.

- Add and search is proportional to the height of the tree.
- Uses idea of keeping the tree balanced, so that height is $\log n$.
- log(500 million) is about 29.

Document search using BST

Now is the time for all good men to come to the aid of their party ...



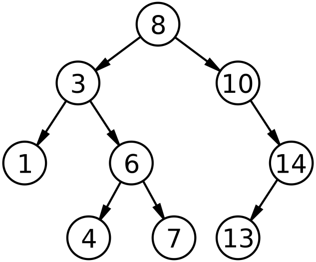
- Document indexing would take $O(n \cdot \log n)$ time rather than $O(n^2)$.

Binary Search Tree

- A binary search tree data-structure is a binary tree in which each node has a "key" value associated with it
- The tree satisfies the "search tree property:"

  The key values in the left subtree of a node are at most the key value of the node, which in turn is at most the key values of the nodes in the right subtree.

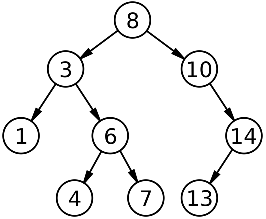
- The height of a tree T, denoted $h(T)$, is the number of edges on the longest path from root to leaf.

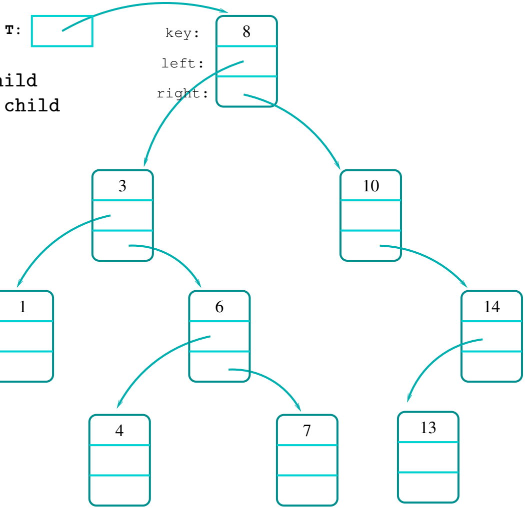


Implementing a BST

```
struct node {
  int key;  // key value
  struct node *left;  // left child
  struct node *right;  // right child
  struct node *p;  // parent
}

struct bst {
  struct node *root;
} *T; // BST T
```

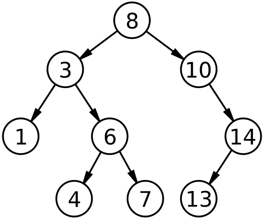




Exercise

Write a C program that creates the BST alongside.

```
void main() {
  struct bst T;
  struct node n1, n2, ..., n9;
  ...
}
```



Operations supported by a BST

A binary search tree supports the following operations in worst-case time of $O(\log h(T))$:

- SEARCH (retrieve a given key from the tree).
- INSERT (insert a given key value in the tree).
- DELETE (insert a given key value in the tree).

Additional operations include

- PRINT-KEYS (Print out keys in sorted order).
- MIN (Return minimum key value in the tree).
- MAX (Return maximum key value in the tree).
- SUCCESSOR (successor key value of a given key).
- PREDECESSOR (predecessor key value of a given key).

Search – Iterative version

Given a pointer x to a node in a BST, and a key value k, return a pointer to a node with key k if it exists in the subtree of x, else NULL.

```
Search(x, k) {
  while (x != NULL and x.key != k) {
    if (k < x.key)
      x := x.left;
    else
      x := x.right;
  }
  return x;
}
```

Search – Recursive version

Given a pointer x to a node in a BST, and a key value k, return a pointer to a node with key k if it exists in the subtree of x, else NULL.

```
Search(x, k) { // x is ptr to a node
  if (x = NULL or x.key = k)
    return x;
  if (k < x.key)
    return Search(x.left, k);
  else
    return Search(x.right, k);
}
```

Exercise

Write a function

```
struct node* minimum(struct node *x) {
  ...
}
```

which returns a pointer to a node with the smallest key value in the tree rooted at x.

Printing in ascending order

```
Print(T) {
  Inorder-Print(T.root);
}

Inorder-Print(x) {
  if (x != NULL) {
    Inorder-Print(x.left);
    print(x.key);
    Inorder-Print(x.right);
  }
}
```

Insert

```
Insert(T, z) { // z is pointer to node to be inserted
  x := T.root;
  y := NULL;
  while (x != NULL) {
    y := x;
    if (z.key < x.key)
      x := x.left;
    else
      x := x.right;
  }
  if (y = NULL)
    T.root := z;
  elseif (z.key < y.key)
    y.left := z;
  else
    y.right := z;
  z.p := y;
}
```

Successor

Given a node x, return (a pointer to) the node u such that $x.key \leq u.key$ and $u.key$ is the smallest such value.

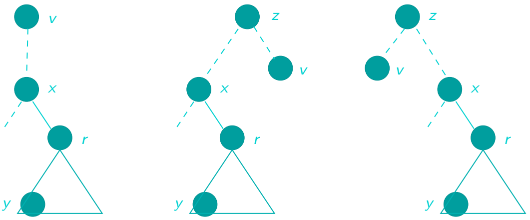
```
Successor(x) {  // x is the node whose succ is to be returned
  if (x.right != NULL)
    return Minimum(x.right);
  y := x.p;
  while (y != NULL and x = y.right) {
    x := y;
    y := y.p;
  }
  return y;
}
```

Correctness of Successor

We assume that keys in the given tree are distinct.

Claim 1: If x has a non-empty right subtree, then minimum node y in this right subtree is the successor of x.
Argue that:

- If v is an ancestor of x, it cannot be successor.
- If v is not an ancestor of x, consider the least common ancestor of x and v, say z, and argue that v cannot be successor of x.

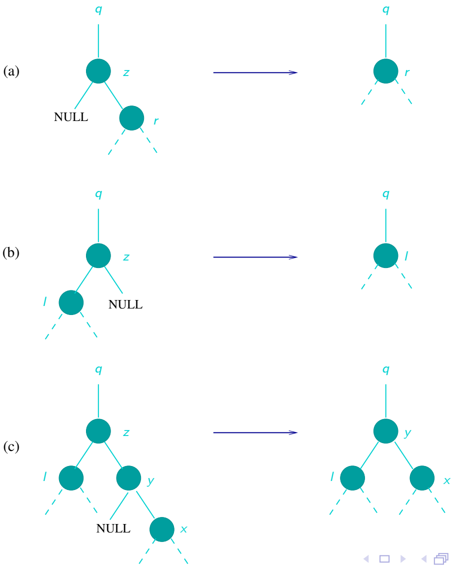


Correctness of Successor

Similarly:
Claim 2: If x has an empty right subtree, then the smallest ancestor u of x such that x lies in u's left subtree, is the successor of x.
Argue that:

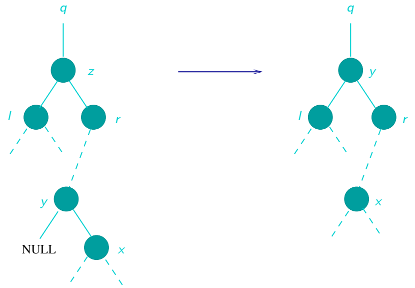
- If v is an ancestor of u, it cannot be successor.
- If v is an ancestor of x but not of u, it cannot be successor.
- If v is not an ancestor of x, consider the least common ancestor of x and u, say z, and argue that v cannot be successor of x.

Delete operation: cases



Delete operation: cases

(d)



Delete

```
Delete(T, z) {  // z is pointer to node to be deleted
  if (z.left = NULL)
    Transplant(T, z, z.right);
  elseif (z.right = NULL);
    Transplant(T, z, z.left);
  else {
    y := Minimum(z.right);
    if (y.p != z) {
      // y not child of z
      Transplant(T, y, y.right);
      y.right = z.right;
      (y.right).p = y
    }
    Transplant(T, z, y)
    y.left = z.left;
    (y.left).p = y;
  }
}
```

```
Transplant(T, u, v) {
  // transplant subtree at u
  // by subtree at v
  if (u.p = NULL)
    T.root = v;
  elseif (u = (u.p).left);
    (u.p).left = v;
  else
    (u.p).right = v;
  if (v != NULL)
    v.p = u.p;
}
```

Balancing BST's

- In general the BST may not be "balanced", leading to height that could be $O(n)$ in the worst case.
- There are some popular schemes for maintaining the BST in a balanced form:
  - Red-Black trees: Nodes have either red or black colour, root and leaves are black, red nodes have black children, and paths from root to leaf have same number of black nodes.
  - AVL trees: for each node, height of its left and right subtrees differ by at most 1.