Contents

Section outline

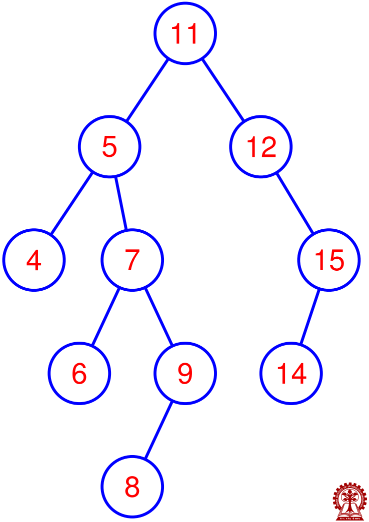
1. **Binary search trees (BST)**
   - BST definition
   - Searching in a BST
   - Inserting in a BST
   - Deletion from a BST
   - Average case searching time in a BST

BST definition

Definition (Binary search tree)

It is a binary tree having a key in each node and satisfying the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- Both the left and right subtrees must also be binary search trees

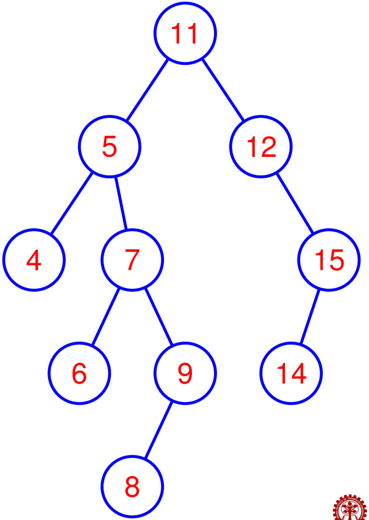


Invented by P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard in 1960

BST typedef

```
typedef int keyTyp;

typedef struct binTTag {
  keyTyp key;
  struct binTTag
    *lChild, *rChild;
} binTreeTyp, *binTreePtr;
```



Searching in a BST

binTreePtr *bstSearch(binTreePtr bstP, keyTyp ky)

Base cases

CBa bstP==NULL // empty BST

ABa return NULL;

CBb bstP->key == ky // key found

ABb return bstP;

Searching in a BST

binTreePtr *bstSearch(binTreePtr bstP, keyTyp ky)

	Base cases
CBa	bstP==NULL // empty BST
ABa	return NULL;
CBb	bstP->key == ky // key found
ABb	return bstP;
	Inductive/recursive case
CIa	ky<bstP->key // check in left subtree
AIa1	return bstSearch(bsTP->lChild, ky);
CIb	ky>bstP->key // check in right subtree
AIb1	return bstSearch(bsTP->rChild, ky);

Worst case time complexity for a BST with n keys is $O(n)$, when the tree is skewed

Programming friendly searching in BST

binTreePtr *bstSearch(binTreePtr *bstPP, keyTyp ky)

Base cases

CBa *bstPP==NULL // empty BST

ABa return bstPP; // caller checks whether *bstPP==NULL

CBb (*bstPP)->key == ky // key found

ABb return bstPP;

Programming friendly searching in BST

binTreePtr *bstSearch(binTreePtr *bstPP, keyTyp ky)

	Base cases
CBa	*bstPP==NULL // empty BST
ABa	return bstPP; // caller checks whether *bstPP==NULL
CBb	(*bstPP)->key == ky // key found
ABb	return bstPP;
	Inductive/recursive case
CIa	ky<(*bstPP)->key // check in left subtree
AIa1	return bstSearch(&((*bstPP)->lChild), ky);
CIb	ky>(*bstPP)->key // check in right subtree
AIb1	return bstSearch(&((*bstPP)->rChild), ky);

Use pointer to the pointer to the BST, through which a new node can be inserted or an existing node deleted

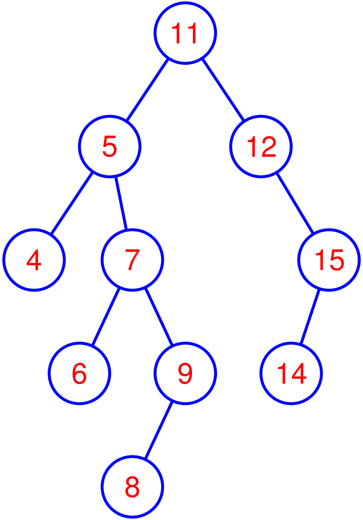
Inserting in a BST

```
binTreePtr *bstIns(binTreePtr *bstPP, keyTyp ky) {
  // first locate the node or
  // the point where the search fails
  binTreePtr *bstSrchP = bstSearch(bstPP, ky);
  if (*bstSrchP==NULL) {
    *bstSrchP = binTreeLeafNode(ky);
  } // else ky is already present
}

binTreePtr binTreeLeafNode(ky) {
  binTreePtr nP = malloc(sizeof(binTreeTyp));
  nP->ky = ky; nP->lChild = nP->rChild = NULL;
}
```

Worst case time complexity is determined by that of bstSearch()

Deletion from a BST



First locate the node containing the key to be deleted, then continue as follows:

Node is a leaf? Delete directly

Node has only one child? Delete, moving the subtree in its place

Node has both children? – cannot be deleted simply

1. Identify the predecessor or successor of the node in the subtree rooted at that node
2. Replace the node with the identified predecessor or successor, deleting that

Deletion from a BST (contd.)

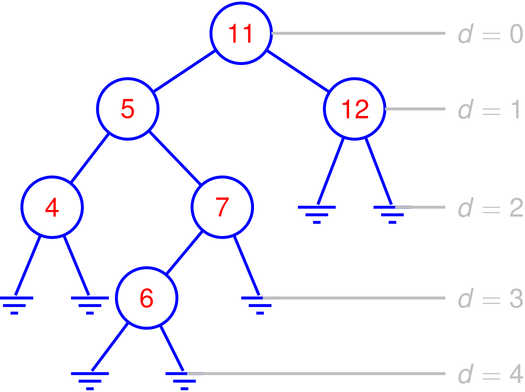
```
binTreePtr bstFndDelPred
      (bstTreePtr nP) {
// find and delete the
// predecessor of node in the
// subtree rooted at it
   binTreePtr *nPP =
      &(nP->lChild);
   while (*nPP->rChild) {
      nPP = &(*nPP->rChild);
   } // keep going right in loop
   nP = *nPP; // the pred node
   // now short circuit pred node
   *nPP = nP->lChild;
   nP->lChild = NULL;
   return nP; // with pred ky
}
```

Deletion from a BST (contd.)

```
binTreePtr bstFndDelPred
      (bstTreePtr nP) {
// find and delete the
// predecessor of node in the
// subtree rooted at it
   binTreePtr *nPP =
      &(nP->lChild);
   while (*nPP->rChild) {
      nPP = &(*nPP->rChild);
   } // keep going right in loop
   nP = *nPP; // the pred node
   // now short circuit pred node
   *nPP = nP->lChild;
   nP->lChild = NULL;
   return nP; // with pred ky
}
```

```
binTreePtr *bstDel(binTreePtr *bstPP,
     keyTyp ky) { // try locating the key
  binTreePtr pP, *kyNodePP =
     bstSearch(bstPP, ky);
  if (*kyNodePP == NULL) return; // absent
  if ((*kyNodePP)->rChild == NULL) {
     pP = *kyNodePP; // rChild absent
     *kyNodePP = pP->lChild;
  } else if ((*kyNodePP)->lChild == NULL) {
     pP = *kyNodePP; // lChild absent
     *kyNodePP = pP->rChild;
  } else { // both l&r child, so pred ky to del ky
     pP = bstFndDelPred
     (*kyNodePP);
     (*kyNodePP)->ky = pP->ky;
  }
  free(pP); // del ky or vacated pred ky
}
```

Average case searching time in a BST



- Let T be *any* BST with n nodes
- $|\text{NULLs}| = 1 + |\text{BSTNodes}|$
- $I_n \equiv$ sum of levels of all BSTNodes
- $E_n \equiv$ sum of levels of all NULLs

- $n = 6$, $|\text{NULLs}| = n + 1 = 7$, $I = 0+1\times 2+2\times 2+3\times 1 = 9$, $E = 2\times 2 + 3\times 3 + 4\times 2 = 21 = I + 2n$
- $E_n = I_n + 2n$ (5.1)
- $S_n \equiv$ average #comparisons for successful search
- $U_n \equiv$ average #comparisons for unsuccessful search
- Now, $U_n = \frac{E_n}{n+1}$ (5.2)
- Also, $S_n = \frac{I_n+n}{n} = \frac{E_n-n}{n}$ [by (5.1)]
- There are $n\,!$ equi-probable insertion sequences

- Consider each such sequence $\{x_1, x_2, \ldots, x_n\}$
- Denote by $T_\imath$ $(1 \leq \imath \leq n)$ the partial BST with $x_1, x_2, \ldots, x_\imath$ only
- Let $X_\imath \equiv$ number of comparisons for successful search of $x_\imath$ in $T_\imath$
- By definition, $U_{\imath-1} \equiv$ average number of comparisons for unsuccessful search of $x_\imath$ in $T_{\imath-1}$
- The unsuccessful search of $x_\imath$ in $T_{\imath-1}$ ends at the parent of $x_\imath$ in $T_\imath$
- Hence,

$$X_\imath = 1 + U_{\imath-1} \tag{5.3}$$

- From (5.3),

$$S_n = \frac{1}{n}\sum_{\imath=1}^{n} X_\imath = 1 + \frac{1}{n}\sum_{i=0}^{n-1} U_\imath \Rightarrow nS_n = n + \sum_{i=0}^{n-1} U_\imath \tag{5.4}$$

- From (5.2),

$$(n+1)U_n = E_n,\ nS_n = E_n - n \Rightarrow (n+1)U_n = nS_n + n \tag{5.5}$$

- So, from (5.4),

$$(n+1)U_n = 2n + \sum_{\imath=0}^{n-1} U_\imath$$

$$\Rightarrow \quad nU_{n-1} = 2(n-1) + \sum_{\imath=0}^{n-2} U_\imath \quad \text{[replacing } n \text{ by } n-1\text{]}$$

$$\Rightarrow (n+1)U_n - nU_{n-1} = 2 + U_{n-1} \quad \text{[subtracting 2nd from the 1st]}$$

$$\Rightarrow \quad U_n = U_{n-1} + \frac{2}{n+1}$$

- With $U_1 = 1$, unfold the above recurrence to get

$U_2 = 1 + \frac{2}{3}$,

$U_3 = 1 + \frac{2}{3} + \frac{2}{4}$,

$U_4 = 1 + \frac{2}{3} + \frac{2}{4} + \frac{2}{5}$,

$\vdots$

$U_n = 1 + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n+1} = 2\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}\right) - 2$

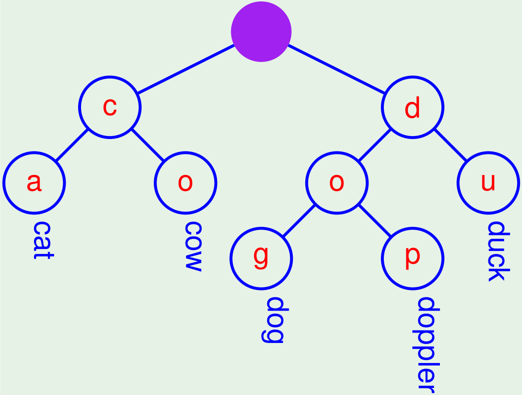
- $U_n = 2H_{n+1} - 2 \in O(\log n)$
- From (5.5),
$$S_n = \left(1 + \frac{1}{n}\right) U_n - 1 \in O(\log n)$$
- Even though the worst case search time is linear, the average search time is logarithmic

Section outline

2. **Prefix trees (Tries)**
   - Efficient key retrieval
   - Storage mechanism

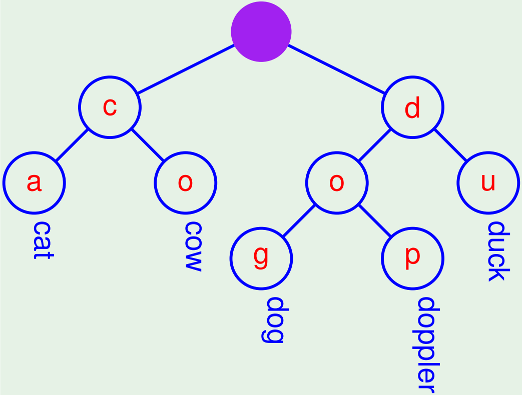
Efficient key retrieval

Example (Storing keys using tries)



Efficient key retrieval

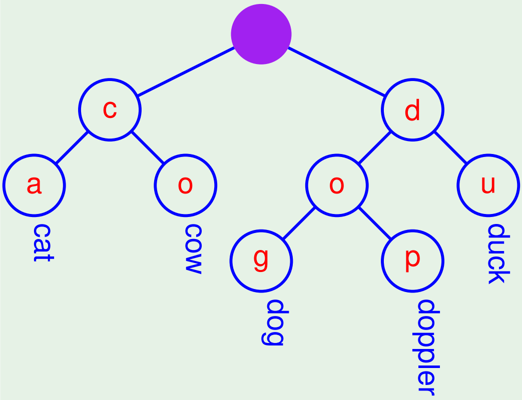
Example (Storing keys using tries)



- *Trie* derived from retrieval
- Not all words are at leaves: cat, cataclysm, cataclysmic

Efficient key retrieval

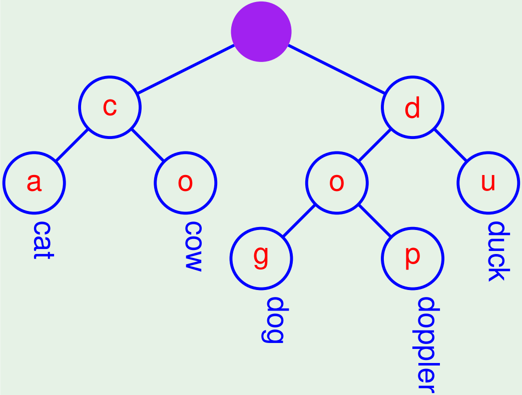
Example (Storing keys using tries)



- *Trie* derived from retrieval
- Not all words are at leaves: cat, cataclysm, cataclysmic
- Initially, one letter is enough
- Branching needed after divergence from common prefix
- Recognition of key may happen after branching

Efficient key retrieval

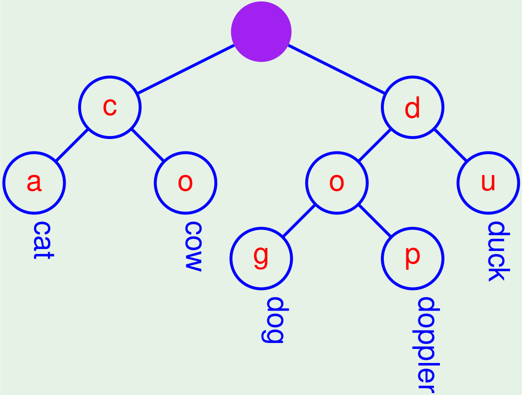
Example (Storing keys using tries)



- *Trie* derived from retrieval
- Not all words are at leaves: cat, cataclysm, cataclysmic
- Initially, one letter is enough
- Branching needed after divergence from common prefix
- Recognition of key may happen after branching
- Insertion, search, delete key of length k: $O(k)$

Efficient key retrieval

Example (Storing keys using tries)



- *Trie* derived from retrieval
- Not all words are at leaves: cat, cataclysm, cataclysmic
- Initially, one letter is enough
- Branching needed after divergence from common prefix
- Recognition of key may happen after branching
- Insertion, search, delete key of length k: $O(k)$
- Independent of keys stored (n) and may be better than $\lg n$

Storage mechanism

Example (Storing keys using tries)