

Searching: Binary Trees and Hash Tables

Chapter 12

Chapter Contents

12.1 Review of Linear Search and Binary Search

12.2 Introduction to Binary Trees

12.3 Binary Trees as Recursive Data Structures

12.4 Binary Search Trees

12.5 Case Study: Validating Computer Logins

12.6 Threaded Binary Search Trees (Optional)

12.7 Hash Tables

Chapter Objectives

- Look at ways to search collections of data
 - Begin with review of linear and binary search
- Introduce trees in general and then focus on binary trees, looking at some of their applications and implementations
- See how binary trees can be viewed as recursive data structures and how this simplifies algorithms for some of the basic operations
- Develop a class to implement binary search trees using linked storage structure for the data items
- (Optional) Look briefly at how threaded binary search trees facilitate traversals
- Introduce the basic features of hash tables and examine some of the ways they are implemented

Linear Search

- Collection of data items to be searched is organized in a list

$$X_1, X_2, \dots X_n$$

- Assume $=$ and $<$ operators defined for the type
- Linear search begins with item 1
 - continue through the list until target found
 - or reach end of list

Linear Search

Vector based search function

```
template <typename t>
void LinearSearch (const vector<t> &v,
                  const t &item,
                  boolean &found, int &loc)
{ found = false;  loc = 0;
  for ( ; ; )
  {   if (found || loc == v.size()) return;
      if (item == x[loc])  found = true;
      else loc++;  }
}
```

Linear Search

Singly-linked list based search function

```
template <typename t>
void LinearSearch (NodePointer first,
                  const t &item,
                  boolean &found, int &loc)
{ found = false;  locptr = first;
  for ( ; ; )
  {   if (found || locptr ==) return;
      if (item == locptr->data)
          found = true;
      else
          loc++; }
}
```

In both cases, the worst case computing time is still $O(n)$

Binary Search

Binary search function for vector

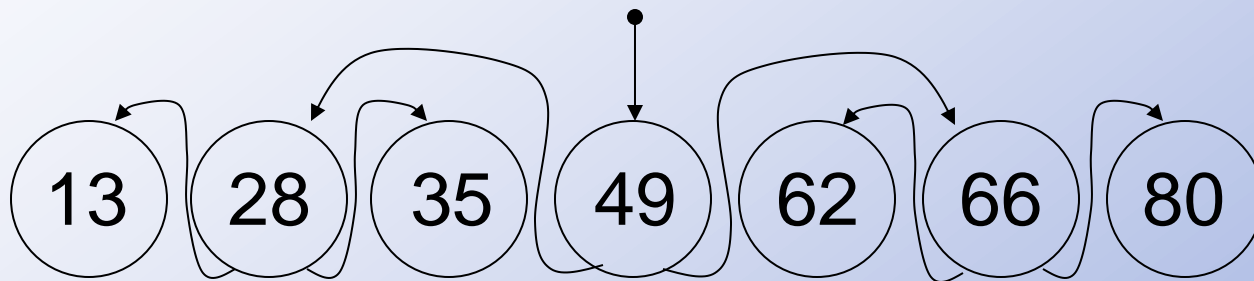
```
template <typename t>
void LinearSearch (const vector<t> &v,
                  const t &item,
                  boolean &found, int &loc)
{
    found = false;
    int first = 0;    last = v.size() - 1;
    for ( ; ; )
    {
        if (found || first > last) return;
        loc = (first + last) / 2;
        if (item < v[loc])
            last = loc + 1;
        else if (item > v[loc])
            first = loc + 1;
        else /* item == v[loc] */ found = true;
    }
}
```

Binary Search

- Usually outperforms a linear search
- Disadvantage:
 - Requires a sequential storage
 - Not appropriate for linked lists (Why?)
- It is possible to use a linked structure which can be searched in a binary-like manner

Binary Search Tree

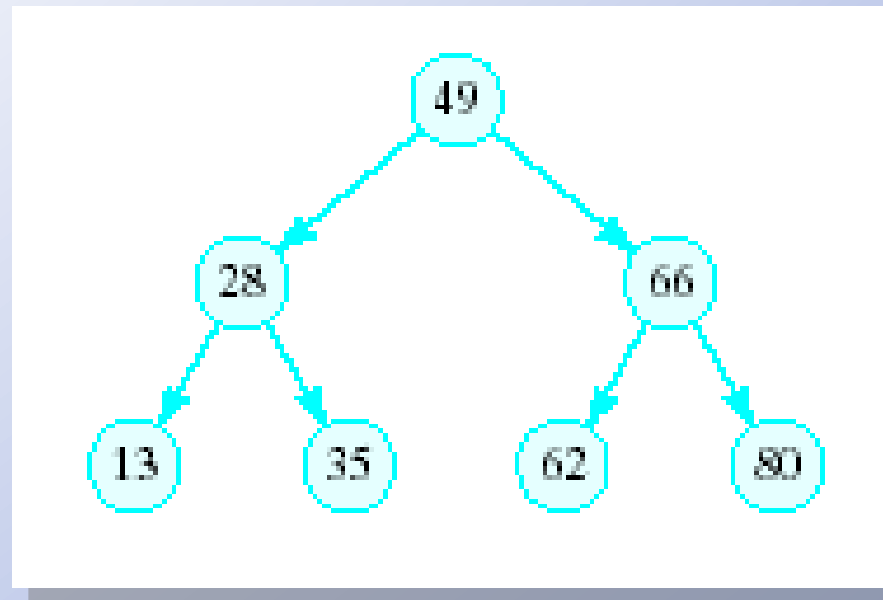
- Consider the following ordered list of integers



1. Examine middle element
2. Examine left, right sublist (maintain pointers)
3. (Recursively) examine left, right sublists

Binary Search Tree

- Redraw the previous structure so that it has a treelike shape – a binary tree

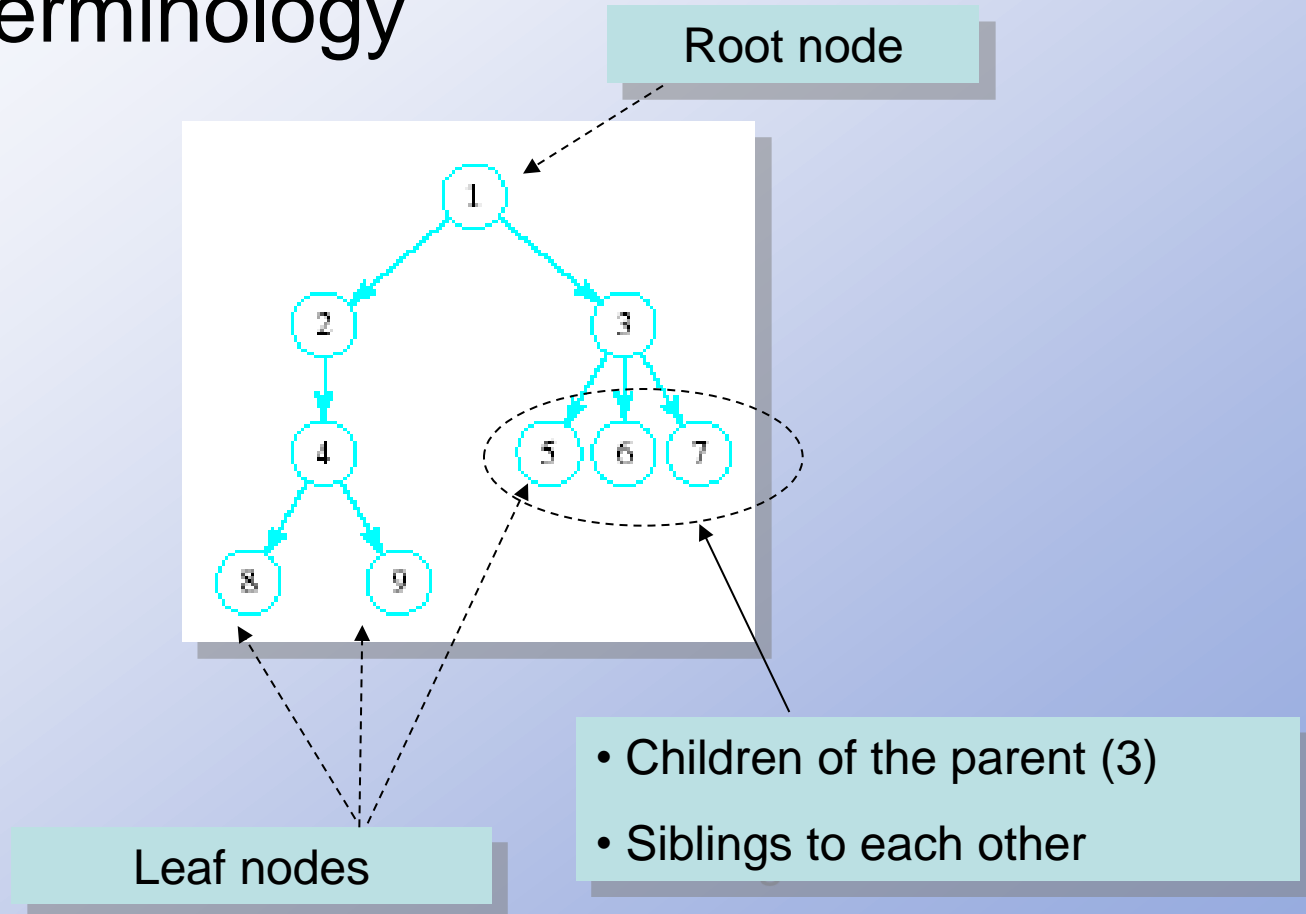


Trees

- A data structure which consists of
 - a finite set of elements called nodes or vertices
 - a finite set of directed arcs which connect the nodes
- If the tree is nonempty
 - one of the nodes (the root) has no incoming arc
 - every other node can be reached by following a unique sequence of consecutive arcs

Trees

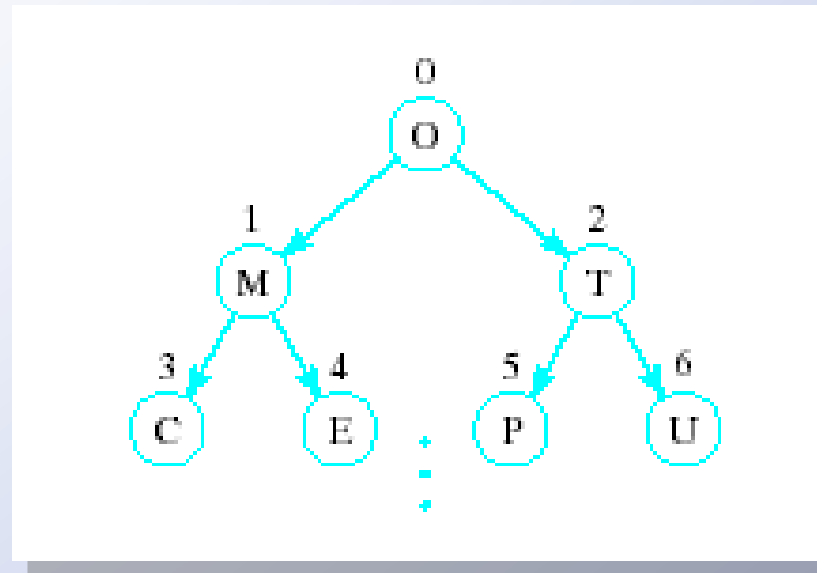
- Tree terminology



Binary Trees

- Each node has at most two children
- Useful in modeling processes where
 - a comparison or experiment has exactly two possible outcomes
 - the test is performed repeatedly
- Example
 - multiple coin tosses
 - encoding/decoding messages in dots and dashes such as Mores code

Array Representation of Binary Trees



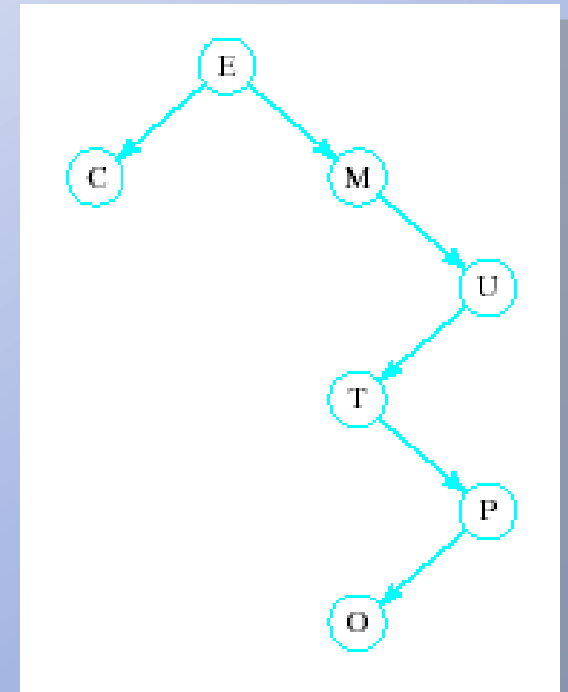
i	0	1	2	3	4	5	6	...
$t[i]$	O	M	T	C	E	P	U	...

- Store the i^{th} node in the i^{th} location of the array

Array Representation of Binary Trees

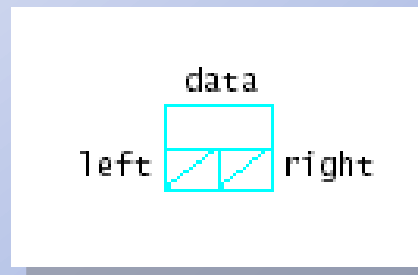
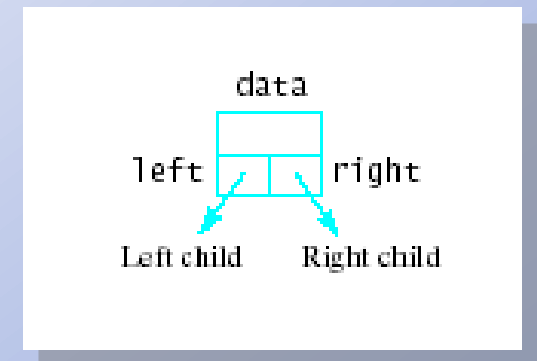
- Works OK for complete trees, not for sparse trees

<i>i</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
<i>t[i]</i>	E	C	M				U							T						
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
									P											
	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
																		O



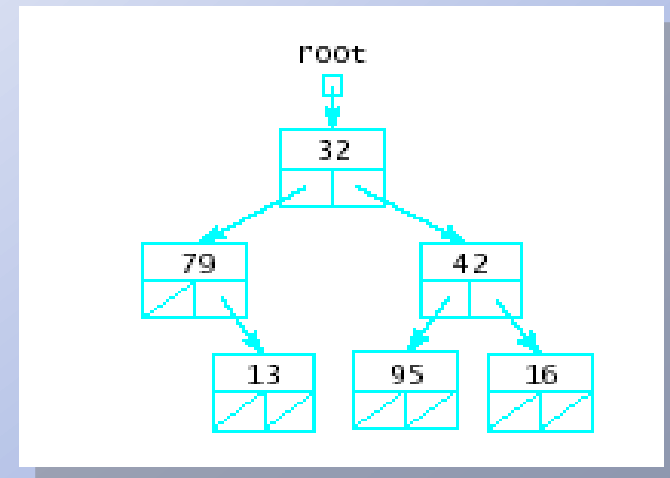
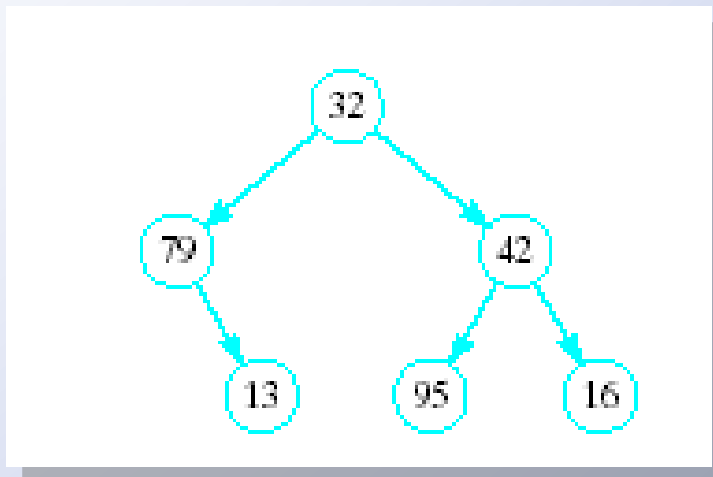
Linked Representation of Binary Trees

- Uses space more efficiently
- Provides additional flexibility
- Each node has two links
 - one to the left child of the node
 - one to the right child of the node
 - if no child node exists for a node, the link is set to NULL



Linked Representation of Binary Trees

- Example



Binary Trees as Recursive Data Structures

- A binary tree is either empty ...
- or

Anchor

- Consists of
 - a node called the root
 - root has pointers to two disjoint binary (sub)trees called ...
- right (sub)tree
 - left (sub)tree

Inductive
step

Which is either empty ...
or ...

Which is either empty ...
or ...

Tree Traversal is Recursive

If the binary tree is empty then
do nothing

Else

N: Visit the root, process data

L: Traverse the left subtree

R: Traverse the right subtree

The "anchor"

The inductive step

Traversal Order

Three possibilities for inductive step ...

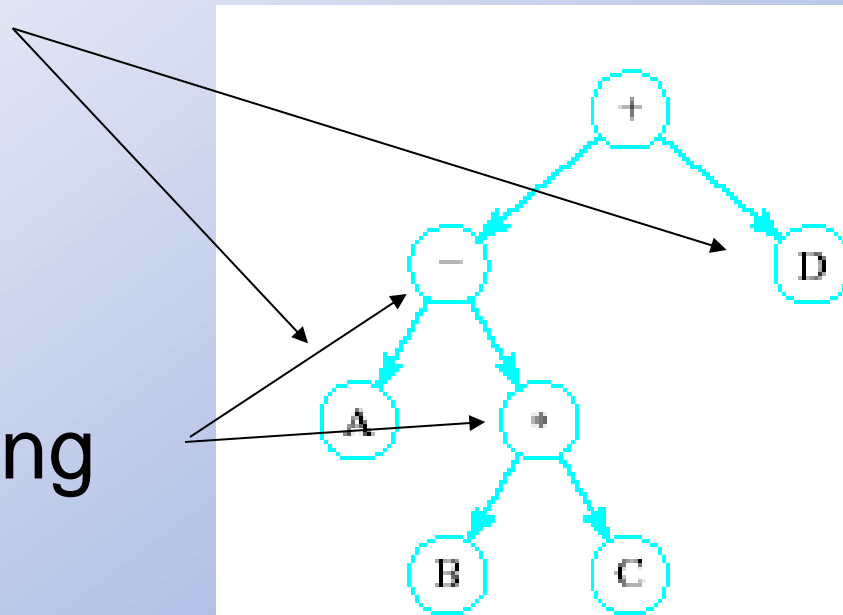
- Left subtree, Node, Right subtree
the inorder traversal
- Node, Left subtree, Right subtree
the preorder traversal
- Left subtree, Right subtree, Node
the postorder traversal

Traversal Order

- Given expression

A - B * C + D

- Represent each operand as
 - The child of a parent node
- Parent node, representing the corresponding operator



Traversal Order

- Inorder traversal produces infix expression

A - B * C + D

- Preorder traversal produces the prefix expression

+ - A * B C D

- Postorder traversal produces the postfix or RPN expression

A B C * - D +

ADT Binary Search Tree (BST)

- Collection of Data Elements
 - binary tree
 - each node x ,
 - value in left child of $x \leq$ value in $x \leq$ in right child of x
- Basic operations
 - Construct an empty BST
 - Determine if BST is empty
 - Search BST for given item

ADT Binary Search Tree (BST)

- Basic operations (ctd)
 - Insert a new item in the BST
 - Maintain the BST property
 - Delete an item from the BST
 - Maintain the BST property
 - Traverse the BST
 - Visit each node exactly once
 - The *inorder traversal* must visit the values in the nodes in ascending order

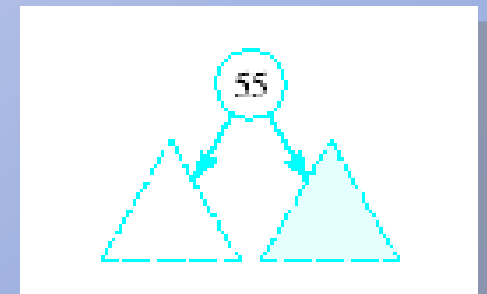
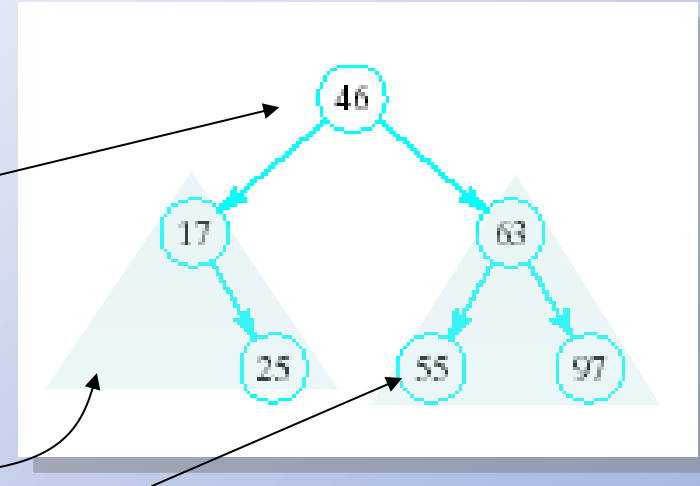
[View BST class template, Fig. 12-1](#)

BST Traversals

- Note that recursive calls must be made
 - To left subtree
 - To right subtree
- Must use two functions
 - Public method to send message to **BST** object
 - Private auxiliary method that can access **BinNodes** and pointers within these nodes
- Similar solution to graphic output
 - Public graphic method
 - Private **graphAux** method

BST Searches

- Search begins at root
 - If that is desired item, done
- If item is less, move down left subtree
- If item searched for is greater, move down right subtree
- If item is not found, we will run into an empty subtree
- View [search\(\)](#)



Inserting into a BST

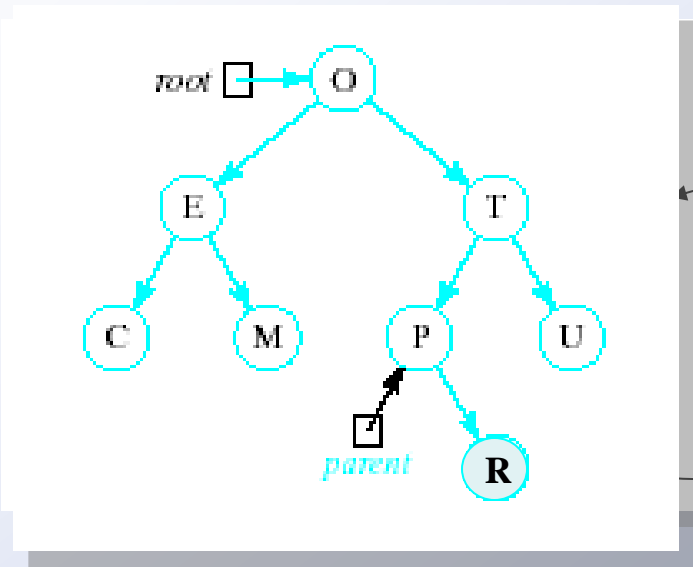
- Insert function

- Uses modified version of search to locate insertion location or already existing item

- Pointer **parent** trails search pointer **locptr**, keeps track of **parent** node

- Thus new node can be attached to BST in proper place

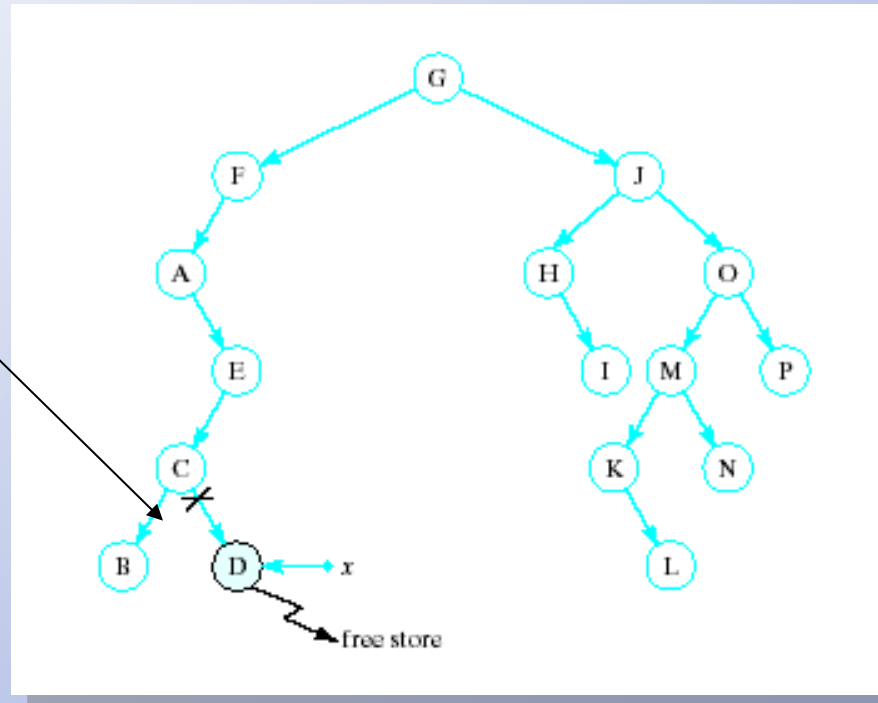
- View **insert()** function



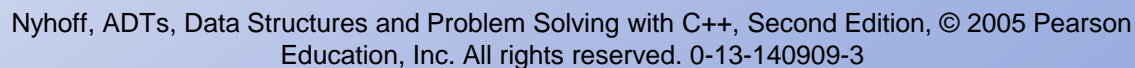
Recursive Deletion

Three possible cases to delete a node, x , from a BST

1. The node, x , is a leaf



2. The node, x has one child

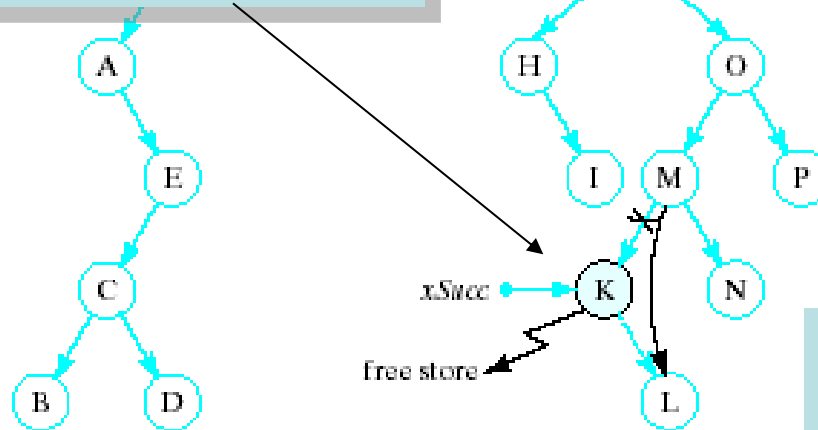


Recursive Deletion

- x has two children

Delete node pointed to by $xSucc$ as described for cases 1 and 2

Replace content
inorder succ



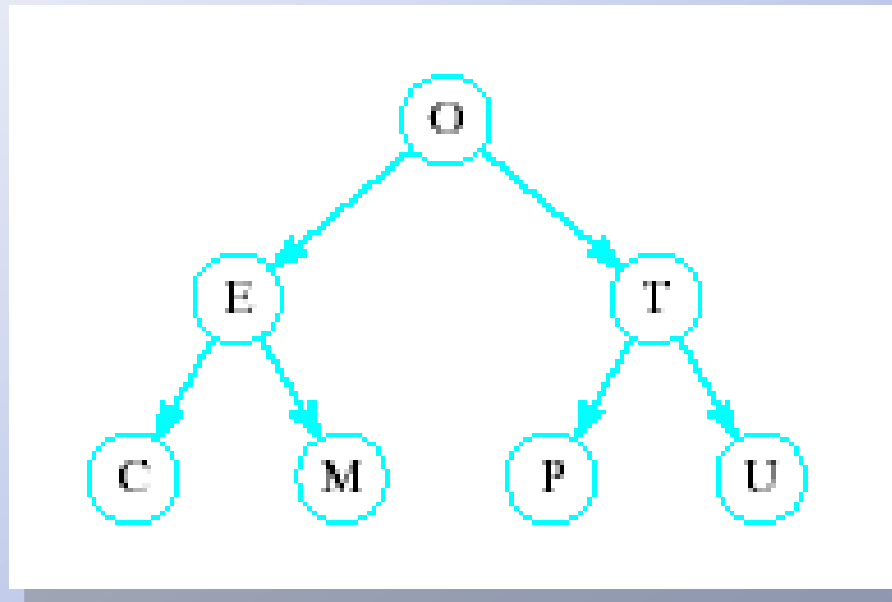
View
`remove()`
function

BST Class Template

- View complete binary search tree template, [Fig. 12.7](#)
- View test program for BST, [Fig. 12.8](#)

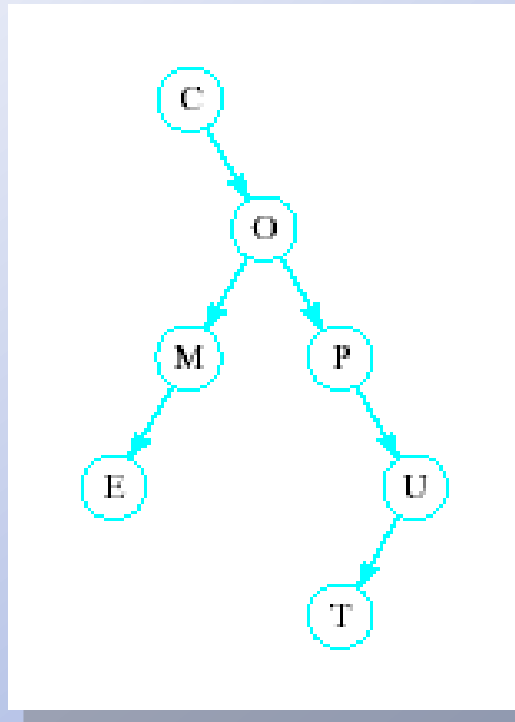
Problem of Lopsidedness

- Tree can be balanced
 - each node except leaves has exactly 2 child nodes



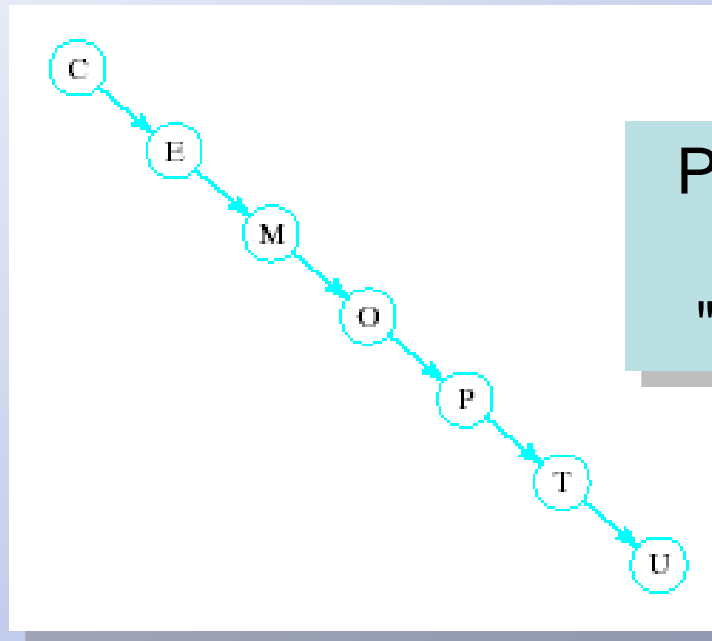
Problem of Lopsidedness

- Trees can be unbalanced
 - not all nodes have exactly 2 child nodes



Problem of Lopsidedness

- Trees can be totally lopsided
 - Suppose each node has a right child only
 - Degenerates into a linked list



Processing time
affected by
"shape" of tree

Case Study: Validating Computer Logins

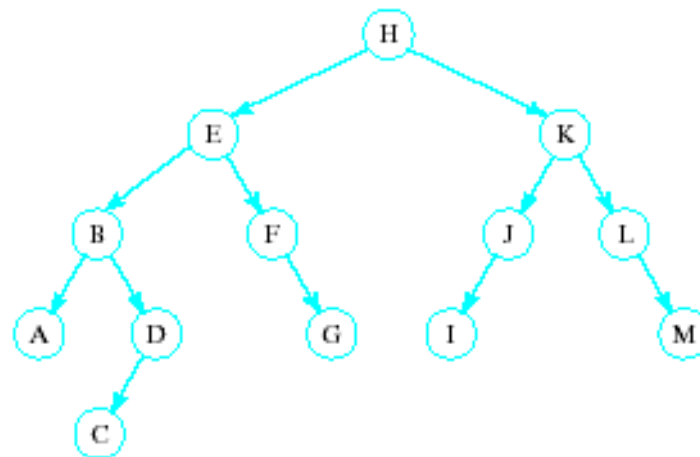
- Consider a computer system which logs in users
 - Enter user name
 - Enter password
 - System does verification
- The structure which contains this information must be able to be searched rapidly
 - Must also be dynamic
 - Users added and removed often

Case Study: Validating Computer Logins

- Design
 - Create class, **UserInfo** which contains ID, password
 - Build BST of **UserInfo** objects
 - Provide search capability for matching ID, password
 - Display message with results of validity check
- View source code, [Fig. 12.9](#)

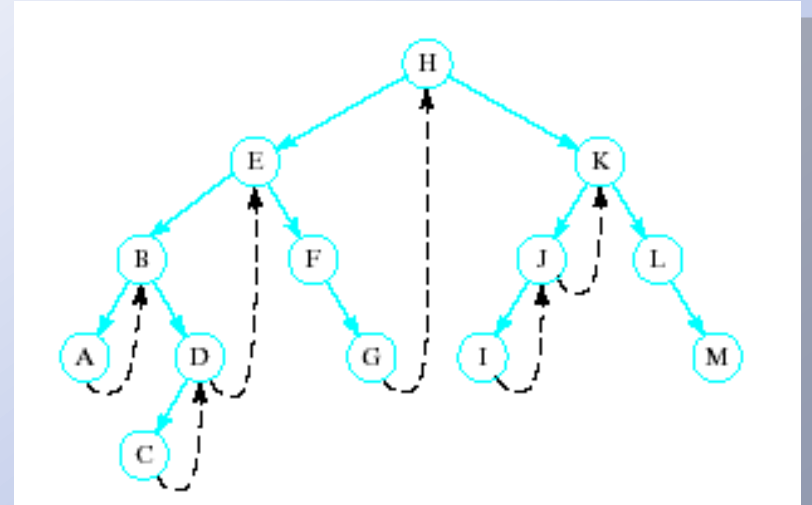
Threaded Binary Search Trees

- Use special links in a BST
 - These *threads* provide efficient nonrecursive traversal algorithms
 - Build a run-time stack into the tree for recursive calls
- Given BST



Threaded Binary Search Trees

- Note the sequence of nodes visited
 - Left to right
- A right threaded BST
 - Whenever a node, x , with null right pointer is encountered, replace right link with thread to inorder successor of x
 - Inorder successor of x is its parent if x is a left child
 - Otherwise it is nearest ancestor of x that contains x in its left subtree



Threaded Binary Search Trees


Traversal algorithm

1. Initialize a pointer, `ptr`, to the root of tree
2. While `ptr != null`
 - a. While `ptr->left` not null
replace `ptr` by `ptr->left`
 - b. Visit node pointed to by `ptr`
 - c. While `ptr->right` is a thread do following
 - i. Replace `ptr` by `ptr->right`
 - ii. Visit node pointed to by `ptr`
 - d. Replace `ptr` by `ptr->right`
- End while

Threaded Binary Search Trees

```
public:
    DataType data;
    bool rightThread;
    BinNode * left;
    BinNode * right;
    // BinNode constructors
    // Default -- data part is default DataType value
    //           -- right thread is false; both links are null
    BinNode()
    : rightThread(false, left(0), right(0) { }

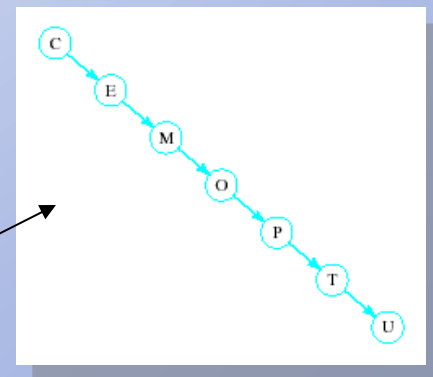
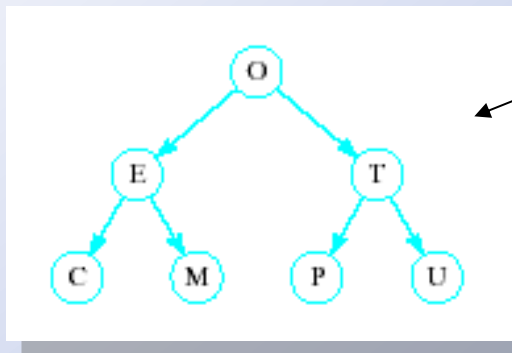
    //Explicit Value -- data part contains item; rightThread
    //                 -- is false; both links are null
    BinNode (DataType item)
    : data(item), rightThread(false), left(0), right(0) { }
}; // end of class BinNode declared
```



Additional field,
rightThread
required

Hash Tables

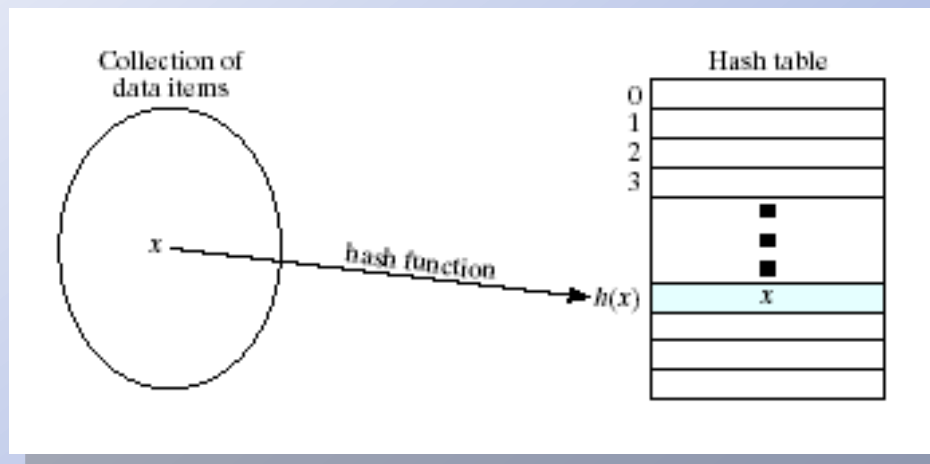
- Recall order of magnitude of searches
 - Linear search $O(n)$
 - Binary search $O(\log_2 n)$
 - Balanced binary tree search $O(\log_2 n)$



- Unbalanced binary tree can degrade to $O(n)$

Hash Tables

- In some situations faster search is needed
 - Solution is to use a hash function
 - Value of key field given to hash function
 - Location in a hash table is calculated



Hash Functions

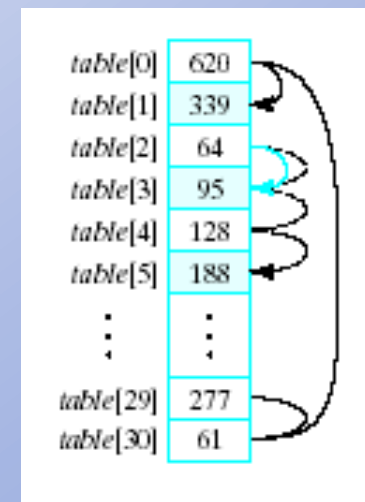
- Simple function could be to mod the value of the key by some arbitrary integer

```
int h(int i)
{ return i % someInt; }
```

- Note the max number of locations in the table will be same as `someInt`
- Note that we have traded speed for wasted space
 - Table must be considerably larger than number of items anticipated

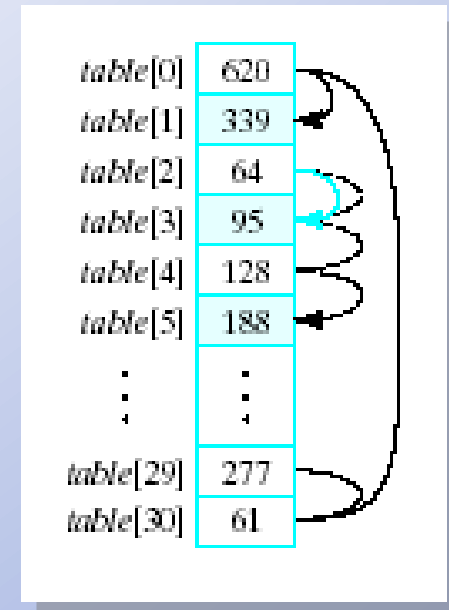
Hash Functions

- Observe the problem with same value returned by $h(i)$ for different values of i
 - Called collisions
- A simple solution is linear probing
 - Linear search begins at collision location
 - Continues until empty slot found for insertion



Hash Functions

- When retrieving a value linear probe until found
 - If empty slot encountered then value is not in table
- If deletions permitted
 - Slot can be marked so it will not be empty and cause an invalid linear probe



Hash Functions

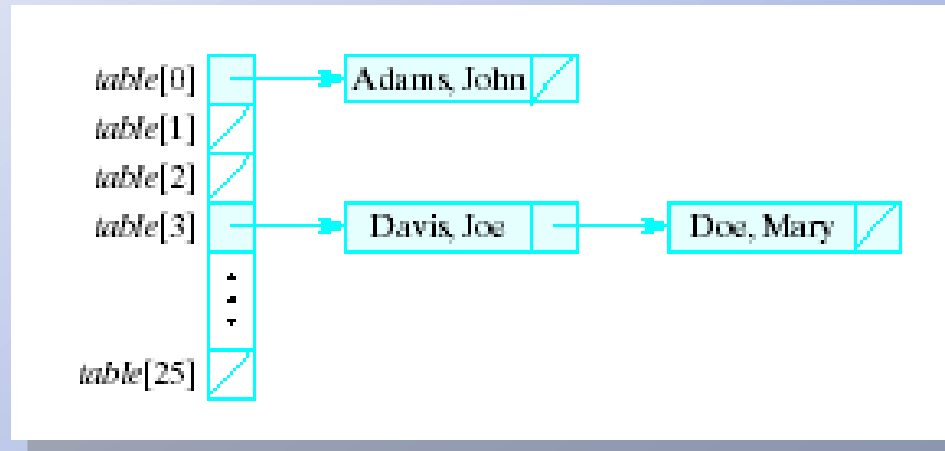
- Strategies for improved performance
 - Increase table capacity (less collisions)
 - Use different collision resolution technique
 - Devise different hash function
- Hash table capacity
 - Size of table must be 1.5 to 2 times the size of the number of items to be stored
 - Otherwise probability of collisions is too high

Collision Strategies

- Linear probing can result in primary clustering
- Consider quadratic probing
 - Probe sequence from location i is
 $i + 1, i - 1, i + 4, i - 4, i + 9, i - 9, \dots$
 - Secondary clusters can still form
- Double hashing
 - Use a second hash function to determine probe sequence

Collision Strategies

- Chaining
 - Table is a list or vector of head nodes to linked lists
 - When item hashes to location, it is added to that linked list



Improving the Hash Function

- Ideal hash function
 - Simple to evaluate
 - Scatters items uniformly throughout table
- Modulo arithmetic not so good for strings
 - Possible to manipulate numeric (ASCII) value of first and last characters of a name