12.1 Linear & Binary Search
assumes data is in a list/array
linear search
start at beginning
check each element until match found or all elements checked
does not need to be sorted
best case - 1st element is match
worst case - no match found, linear
average case - match found midway through

binary search
needs a sorted list
needs random access to elements in list
w/o random access, like STL list, must iterate pointer to search location
cut search space in half each iteration
best case - 1st element is match
worst case - no match found, \( \log_2 n \)
only log because do not search each element
faster as \( n \) increases
Ex \( n = 8,000,000 \) \( \log_2 n = 23 \)

Iterative Pseudocode
takes array called a, search val called item
1. set found to false
2. set first to 0
3. set last to size of a - 1
4. while first \( \leq \) last and not found
   a. calculate \( \text{loc} = (\text{first} + \text{last})/2 \)
   b. if item < a[\text{loc}] then
      set last to \text{loc}-1
   else if item > a[\text{loc}]
      set first to \text{loc} + 1
   else // item == a[\text{loc}]
      set found to true

Recursive Pseudocode
takes array a, search val item, first, last
1. set found to false
2. calculate \( \text{loc} = (\text{first} + \text{last})/2 \)
3. if item < a[\text{loc}] then
   found = bin-search (a, item, first, \text{loc}-1)
else if item > a[\text{loc}]
   found = bin-search (a, item, \text{loc}+1, last)
else
   found = true
4. return found

Hidden time cost-sorted assumption
takes time to sort an unsorted list
would be nice to have a data structure that sorts on
insert/delete
binary search tree is such a data struct.
consider bin-search as following
  right search - location - left search
treat location as root
calculate right & left search into right & left subtrees

12.2 Intro to Binary Trees
Tree Terminology
  nodes/vertices contain the data
  directed arcs/edges connect nodes
  root node has no incoming arcs & can reach all other nodes from
  its outgoing arcs
  path is a sequence of arcs from root to a node (or between two
  nodes)
  leaves are nodes w/ no outgoing arcs
  children are the direct subnodes of a node (1 level down)
  parent is node 1 level up
  siblings are nodes on same level w/ same parent
  descendants are in levels below a node
  ancestors are in levels above a node
  subtree - select one descendant & all of its children &
  descendants
  binary tree has two or less children
Examples of binary trees
  binary search tree
  outcome of a binary trial
    eg flipping a coin
    use a dummy root node
    # levels below root is # trials
    paths show outcome sequences
  decision tree
    each node contains a Y/N question
    follow one child for Y response
    follow other child for N
  construct a code w/ two symbols
    eg Morse code
    arc is labeled w/ symbol
    node contains decoded value for path leading from root to
    that node
    Ex: . E, - T, .. I, .- A, -. N, -- M
Array representation
  slot 0 1 2 3 4 5 6
  node root 0L 0R 1L 1R 2L 2R
  level 0 1 1 2 2 2
  works best for complete trees
  empty slots w/ incomplete trees
  would need a way to indicate empty
  balanced tree
  height of right & left subtree for any node differs by only one
  height is # levels in a tree/subtree
  unbalanced trees not good for array storage
Linked node representation
Linked node representation
node contains storage for data, pointer to left child & pointer to right child
make pointer NULL if no child
very common way to represent trees

12.3 Binary Trees as Recursive Data Struct.
right & left subtrees are also binary trees
recursive definition:
a binary tree is either empty or has a root node, left subtree and right subtree
can use recursive algorithms for tree operations
common operation is traversals
Traversals
visit each node in the tree once
order of visiting nodes is not as vital
simple traversal
1. if tree is empty, do nothing
2. do traversal operation on root (V)
3. traverse left subtree (L)
4. traverse right subtree (R)
changing the order of steps 2-4 is valid
will change order by which nodes are processed
6 ways to order steps 2-4
LVR
VLR
LRV
VRL
RVL
RLV
special terms for certain orders
inorder LVR (infix)
preorder VLR (prefix)
postorder LRV (postfix)
-show math equation example

12.4 Binary Search Trees
is a binary tree w/ bin search tree (BSt) property:
left subtree values are less than root
right subtree values are greater than root
operations
construct empty BST
check empty
search for an item
insert a new item
delete an item
inorder, preorder & postorder traversals
(book only has inorder traversal)
Operation Pseudocode
construct empty
set root to NULL
check empty
if root is NULL
if root is NULL
    return true
else
    return false

search for an item
if tree is empty
    return false
else if item < root's data
    return search left subtree
else if item > root's data
    return search right subtree
else
    return true

insert item into tree
if tree is empty
    allocate node for item
    set root to node
else if item < root's data
    insert item in left subtree
else if item > root's data
    insert item in right subtree
else
    output (either cout or cerr) that item is already in the tree

delete an item from a tree
Issue: filling the deleted node while maintaining BST property
Three cases for deleted node:
    it is a leaf - delete it
    it has one child - move child up into its place
    it has two children - replace w/ either inorder successor or predecessor
      (largest value in left subtree or smallest value in right subtree)
then delete the replacement node
    replacement node should be leaf or have just one child
    since we only allow unique valves in the tree

Pseudocode
    // Find item's node & parent node
    set found to false
    set node to root
    set parent to NULL
    while not found and node is not NULL
        if item < node's data
            set parent to node
            set node to node's left child
        else if item > node's data
            set parent to node
            set node to node's right child
        else
            set found to true
    if not found
        issue "item not in tree" error
        return from function
    if node has two children
        set replacement to node's right child
set parent to node
while replacement has a left child
    set parent to replacement
    set replacement to its left child
    set node's data to replacement's data
    set node to replacement
    set subtree to node's left child
    if subtree is NULL
        set subtree to node's right child
    if parent is NULL
        set root to subtree
    else if parent's left child is node
        set parent's left child to subtree
    else
        set parent's right child to subtree
    delete node
traverse tree in order, prints ascending values
    if tree is empty
        do nothing
    traverse left subtree
    print root's data
    traverse right subtree

Problem of lopsidedness
BST property does not ensure that the tree is complete or balanced
insertion order can greatly affect balance
worst case - insert in sorted order, either ascending or descending
    results in a linked list
balanced trees take log2n for insert, delete, & search
unbalanced trees can be as bad as linked lists, so can be linear
rebalancing trees can solve this
    will discuss at end of quarter

12.7 Hash Tables
very fast searching, but sacrifices storage space
average time for insertions, deletions & searches is constant
hashing eliminates trial and error searching like w/ trees
has a table to store data (hash table)
hash function ideally stores each item in a unique slot
    not always possible in practice since hash table is finite &
data to store can be infinite
uniqueness of slot also affected by nature of hash function
Hash Functions
purpose is to take an element & generate a key
key is a slot in the hash table
Modulo function
    take the element and modulo it by the hash table size
issue is that elements will overlap
    Example: hash table size is 100
        then 0, 100, 200, etc will all map to key 0
        this is called a collision
    if element is not an int, have to compute an int off its value
Example: add up int value of chars in a string
no one perfect hash function for all datatypes
goal is to evenly distribute the elements across the whole hash table
Random hashing
\[
\text{randInt} = (\ (\text{MULT} \times \text{item}) + \text{ADD}) \mod \text{MOD};
\]
\[
\text{key} = \text{randInt} \mod \text{tableSize};
\]
Collision Strategies
how to handle when function does not generate unique keys
Increased Hash Table size
if capacity is 1.5 to 2 times greater than expected number of items, fewer collisions occur
prime number sizes best for modulo hash functions
can't arbitrarily increase size & expect better performance
if storing 0-500, then for table sizes > 500, the upper slots will never be result of hash function
Linear Probing
search linearly through table for an empty slot on insert
requires an "empty slot" value to tell used & unused slots apart
on search, if key shot does not match, probe ahead until a match or empty slot is found
on delete, use a "deleted" value so search knows to keep probing
issue: primary clustering
   elements that map to same/close key start forming clusters
   causes increased time for insert, delete & search
   linear in worst case if whole table is probed
Quadratic Probing
try to avoid primary clustering
search slots in following order:
\[
\text{key} + 1, \text{key} - 1, \text{key} + 2^2, \text{key} - 2^2, \text{key} + 3^2, \text{key} - 3^2, ...
\]
issue: secondary clustering
   same key probes same sequence
Double Hashing
use a second, different hash function for probe sequence
probe sequence is:
\[
\text{key}, \text{key} + 2\text{nd key}, \text{key} + (2\text{nd key})^2, \text{key} + (2\text{nd key})^2, ...
\]
second key should never be zero since 0*2 is still 0
good choice for second function is:
\[
\text{R} - (\text{item} \mod \text{R})
\]
where R is a prime number smaller then the hash table size
table size should also be prime for double hashing
if not prime, sequence could wrap around & probe the same slot(s)
Example: table size = 10, key = 0, 2nd key = 5
probes sequence: 0, 5, 0, 5, 0, 5, ...
Separate Chaining
don't probe ahead for a free slot
instead, store linked list of collisions for each slot
have to traverse list on delete & search
  (head insert removes need to traverse on insert)
increases time for those operations from constant to the
chain length
Rehashing
  hash tables are less efficient as they fill up
  rehashing increases the hash table size
    usually to a prime approximately twice the size of she
current table
  all elements are removed from original table & have their
keys recomputed