目录

Trees and Binary Search Trees

(Chapter 5.4 - 5.7 Sedgewick)

Tree are data structures, like arrays and linked lists.

Trees are like doubly-linked lists: nodes contain data and multiple links to other nodes.

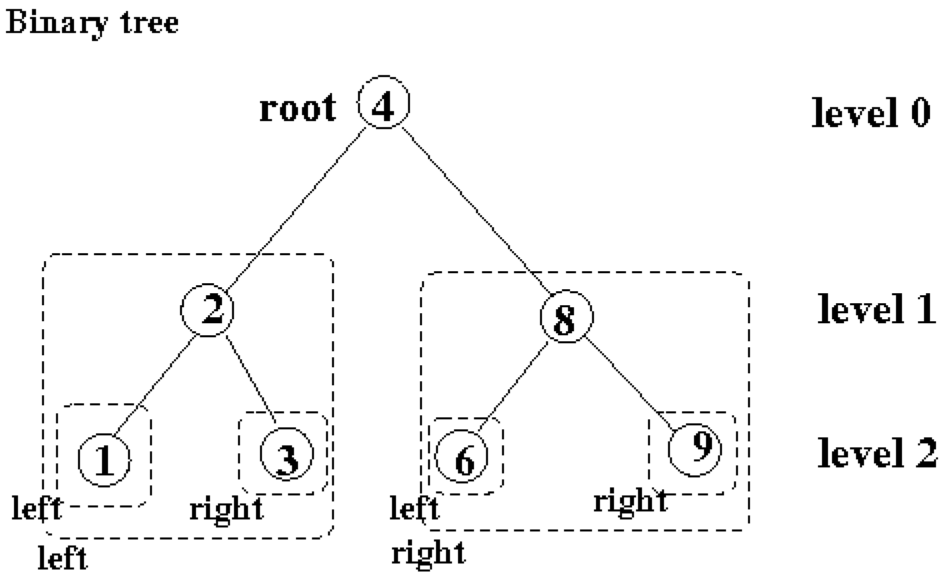
- Nodes are:
    - internal, and have links to other nodes, called their children
    - external, called *leaves* or *terminals*, and have no links to other nodes
- Every node has a parent node, except one, which is called the *root* node
- The *descendants* of a node consist of all the nodes reachable on a path from that node.
- Children with the same parents are called *siblings*

Formally, a tree is an acyclic graph in which each child has at most one parent.

Connections between nodes are called edges or links

- A **path** is a set of connected edges
- We usually consider paths that go only one way i.e. only up or down
- **Height** is the length of the longest path from the root
    - so the root node is at height 0
    - children of the root node are at height 1
- **Node level** or **depth** is the path length from the root to the node

        - Depth of the root is 0
- Hierarchy of trees and subtrees
    - assuming 2 children for each internal node:
        - tree at left child is called a *left subtree*
        - tree at right child is called a *right subtree*



Types of trees

Assume we have a tree with internal nodes and leaves, and where each node has a data value

- *ordered* tree
    - children are in order (left and right children have order)
- *binary* tree
    - each internal node has at most 2 children
- *full binary* tree
    - each internal node has exactly 2 children
- *perfect binary* tree
    - binary tree in which all leaves are at the same depth
- ***ordered binary tree***
    - left subtree values <= parent value
    - right subtree values >= parent value
        - great for searching: e.g.
            - search for smallest: keep going left down the tree
            - search for largest: keep going right down the tree
            - search for a specific element: use 'classic' binary search
    - also called a ***binary search tree***
- *full m-ary* tree
    - each internal node has exactly *m* children

Binary Trees

Binary trees can be

- **balanced**
    - tree has minimal height for the given number of nodes
    - number of nodes in the left and right subtrees differ by at most 1

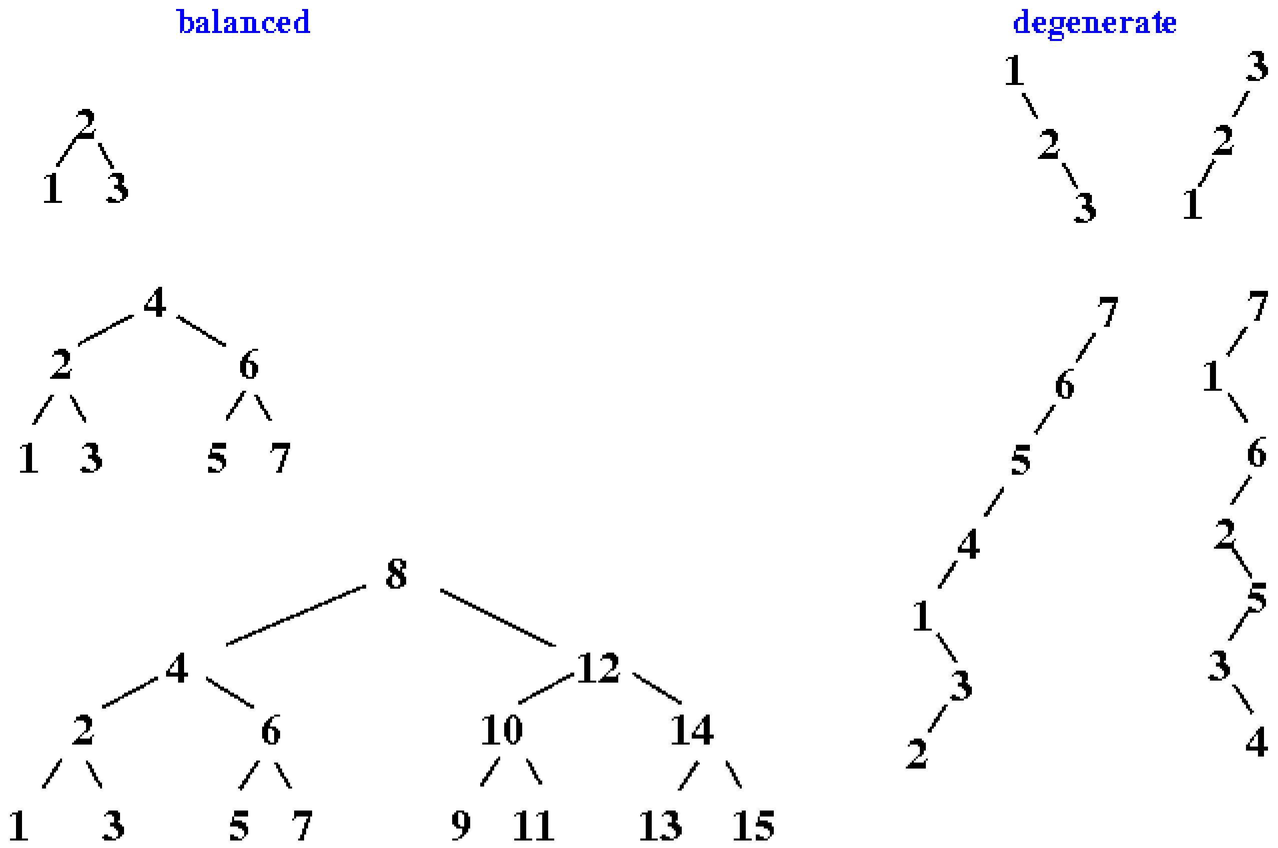
- **degenerate**
    - tree with maximal height (i.e. every parent has 1 child)

Wikipedia: binary trees

Height of a binary tree

What is the height of a binary tree consisting of *n* nodes:

- what was the definition of height?
    - the length of the longest path
- what is the maximum height?
    - the tree is degenerate
        - height is *n-1*
- what is the minimum height
    - the tree is balanced
        - height is *ln(n)*



Number	Balanced Height	Degenerate Height
3	1	2
7	2	6
15	3	14
n	lg n	n−1

Depth of a binary tree

The depth of a node *x* is the length (in edges) of the path from *x* to the root. Computationally,

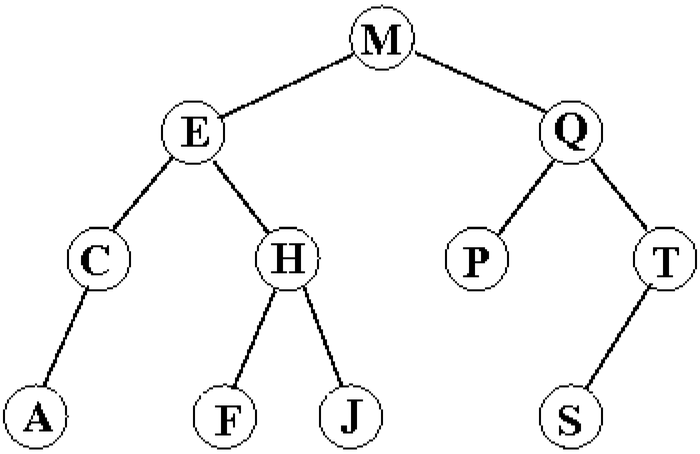
- if *n* is a root node then *depth(n) = 0*
- else *depth(n) = 1 + depth(parent_of n)*

The maximum depth of any node in a tree is the height of the tree

Binary Search Tree

```
  A BST is a tree where for every ('parent') node:
   * if the node has a left child, its key is smaller than the key of the
node
   * if it has a right child, its key is larger than or equal to the key
of the node
```

Example of a BST:



- notice they are ordered from left to right if you 'abstract away' the height (i.e. flatten the tree)

Data structure for a binary tree

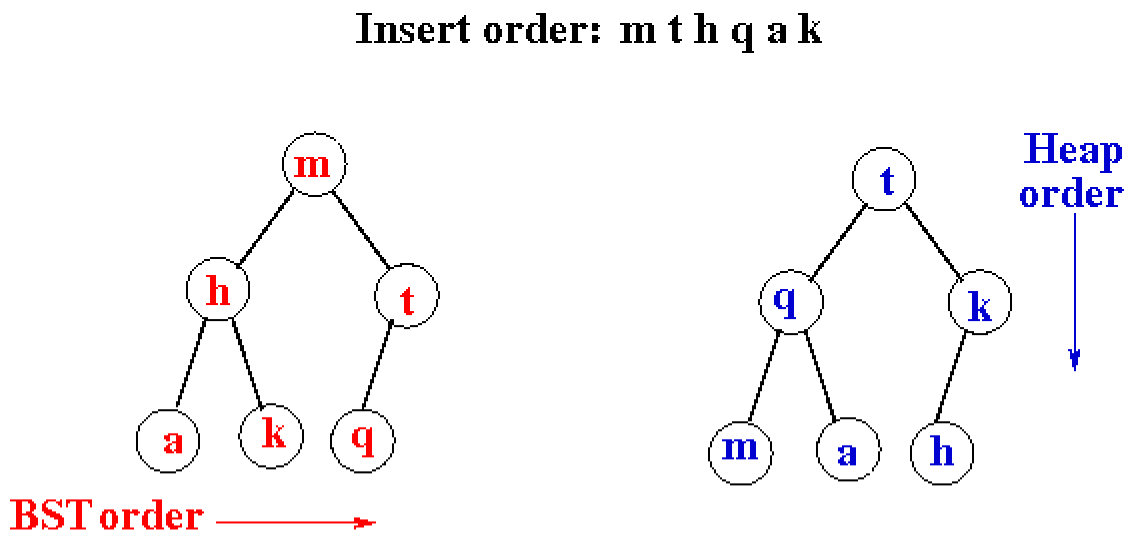
切换行号显示

```
typedef struct node *Tree;
struct node {
   int data;
   Tree left;
   Tree right;
};
```

Comparing Binary Search Trees and Heaps

- **Heaps** are trees with ***top-to-bottom ordering***
  - satisfy the *Complete Tree Property*
- **Binary Search Trees** are trees with ***left-to-right ordering***
  - there is NO *Complete Tree Property* for BSTs
    - ... they can be degenerate!
      - ... hence cannot be implemented as arrays. We must use linked lists.

Here is an example of a BST and a heap for the same input:



A BST satisfies the property:

- for node with key *k*
  - all node keys in left subtree <*k*
  - all node keys in right subtree >= *k*
  - property applies to all nodes in the BST

BSTs can be great for searching

- if *n* is the size of the input, and the height of the BST is *ln(n)* then
  - binary search performs as *O(ln(n))*

... but the bad news is that BSTs can be degenerate

- the BST then has height *n*
- this is the worst-case behaviour
- binary search then performs as *O(n)*
  - this is just linear search, which is much slower (*remember the Sydney phone book analogy!*)

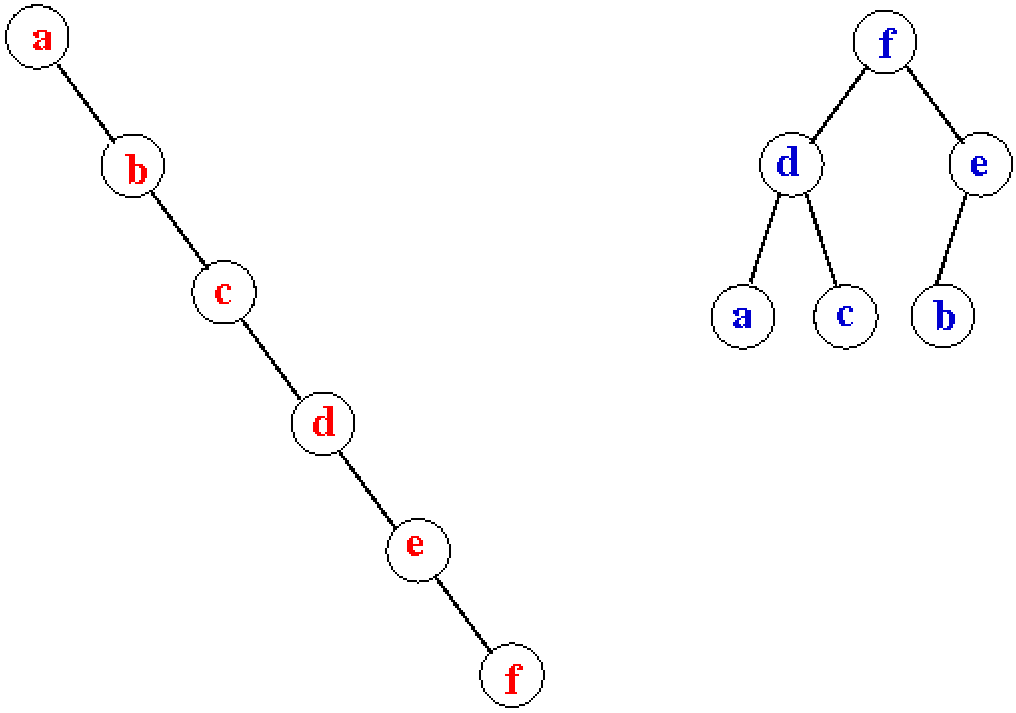
We construct a BST as we read the elements

- we cannot control the order of the input ...
- in fact, we will build a degenerate tree if the input is ordered

On the left is the result if we simply insert the nodes as we read

- on the right is what we would like the result to be

Insert order: a b c d e f



Searching in BSTs

A BST is a perfect data structure to do binary search

Reminder: what is a *binary search*?

- Prerequisite:
    - the items in a sequence must be sorted
- It is a *divide and conquer* technique
    - split the data into 2 parts
        - determine to which part the item belongs
        - recurse down until arrive at the base case
            - which is either NULL (element not found)
            - or the element itself

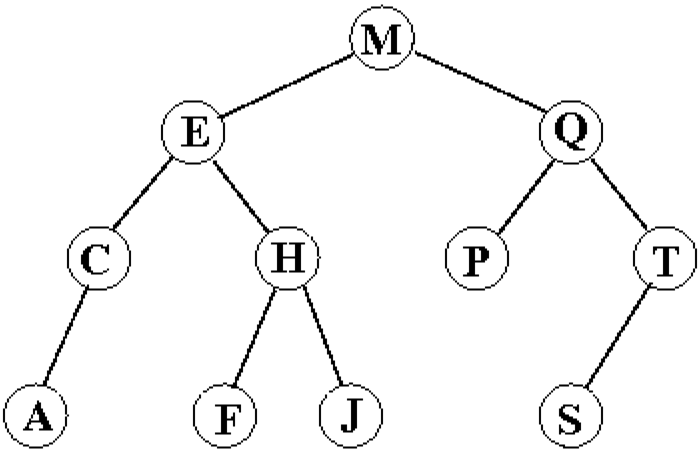
Assume a node contains a single key (and maybe some more data).

- a key is just the data that we want to order by, and search for

Basic idea:

```
If the value of the item is less than the item in the current node, then
go left, otherwise go right
```

For example, given the following BST:



To search for item *F*

- *searchTree(M,F)*
  - go left and *searchTree(E,F)*
    - go right and *searchTree(H,F)*
      - go left and *searchTree(F,F)*
        - *success*

If the search was for item *G*, then we would have had the sequence

- *searchTree(M,G)*
  - go left and *searchTree(E,G)*
    - go right and *searchTree(H,G)*
      - go left and *searchTree(F,G)*
        - *F* is a leaf and *F != G* so *failure*

切换行号显示

```
int searchTree(Tree t, int v){ // Search for the node with value v
   int ret = 0;
   if (t != NULL) {
      if (v < t->data) {
         ret = searchTree(t->left, v);
      }
      else if (v > t->data) {
         ret = searchTree(t->right, v);
      }
      else { // v == t->data
         ret = 1;
      }
   }
   return ret;
} // returns non-zero if found, zero otherwise

```

Creating a node in a BST

Just like a linked list, we must:

- call *malloc()* to create the tree node
- initialise the data
- initialise the pointers

切换行号显示

```
typedef struct node *Tree;
struct node {
   int data;
   Tree left;
   Tree right;
};

Tree createTree(int v) {
   Tree t;
   t = malloc(sizeof(struct node));
   if (t == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
   }
   t->data = v;
   t->left = NULL;
   t->right = NULL;
   return t;
}
```

Freeing a node in a BST

The pointers in a BST node point to other nodes

- we need to follow the pointers to the last node, and work backwards freeing nodes
- otherwise we will have (severe) memory leaks

In the following code, we recurse down the tree, and call *free(t)* for each node from the bottom up

切换行号显示

```
   void freeTree(Tree t) { // free in postfix fashion
      if (t != NULL) {
         freeTree(t->left);
         freeTree(t->right);
         free(t);
      }
      return;
   }
```

Inserting a node in a BST

Trees seem to be linked lists with 2 links instead of 1(?)

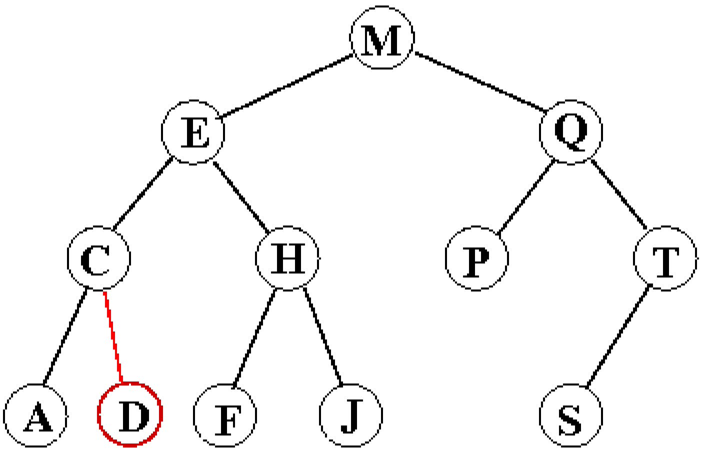
- **traversing** is similar (just follow the left or right link)
- **insertion**?
  - obvious in a linked list, but what strategy is used in a BST?
- **deletion**?
  - obvious in a linked list, but what happens to the 'children' in a BST?

In a BST, when we insert a new node:

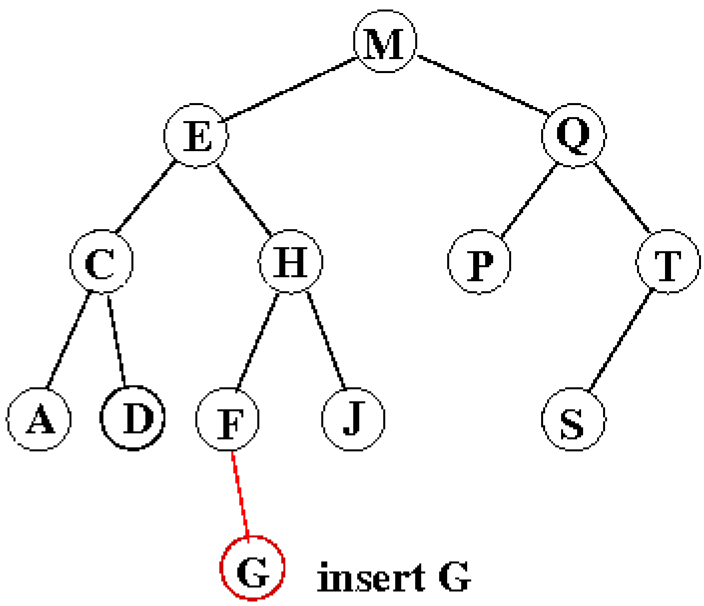
- it **always becomes a leaf** (impossible to insert a non-leaf node)
- it **always maintains the ordering** of the tree
  - it must be on the left of all nodes larger than it
  - it must be on the right of all nodes smaller than or equal to it

Algorithm:

i. follow the path from the root towards the leaves as though we were searching for the item
ii. when we get to a NULL, we have found our insertion point
    - the NULL must be either the left or right child of a node
        - if the value is smaller than the current node, then the NULL is the left node
        - otherwise it becomes the right child (in our implementation, duplicates go on the right)
iii. create a node for the item, and link it in by replacing the NULL with it



insert D



Finding the insertion point can be done recursively:

切换行号显示

```
Tree insertTree(Tree t, int v) {
   if (t == NULL) {
      t = createTree(v);
   }
   else {
      if (v < t->data) {
        t->left = insertTree (t->left, v);
      }
      else {
        t->right = insertTree (t->right, v);
      }
```

```
    }
    return t;
}
```

or we can do it iteratively:

切换行号显示

```
Tree insertTreeI(Tree t, int v) { // An iterative version of the above
   if (t == NULL) {
      t = createTree(v);
   }
   else { // t != NULL
      Tree parent = NULL;  // remember the parent to link in new child
      Tree step = t;
      while (step != NULL) { // this is the iteration
         parent = step;
         if (v < step->data) {
           step = step->left;
         }
         else {
           step = step->right;
         }
      } // step == NULL
      if (v < parent->data) {
         parent->left = createTree(v);
      }
      else {
         parent->right = createTree(v);
      }
   }
   return t;
}
```

The order of the input can make a huge difference in the structure of the BST

For example consider the input values 1, 2, 3 and 4

- There are 4*3*2 possible orders of these 4 numbers:

```
1234 1243 1324 1342 1423 1432
2134 2143 2314 2341 2413 2431
3124 3142 3214 3241 3412 3421
4123 4132 4213 4231 4312 4321
```

What is the BST that results from each of these 24 'orders':

```
"1234"    "1243"    "1432"    "1423"    "1324"
                                        "1342"

1         1           1         1        1
 \         \           \         \        \
  2         2           4         4        3
   \         \         /         /        / \
    3         4       3         2        2   4
     \       /       /           \
      4     3       2             3
```

```
"4321"    "4312"    "4123"    "4132"    "4213"
                                        "4231"
      4       4       4          4         4
     /       /       /          /         /
    3       3       1          1         2
```

```
   /       /         \          \       / \
  2       1           2          3     1   3
 /         \           \        /
1           2           3      2
```

```
"2134"     "2143"
"2341"     "2431"
"2314"     "2413"
  2             2
 / \           / \
1   3         1   4
     \             \
      4             3
```

```
"3214"     "3124"
"3421"     "3412"
"3241"     "3142"
    3           3
   / \         / \
  2   4       1   4
 /             \
1               2
```

Notice, different input can result in the same BST.

Example: putting the basic tree operations together

切换行号显示

```
// basic.c: insert nodes into a BST, print the tree and free all nodes
#include <stdio.h>
#include <stdlib.h>

typedef struct node *Tree;
struct node {
   int data;
   Tree left;
   Tree right;
};

Tree insertTree (Tree, int);
Tree createTree (int);
void printTree  (Tree);
void freeTree   (Tree);

int main(void) {
   Tree t;

   t = createTree (7);
   t = insertTree(t, 8);
   t = insertTree(t, 6);
   t = insertTree(t, 5);
   t = insertTree(t, 4);
   t = insertTree(t, 3);
   t = insertTree(t, 2);
   t = insertTree(t, 1);
   printTree(t);
   putchar('\n');
   freeTree(t);
   return EXIT_SUCCESS;
}

Tree insertTree(Tree t, int v) {
   if (t == NULL) {
      t = createTree(v);
```

```
   }
   else {
      if (v < t->data) {
        t->left = insertTree (t->left, v);
      }
      else {
        t->right = insertTree (t->right, v);
      }
   }
   return t;
}

Tree createTree (int v) {
   Tree t = NULL;

   t = malloc (sizeof(struct node));
   if (t == NULL) {
      fprintf(stderr, "Memory is exhausted: exiting\n");
      exit(1);
   }
   t->data = v;
   t->left = NULL;
   t->right = NULL;
   return t;
}

void printTree(Tree t) { // not the final version
    if (t != NULL) {
        printTree (t->left);
        printf ("%d  ", t->data);
        printTree (t->right);
    }
    return;
}

void freeTree(Tree t) { // free in postfix fashion
   if (t != NULL) {
      freeTree(t->left);
      freeTree(t->right);
      free(t);
   }
   return;
}
```

There is no input to the program: the values are hard-coded into the program

- insert 7, 8, 6, 5, 4, 3, 2, 1

The output is:

```
1  2  3  4  5  6  7  8
```

Notice that 1 was the last input value, but 1 is the first output

- ... because it is the most left-most descendent of the root

Let's consider how we can draw a tree as a 2D structure (and not a sequence of numbers)

Printing a BST

The *printTree()* function actually converts the 2-D tree into 1-D **infix notation**:

- left child -- parent -- right child

- the order is correct, reading left to right, but there is no structure

What does the 'real' BST above look like?

```
             7
            / \
           6   8
          /
         5
        /
       4
      /
     3
    /
   2
  /
 1
```

Our *infix printTree()* above recursively descends the tree as follows:

```
left child of 7 is 6
  left child of 6 is 5
    left child of 5 is 4
      left child of 4 is 3
        left child of 3 is 2
          left child of 2 is 1
            left child of 1 is NULL
            parent; print '1'        <== 1
            right child of 1 is NULL
          parent; print '2'          <== 2
          right child is NULL
        parent; print '3'            <== 3
        right child is NULL
      parent; print '4'              <== 4
      right child is NULL
    parent; print '5'                <== 5
    right child is NULL
  parent; print '6'                  <== 6
  right child is NULL
parent; print '7'                    <== 7
right child of 7 is 8
  left child of 8 is NULL
  parent; print '8'                  <== 8
  right child of 8 is NULL
```

Can we draw the BST properly (as a 2-D structure) based on the above?

What about printing the tree 'on its side'

```
                                                 1
                                         2
                                 3
                         4
                 5
         6
7
         8
```

- notice the order is 1...8
- each new 'generation' goes on a newline
- the deeper we go, the bigger the indent
    - pass a 'depth' parameter down the tree to print leading spaces or tabs

Here is the result:

切换行号显示

```
void printTree(Tree t, int depth) { // extra depth parameter
    if (t != NULL) {
        depth++;
        printTree (t->left, depth);
           for (int i=1; i<depth; i++){ // 'depth'*whitespace
              putchar('\t');
           }
           printf ("%d\n", t->data); // node to print
        printTree (t->right, depth);
    }
    return;
}
```

The result is that we indent every node by *n* tabs, where *n* is the depth of the node

- '1' is at level 6
- '2' is at level 5
- ...
- '6' and '8' are at level 1
- '7' is at level 0

This generates the BST lying on its side shown above.

More functions on a BST

Count the number of nodes in a BST

切换行号显示

```
int count(Tree t){
   int countree = 0;
   if (t != NULL) {
      countree = 1 + count(t->left) + count(t->right);
   }
   return countree;
}
```

Find the height of a BST

First define a helper function to find the maximum of two numbers:

切换行号显示

```
int max(int a, int b){
   if (a >= b) {
      return a;
   }
   return b;
}
```

Now a function that returns the height of a BST:

切换行号显示

```
int height(Tree t){
   int heightree = -1;
   if (t != NULL){
      heightree = 1 + max(height(t->left), height(t->right));
   }
   return heightree;
}
```

How balanced is a BST?

How do the number of nodes in the left sub-tree and the right sub-tree compare?

切换行号显示

```
int balance (Tree t){ // calculates the difference between left and right
   int diff = 0;

   if (t != NULL) {
      diff = count(t->left) - count(t->right); // count declared elsewhere
      if (diff < 0) {
         diff = -diff;
      }
   }
   return diff;
}
```

Example: Putting it all together

- create a tree and output its balance, height and count, and free the data structure

切换行号显示

```
// unbalanced.c: create and check the balance of a tree
#include <stdio.h>
#include <stdlib.h>

typedef struct node *Tree;
struct node {
   int data;
   Tree left;
   Tree right;
};

void printTree(Tree, int); // print a BST with indentation
Tree createTree(int);         // create a BST with root 'v'
Tree insertTree(Tree, int);// insert a node 'v' into a BST
void freeTree(Tree);        // give the memory back to the heap

int count(Tree);
int balance(Tree);
int height(Tree);

int main(void) {
   Tree t;

   t = createTree(7);
   t = insertTree(t, 8);
   t = insertTree(t, 6);
   t = insertTree(t, 5);
   t = insertTree(t, 4);
   t = insertTree(t, 3);
```

```
   t = insertTree(t, 2);
   t = insertTree(t, 1);
   printTree (t, 0);
   printf("Balance = %d\n", balance(t));
   printf("Height = %d\n", height(t));
   printf("Count = %d\n", count(t));

   freeTree(t);
   return EXIT_SUCCESS;
}
void printTree(Tree t, int depth) {
    if (t != NULL) {
        depth++;
        printTree (t->left, depth);
        int i;
        for (i=1; i<depth; i++){
            putchar('\t');
        }
        printf ("%d\n", t->data);
        printTree (t->right, depth);
    }
    return;
}

Tree createTree (int v) {
   Tree t;
   t = malloc(sizeof(struct node));
   if (t == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
   }
   t->data = v;
   t->left = NULL;
   t->right = NULL;
   return t;
}

Tree insertTree(Tree t, int v) {
   if (t == NULL) {
      t = createTree(v);
   }
   else {
      if (v < t->data) {
        t->left = insertTree (t->left, v);
      }
      else {
        t->right = insertTree (t->right, v);
      }
   }
   return t;
}

int count(Tree t){
   int countree = 0;
   if (t != NULL) {
      countree = 1 + count(t->left) + count(t->right);
   }
   return countree;
}

   int max(int a, int b){
      if (a >= b){
         return a;
      }
      return b;
   }

int height(Tree t){
   int heightree = -1;
```

```
   if (t != NULL){
      heightree = 1 + max(height(t->left), height(t->right));
   }
   return heightree;
}

int balance (Tree t){ // calculates the difference between left and right
   int diff = 0;

   if (t != NULL) {
      diff = count(t->left) - count(t->right);
      if (diff < 0) {
         diff = -diff;
      }
   }
   return diff;
}

void freeTree(Tree t) { // free in postfix fashion
   if (t != NULL) {
      freeTree(t->left);
      freeTree(t->right);
      free(t);
   }
   return;
}
```

The output is:

```
                                                        1
                                                2
                                        3
                                4
                        5
                6
        7
                8
Balance = 5
Height = 6
Count = 8
```

Deleting a node from a BST

Deletion is harder than insertions. We could:

- find the node to be deleted
- unlink the node from its parent

But what do we do with the deleted node's children?

Easy option, don't delete, just mark the node as deleted

- Future searches ignore this item
- Problem? Tree can become full of 'deleted nodes'!

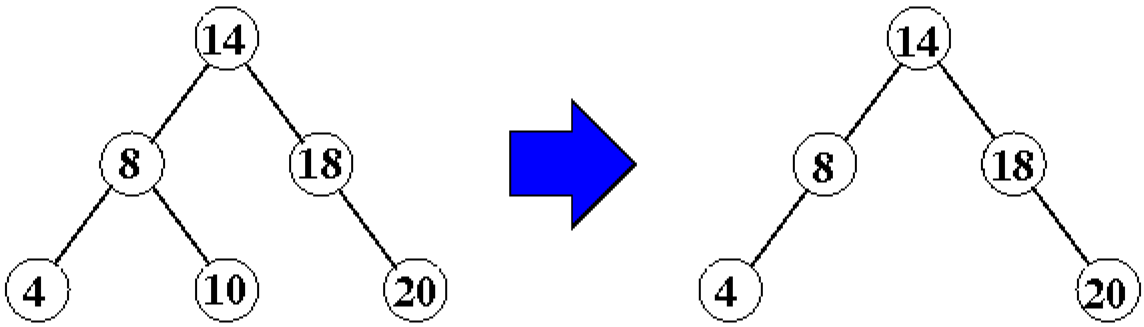
Hard option has 3 cases:

1. node is a **leaf**
    - there are no children, so unlink node from parent
2. node has **1 child**

   - simply replace the node by its child
3. node has **2 children**
   - *we need to rearrange the tree in some way*

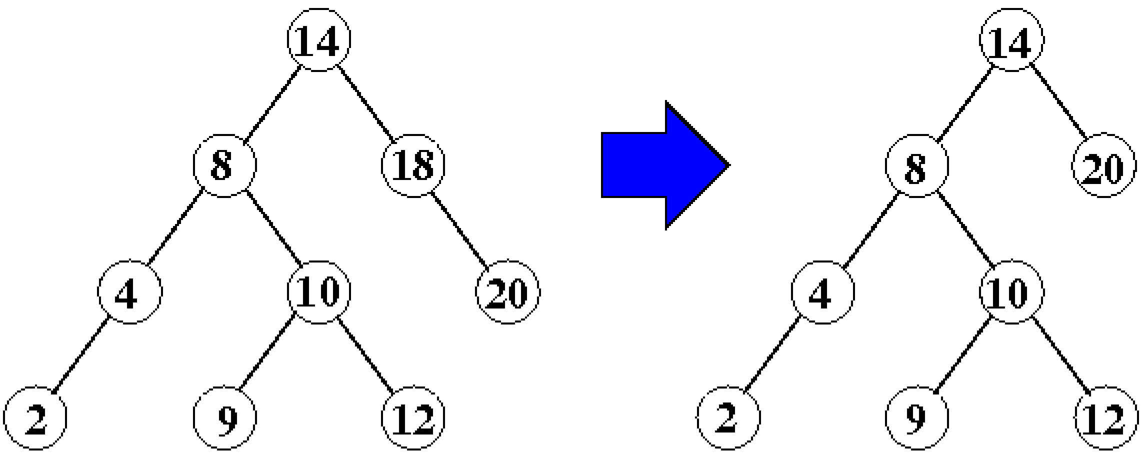
node is a leaf

- *Delete 10 from this tree*



node has 1 child

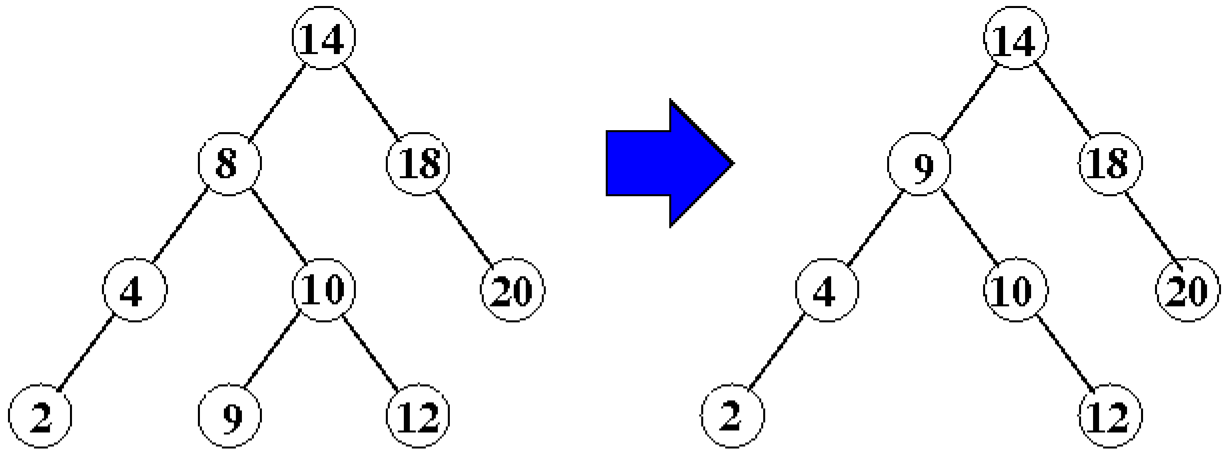
- *Delete 18 from this tree*



node has 2 children

- *Delete 8 from this tree*
- Join trees by replacing the node to deleted with a 'descendant' of the deleted node
  - the descendant is either:
    1. the **right child's** Deepest Left-Most Descendent (DLMD)
       - this node has the 'smallest' larger value than the deleted node
    2. the **left child's** Deepest Right-Most Descendent (DRMD)
       - this node has the 'largest' smaller value than the deleted node

We use the first strategy here.



Notice:

1. the DLMD of the right child is **9** and we replaced **8** by this node
2. we could have chosen instead the DRMD of the left child is **4**
   - ... which is 4, because 4 has no right descendents

Delete a node code

Here is the code that deletes an arbitrary node in a BST with 0, 1 or 2 children:

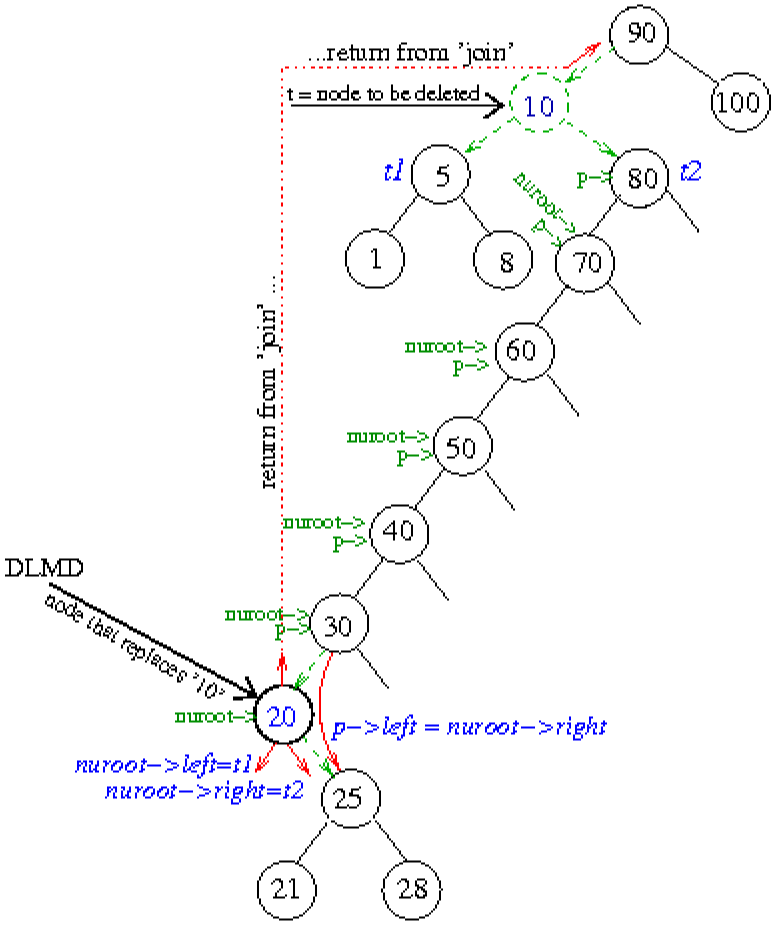
切换行号显示

```
Tree deleteTree(Tree t, int i){ // delete node with value 'v'
  if (t != NULL) {
    if (v < t->data) {
      t->left = deleteTree(t->left, v);
    }
    else if (v > t->data) {
      t->right = deleteTree(t->right, v);
    }
    else {     // v == t->data, so the node 't' must be deleted
      // next fragment of code violates style, just to make logic clear
      Tree n;                                         // temporary
      if (t->left==NULL && t->right==NULL) n=NULL;    // 0 children
      else if (t->left ==NULL)             n=t->right;   // 1 child
      else if (t->right==NULL)             n=t->left;    // 1 child
      else                                 n=joinDLMD(t->left,t->right);
      free(t);
      t = n;
    }
  }
  return t;
}
```

切换行号显示

```
// Joins t1 and t2 with the deepest left-most descendent of t2 as new root.
Tree joinDLMD(Tree t1, Tree t2){
   Tree nuroot;
   if (t1 == NULL) {         // actually should never happen
      nuroot = t2;
   }
```

```
    else if (t2 == NULL) {     // actually should never happen
       nuroot = t1;
    }
    else {                     // find the DLMD of the right subtree t2
       Tree p = NULL;
       nuroot = t2;
       while (nuroot->left != NULL) {
           p = nuroot;
           nuroot = nuroot->left;
       }                       // nuroot is the DLMD, p is its parent
       if (p != NULL){
           p->left = nuroot->right; // give nuroot's only child to p
           nuroot->right = t2;      // nuroot replaces deleted node
       }
       nuroot->left = t1;           // nuroot replaces deleted node
    }
    return nuroot;
 }
```



BST operations complexity analysis

Cost of searching:

- Best case: key is at root: O(1)
- Worst case: key is not in BST: search the height of the tree
  - Balanced trees: O(lg n)
  - Degenerate trees: O(n)
- Average case: key is in middle
  - Balanced tree: O(lg n)

- Degenerate trees: O(n)

Insertion and deletion:

- Always traverse height of tree
  - Balanced tree: O(lg n)
  - Degenerate tree: O(n)

Summary of operations

- A Binary Search Tree is an ordered binary tree:
  - *left child < parent < right child*
- Searching in BSTs
  - *int searchTree(Tree, int)*
- Inserting nodes into BSTs
  - *Tree createNode()*
  - *Tree insertTree(Tree, int)* recursive
  - *Tree insertTreeI(Tree, int)* iterative
  - *void freeTree()*
- Printing BSTs
  - laying a tree down 'on its side'
  - *printTree()*
- Characteristics of BSTs
  - *int height(Tree)*
  - *int count(Tree)*
  - *int balance(Tree)*
- Deletion from BSTs
  - *Tree deleteTree(Tree, int)* recursive
  - *Tree joinDLMD(Tree, Tree)* replace the node to be deleted with the DLMD
- Complexity analysis of BSTs

BinarySearchTrees (2019-08-05 11:53:06由AlbertNymeyer编辑)