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 根节点没有parent
- 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*

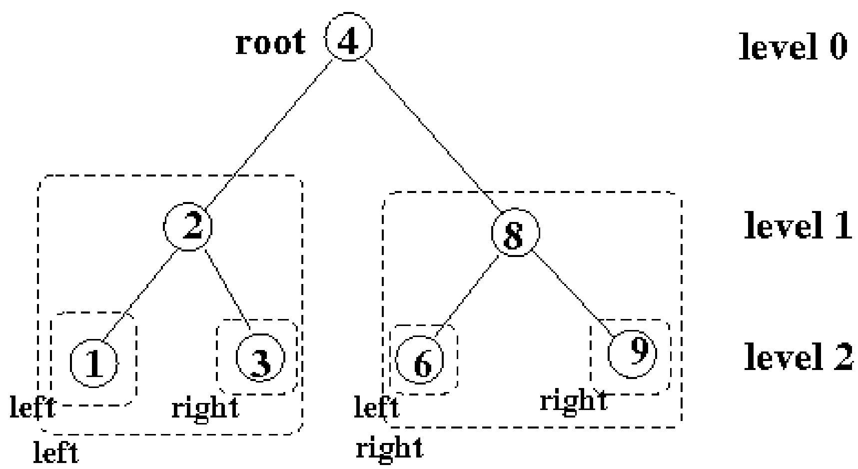
Formally, a tree is an acyclic graph in which each child has at most one parent.

Connections between nodes are called edges or links

- A **path** is a set of connected edges path只考虑向上或者向下
- 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*

Binary tree



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 都正好有两个孩子节点
    - each internal node has exactly 2 children
- *perfect binary* tree 叶子节点具有相同的深度
    - binary tree in which all leaves are at the same depth
- ***ordered binary tree***
    - left subtree values <= parent value 从左到右逐渐增大
    - right subtree values >= parent value
        - great for searching: e.g.
            - search for smallest: keep going left down the tree 最小值：左下
            - search for largest: keep going right down the tree 最大值：右下
            - 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

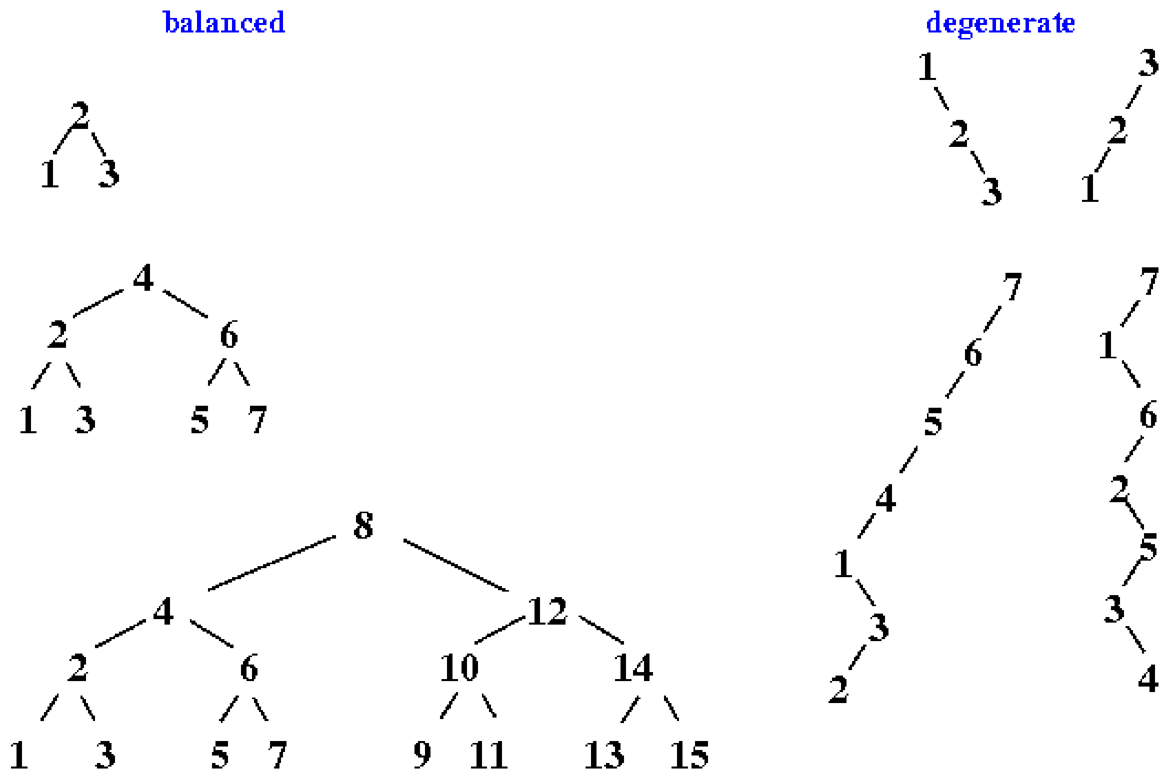
- **balanced**
  - tree has minimal height for the given number of nodes 给定节点数，tree有最小高度
  - number of nodes in the left and right subtrees differ by at most 1
- **degenerate** 退化，最高的
  - tree with maximal height (i.e. every parent has 1 child)

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)*

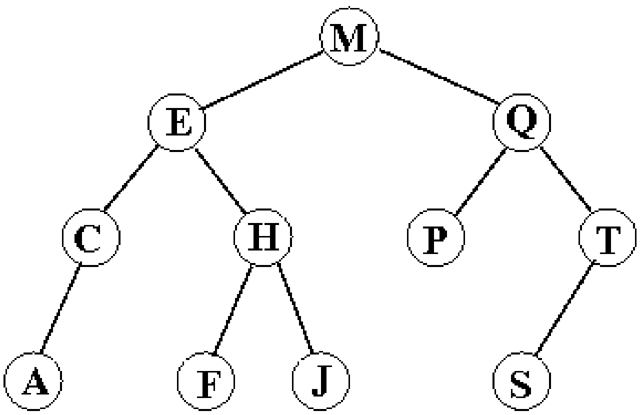
The maximum depth of any node in a tree is the height of the tree

Binary Search Tree

左child小于parent，小于右child

```
  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

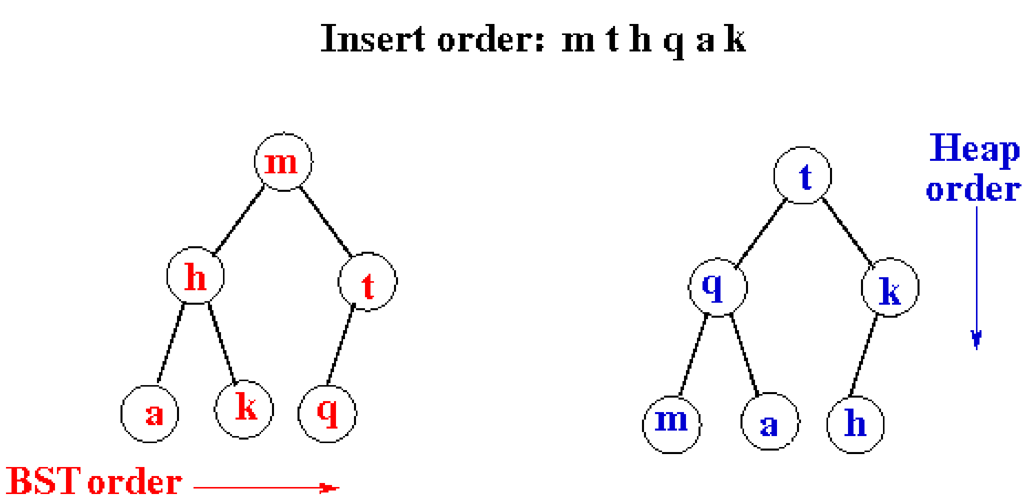
切换行号显示

```
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

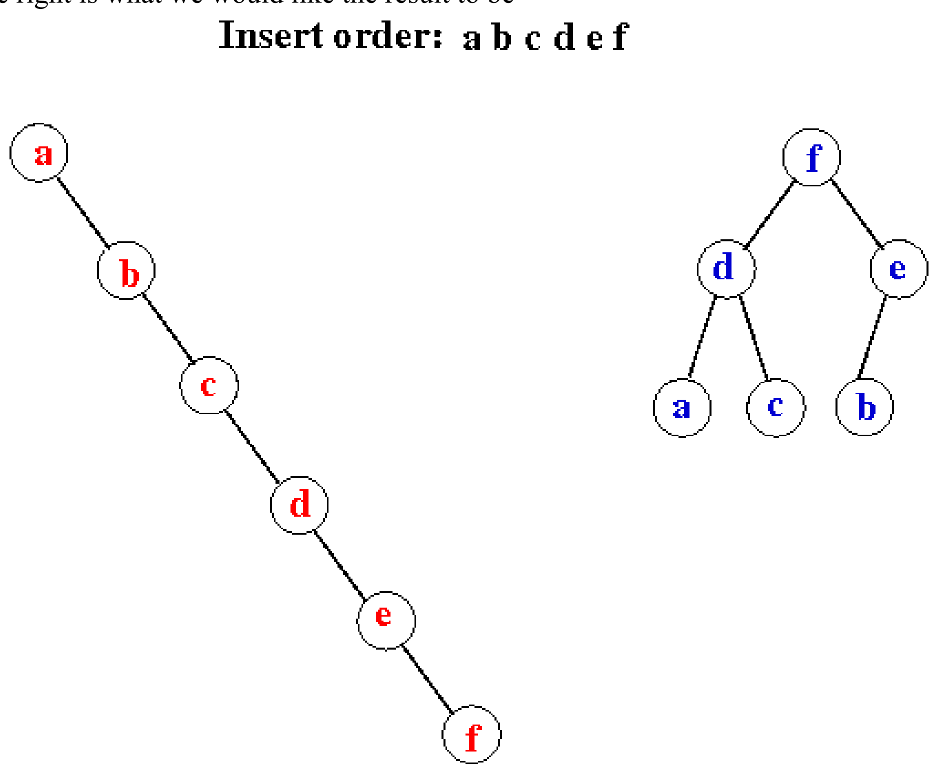
- the BST then has height *n*
- this is the worst-case behaviour
- binary search then performs as *O(n)*
    - this is just linear search, which is much slower (*remember the Sydney phone book analogy!*)

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



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

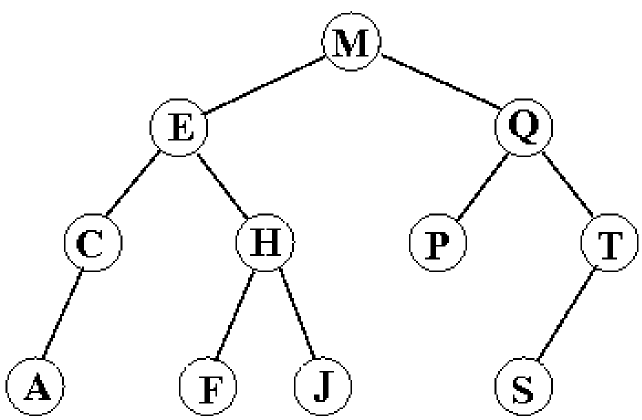
Assume a node contains a single key (and maybe some more data).

- 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(?)

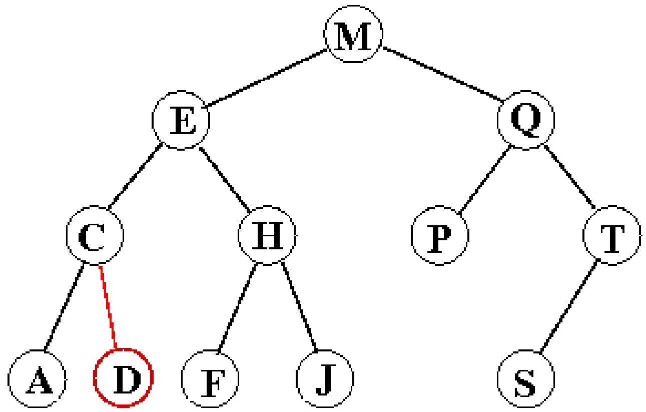
- **traversing** is similar (just follow the left or right link)
- **insertion**?
  - obvious in a linked list, but what strategy is used in a BST?
- **deletion**?
  - obvious in a linked list, but what happens to the 'children' in a BST?

In a BST, when we insert a new node:

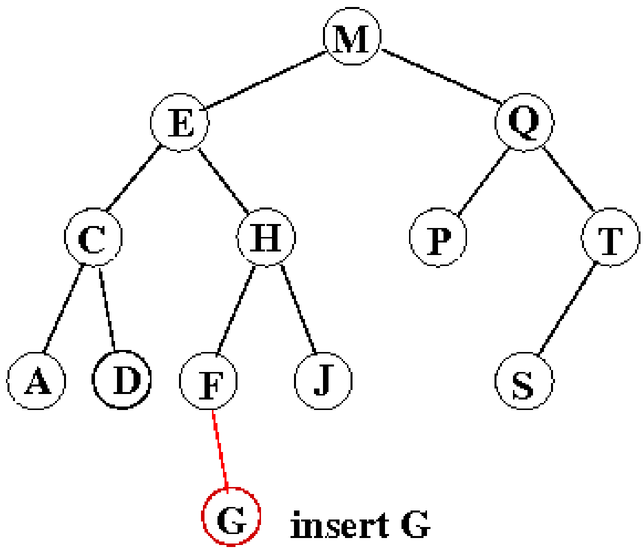
- it **always becomes a leaf** (impossible to insert a non-leaf node)
- it **always maintains the ordering** of the tree
  - it must be on the left of all nodes larger than it
  - it must be on the right of all nodes smaller than or equal to it

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



insert G

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);  首先建立一个根节点
   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;          递归有的时候不需要完全理解
}                                就是按照递归的思路去理解即可
                                 因为按照运行的思路去思考很复杂
Tree insertTree(Tree t, int v) {
   if (t == NULL) {
      t = createTree(v);  递归的终止条件就是：遇到了叶子节点的下一个
   }                      因此需要新建Tree，变成新的叶子节点
   else {
      if (v < t->data) { 如果v小于data，则需要插入到左子树中
        t->left = insertTree (t->left, v);
      }          此时，左子树变成了一颗新树，顶点为t->left
      else {     因此递归操作下去
        t->right = insertTree (t->right, v);
      }          同理，如果v大于data，则需要插入到右子树中
   }             此时右子树变成一颗新树，顶点为t->right
   return t;
}  返回的tree，其实就是节点指向了新的tree

Tree createTree (int v) { 创建一个树的节点，相当于叶子节点
   Tree t = NULL;          左右都是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;
}                  又是一个递归，终止条件就是到达叶子节点进行
                   数值打印
void printTree(Tree t) { // not the final version
    if (t != NULL) {          其实目的很简单就是打印树根节点
       printTree (t->left);   的值，从左支到右支
       printf ("%d  ", t->data);
       printTree (t->right);
    }
    return;
}

void freeTree(Tree t) { // free in postfix fashion
   if (t != NULL) {          就是为了free顶点
      freeTree(t->left);
      freeTree(t->right);
      free(t);          free的顺序就是先左，再右，再中间
   }
   return;
}
```

The output is:

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

But this does not look like a BST!

- *Where is the tree?*
  - *We had better work out a better way of printing it.*

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

Why did we use *infix*?

- Why not *prefix* or *postfix*?

What does the 'real' BST above look like?

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

How did this get converted to infix?

```
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  先left tree
            parent; print '1'        再 data
            right child of 1 is NULL 再right tree
          parent; print '2'
          right child is NULL
        parent; print '3'
        right child is NULL
      parent; print '4'
      right child is NULL
    parent; print '5'
    right child is NULL
  parent; print '6'
  right child is NULL
```

```
parent; print '7'
right child of 7 is 8
  left child of 8 is NULL
  parent; print '8'
  right child of 8 is NULL
```

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

- Consider a more complicated BST:

```
                34
           21          96
        14    26    98     110
     10      23      100
  4                   102
    8                  104
   6
    7
```

To print this would require nodes to be printed in 'row' fashion

- biggest number on the first line
- its potential 2 'neighbours' on the second line, with the correct number of spaces
- the potential 4 'next-neighbours' on the third line, again with the correct number of spaces
- the potential 8 'next-next-neighbours' on the 4th line, with the correct number of spaces
- etc

Tricky to get the children positioned correctly

- problem is that infix, prefix, postfix are all **depth-first** traversals
    - *keep taking a child until NULL is reached, then backtrack to parent ...*

The way we would like to print a BST is:

- print parent
    - print children
        - print grandchildren
            - print great-grandchildren
                - etc

That is, print level by level, which is a breadth-first search technique

We can 'cheat', though, by:

- printing each node on a separate line
    - that is easy, just add a newline in the *printf()*
- adding an indent that is equal to the depth of the node
    - need to increment and pass the depth into the recursive function
        - *printf()* the correct indentation

切换行号显示

```
void printTree(Tree t, int depth) { // extra depth parameter
    if (t != NULL) {
        depth++;          depth在一直递增，最后的值就是叶子的高度
        printTree (t->left, depth);
```

每向上一层，depth减1

```
            int i;
            for (i=1; i<depth; i++){ // print 'depth' ...
               putchar('\t');        // ... tabs
            }
            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

printTree(t, 0)

This is the BST lying on its side:

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

需要歪着头看

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;
}
```

递归获取数目，一次只获取一个
无限递加

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;
}
```

根节点+左右中的最大值

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

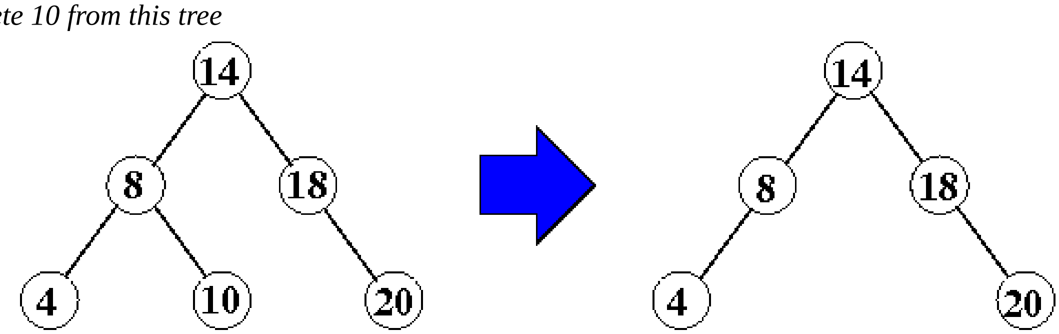
- Future searches ignore this item
- Problem? Tree can become full of 'deleted nodes'!

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*

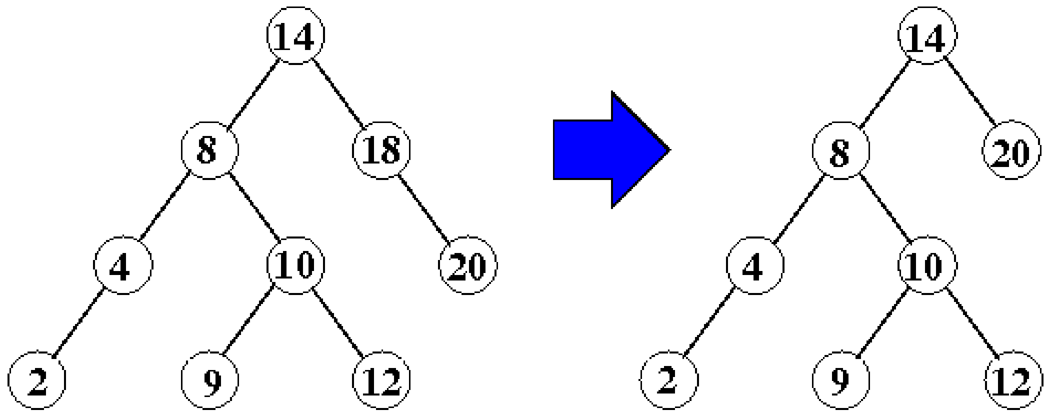
node is a leaf

- *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:
    - the Deepest Left-Most Descendent (DLMD) of the right child of the deleted node
      - being on the right, the DLMD has value greater than the deleted node, but it is the smallest such node 右子树的最左是比deleted node大的最小的
    - an alternative strategy is to use the Deepest Right-Most Descendent (DRMD) of the left child of the deleted node 左子树的最右是比deleted node小的最大的
      - the DRMD is the largest of the nodes that are smaller than the deleted node
  - here we use the first strategy: the ***DLMD replaces the node to be deleted***

思路就是找一个跟deleted node最接近得来代替之

右子树最左

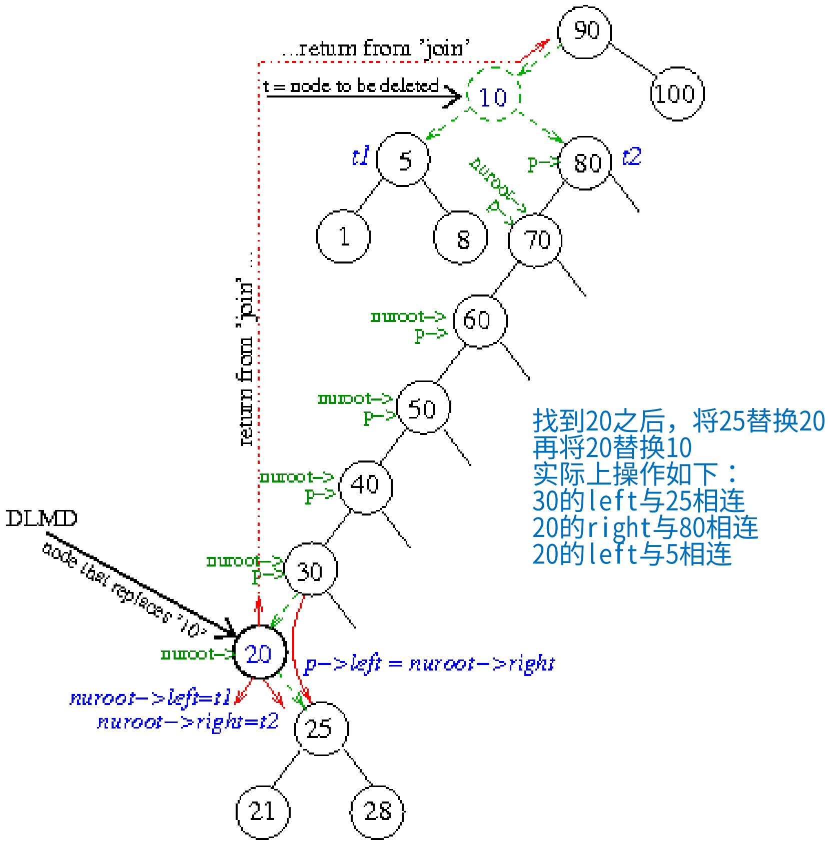
切换行号显示

```
Tree deleteTree(Tree t, int i){ // delete node with value 'i'
  if (t != NULL) {
    if (i < t->data) {
      t->left = deleteTree(t->left, i);
    }
    else if (i > t->data) {
      t->right = deleteTree(t->right, i);
    }
    else {    // i == 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; 没啥意义
   }
   else if (t2 == NULL) {    // actually should never happen
      nuroot = t1; 没啥意义
   }
   else {                    // find the DLMD of the right subtree t2
      Tree p = NULL;
      nuroot = t2;
      while (nuroot->left != NULL) {
          p = nuroot;
          nuroot = nuroot->left;
      }   p找到最左的节点的parent  // nuroot is the DLMD, p is its parent
          nuroot是最左的节点
      if (p != NULL){
          p->left = nuroot->right; // give nuroot's only child to p
          把右节点接到p的左边
          nuroot->right = t2;      // nuroot replaces deleted node
          nuroot要替换节点，因此需要右接t2
      }   左子树接t1
      nuroot->left = t1;           // nuroot replaces deleted node
   }
   return nuroot;
}
```



Binary Search Tree 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

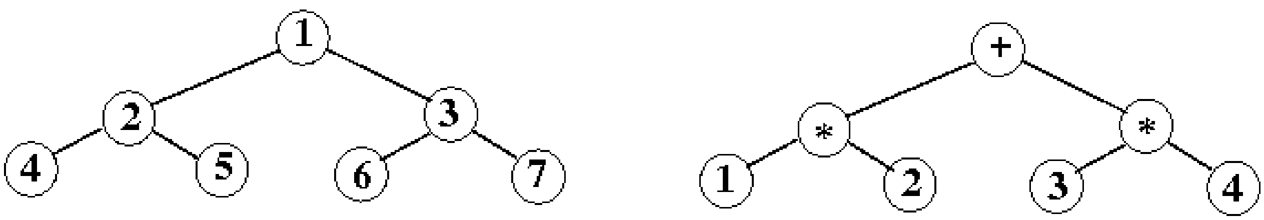
- A Binary Search Tree is an ordered binary tree:
  - *left child < parent < right child*
- Searching in BSTs
  - *int searchTree(Tree, int)*
- Inserting nodes into BSTs
  - *Tree createNode()*
  - *Tree insertTree(Tree, int)* recursive
  - *Tree insertTreeI(Tree, int)*
  - *void freeTree()*

- Printing BSTs
    - why must it be *infix*
    - going from input string to BST
    - *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

内容修改，下面内容添加到了下一节的内容中了

General tree traversals

Consider the following binary trees:



The tree on the right is called an *expression tree* because it represents an arithmetic expression

- the internal nodes in an expression tree are operators
- the leaf nodes in an expression tree are operands

There are many ways to traverse such trees:

1. **prefix** or NLR: visit **N**ode, then **L**eft subtree, then **R**ight subtree
    - it is called *prefix* because the operator *prefixes* (i.e. comes before) its operands

切换行号显示

```
void prefix(Tree t) {
   if (t != NULL) {
       printf("%c ", t->data);
       prefix(t->left);
       prefix(t->right);
   }
   return;
}
```

    - this will result in: *1 2 4 5 3 6 7* and *+ * 1 2 * 3 4* (resp. left and right)
2. **infix** or LNR: visit **L**eft subtree, then **N**ode, then **R**ight subtree
    - it is called *infix* because the operator is *in-between* its operands

切换行号显示

```
void infix(Tree t) {
   if (t != NULL) {
```

```
        infix(t->left);
        printf("%c ", t->data);
        infix(t->right);
    }
    return;
}
```

- will result in *4 2 5 1 6 3 7* and *1 * 2 + 3 * 4*

3. **postfix** or LRN: visit **L**eft subtree, then **R**ight subtree, then the **N**ode
   - it is called *postfix* because the operator is comes after its operands

切换行号显示

```
void postfix(Tree t) {
    if (t != NULL) {
        postfix(t->left);
        postfix(t->right);
        printf("%c ", t->data);
    }
    return;
}
```

- will result in *4 5 2 6 7 3 1* and *1 2 * 3 4 * +*

ASIDE: FUNCTION POINTERS

- Can 'generalise' the traversal and pass the 'action', i.e. a function to do something at the node, by using a function pointer.

切换行号显示

```
void generalInfix(Tree t, void (*dosomething)(int)){ // uses a fn ptr
   if (t != NULL){
      generalInfix(t->left, (*dosomething));
      dosomething(t->data);                        // infix!
      generalInfix(t->right, (*dosomething));
   }
   return;
}
```

The second parameter has type 'pointer to function':

- it is the address of a function (not a variable or struct)
- the function *infix* is called a *high-order function*
  - a high-order function is a function that:
    - has other functions as arguments or
    - returns a function as a result
- e.g. to declare *fname* as a function pointer:
  - *returnType (*fname)(types of arguments)*
- allows run-time choice of the function to 'dosomething'
- example of use

切换行号显示

```c
// funcptr.c
#include <stdio.h>
#include <stdlib.h>

void printint(int);

int main() {
   void (*fp)(int); // declare a function pointer 'fp'

   fp = printint;   // let the fn ptr be the printint() fn
   (*fp)(1);        // call 'fp' with arg '1'
   fp(2);           // call 'fp' with arg '2'
   return EXIT_SUCCESS;
}

void printint(int arg) {
   printf("%d\n", arg);
   return;
}
```

BST traversals

We can apply the above tree traversal techniques to BSTs.

BST infix order traversal

Here is an example of a program that traverses a BST:

- in infix order
- recursively
- using a fn ptr to keep the *infix()* function generic
- the BST is created in the order *4 2 6 3 7 1 5*
  - *what does this BST look like?*

切换行号显示

```c
// infixBST.c: traverse a BST in infix order
#include <stdio.h>
#include <stdlib.h>

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

void printint(int arg);
void infix(Tree, void (*dosomething)(int));
Tree createTree (int);
```

```
Tree insertTree(Tree, int);

int main(void) {
   Tree t = createTree(4);
   t = insertTree(t, 2);
   t = insertTree(t, 6);
   t = insertTree(t, 3);
   t = insertTree(t, 7);
   t = insertTree(t, 1);
   t = insertTree(t, 5);
   infix(t, printint);
   putchar('\n');
   return EXIT_SUCCESS;
}

void printint(int arg){
   printf("%d ", arg);
   return;
}

void infix(Tree t, void (*dosomething)(int)){ // uses a fn ptr
   if (t != NULL){
      infix(t->left, (*dosomething));
      dosomething(t->data); // in the middle, hence infix!
      infix(t->right, (*dosomething));
   }
   return;
}

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;
}
```

To compile, use:

```
gcc -Wall -Werror -O infixBST.c
```

The output is:

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

Notice:

- the program creates a BST and prints it in infix order
- the function *infix()* looks very similar to *printTree()* function, without indents
- *infix()* is recursive, so it uses the system stack (just like *prefix()* and *postfix()*)

Question: *Could you traverse a tree without recursion?*

- Answer: *yes, but then we'll need to use our own stack*

BST prefix-order traversal, recursively and non-recursively

Recall prefix order, visits the nodes in the following order *operator* -- *left child* -- *right child*

The following program traverses a BST in prefix order twice:

1. recursively (like *infix()* above)
2. non-recursively
    - we create and use our own stack (by including the Quack ADT)
    - there is a slight technical problem though
        - the stack can store only integers: we need a stack that stores the addresses of nodes
            - we use C's *casts* to convert between node addresses and integers

              切换行号显示

              ```
              push((int)t, stack);
              ```

              converts a node address *t* to an integer and

              切换行号显示

              ```
              t = (Tree)pop(stack);
              ```

              converts an integer back to a node address *t*
    - the child nodes are pushed in reverse order onto the stack
        - first push the right child, then push the left child
            - when we pop, we get the left child back first, then the right child

切换行号显示

```
// prefixBST.c: traverse a BST in prefix order recursively and non-recursively
#include <stdio.h>
#include <stdlib.h>
#include "quack.h"  // NOTE WE NEED THE QUACK ADT

typedef struct node *Tree;
```

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

void printint(int arg);
void prefixRec(Tree, void (*dosomething)(int));
void prefixNonRec(Tree, void (*dosomething)(int));
Tree createTree (int);
Tree insertTree(Tree, int);

int main(void) {
   Tree t = createTree(4);
   t = insertTree(t, 2);
   t = insertTree(t, 6);
   t = insertTree(t, 3);
   t = insertTree(t, 7);
   t = insertTree(t, 1);
   t = insertTree(t, 5);
   printf("Recursively: ");
   prefixRec(t, printint);
   printf("\nNonRecursively: ");
   prefixNonRec(t, printint);
   putchar('\n');
   return EXIT_SUCCESS;
}

void printint(int arg){
   printf("%d ", arg);
   return;
}

void prefixRec(Tree t, void (*dosomething)(int)){ // recursive prefix traversal
   if (t != NULL){
      dosomething(t->data);
      prefixRec(t->left, (*dosomething));
      prefixRec(t->right, (*dosomething));
   }
   return;
}

void prefixNonRec(Tree t, void (*dosomething)(int)){ // non-recursive traversal
   Quack stack = createQuack();
   push((int)t, stack);               // puts the root node on the stack
   while (!isEmptyQuack(stack)){      // now process the whole tree
      t = (Tree)pop(stack);           // pop the top node
      dosomething(t->data);           // 'visit' this node
      if (t->right != NULL) {         // push the right child of this node
         push((int)t->right, stack);
      }
      if (t->left != NULL) {          // push the left child
```

```
of this node
         push((int)t->left, stack);
      }
   }
}

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;
}
```

To compile we use:

```
gcc -Wall -Werror -O quackAR.c prefixBST.c
```

The output is:

```
Recursively: 4 2 1 3 6 5 7
NonRecursively: 4 2 1 3 6 5 7
```

The two traversals are the same, of course.

BST breadth-first traversal, non-recursively

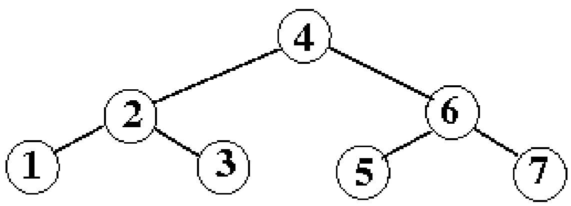
Above we saw how to traverse a BST:

- using recursive *infix*
- using recursive *prefix*
- using *prefix* and a stack
- the BST is created in the order *4 2 6 3 7 1 5*

The traversals *infix*, *postfix* and *prefix* are all *depth-first* techniques

- they contrast with a *breadth-first traversal*
    - a breadth-first traversal visits the nodes level-by-level, left to right
    - and from the root down to the leaves
- generally it is called a *Breadth-First Search* (BFS) as it is normally part of a search technique

Consider the following BST



The breadth-first order, reading left to right, top to bottom is *4 2 6 1 3 5 7*

Just like the *fix*-traversals, we need a data structure to remember nodes

- instead of a stack, however, we use a queue

The breadth-first function that traverses a tree is:

切换行号显示

```
void bfsTree(Tree t, void (*dosomething)(int)){
   Quack que = createQuack();
   qush((int)t, que);                 // puts the root node on the queue
   while (!isEmptyQuack(que)){        // now process the whole tree
      t = (Tree)pop(que);             // pop the top node
      dosomething(t->data);           // 'visit' this node
      if (t->left != NULL) {          // qush the left child of this node
         qush((int)t->left, que);
      }
      if (t->right != NULL) {         // qush the right child of this node
         qush((int)t->right, que);
      }
   }
}
```

Compare this function with the (non-recursive) *prefix()* function, reproduced below:

切换行号显示

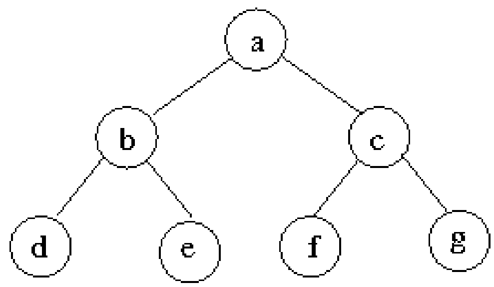
```
void prefixTree(Tree t, void (*dosomething)(int)){
   Quack stack = createQuack();
   push((int)t, stack);                 // puts the root node on the stack
   while (!isEmptyQuack(stack)){        // now process the whole tree
      t = (Tree)pop(stack);             // pop the top node
      dosomething(t->data);             // 'visit' this node
```

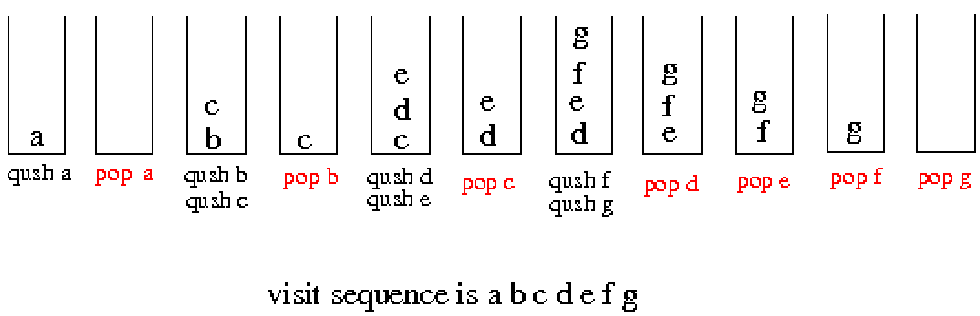
```
        if (t->right != NULL) {         // push the right child of this node
            push((int)t->right, stack);
        }
        if (t->left != NULL) {          // push the left child of this node
            push((int)t->left, stack);
        }
    }
}
```

If we replace the call to the *prefix* traversal by a call to *bfsTree()* in the program *prefixBST.c* above, and compile with the Quack ADT, the output will be:

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

To see what the BFS code is actually doing with the queue, consider the tree below, where we use letters instead of numbers for more clarity.





BinarySearchTrees (2019-08-01 17:52:14由AlbertNymeyer编辑)