Chapter 9  binary tree

Speaker: Lung-Sheng Chien

Reference book: Larry Nyhoff, C++ an introduction to data structures
Reference power point: Enijmax, Buffer Overflow Instruction
OutLine

• Binary search versus tree structure
• Binary search tree and its implementation
  - insertion
  - traversal
  - delete
• Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
• Pitfall: stack limit of recursive call
Recall linear search in chapter 6

- Data type of \textit{key} and \textit{base} are immaterial, we only need to provide comparison operator. In other words, framework of linear search is independent of comparison operation.

\begin{verbatim}
  pseudocode
  Given  array base[0:n-1] and a search key
  key and base may have different data type
  for  j = 0:1:n-1
      if base[j] == key then
          return location of base[j]
      endfor
  return  not-found
\end{verbatim}
linear search for structure-array

1. search **key** must be consistent with **keyval** in comparison operator, say **key** and **keyval** have the same data type, pointer to content of search key

2. **keytab[i]** must be consistent with *found_key*, they must be the same type and such type has **sizeof(keyType)** bytes
Since "endfor" is not a keyword, under linear search algorithm, we need to compare all keywords to reject "endfor". We need another efficient algorithm, binary search, which is the best.
step-by-step of binary search

13 28 35 49 62 66 80

(1) 13 28 35 49 62 66 80

(2) 13 28 35 49 62 66 80

(3) 13 28 35 49 62 66 80
step-by-step of binary search  

Equivalent tree structure

Question: Does binary-search work on sorted Linked-List?
Tree terminology [1]

- A tree consists of a finite set of elements called nodes and a finite set of directed arcs that connect pairs of nodes.

- “root” is one node without incoming arc, and every other node can be reached from root by following a unique sequence of consecutive arcs.

- Leaf node is one node without outgoing arc.

- child node is successor (繼承者) of parent node, parent node is predecessor (被繼承者) of child node

- Children with the same parent are siblings (兄弟姐妹) of each other
Tree terminology

- **Root**: The topmost node in the tree.
- **Leaf**: A node with no children.
- **Right subtree of root**: The subtree rooted at the right child of the root.
- **Incoming arc**: An arc pointing towards a node.
- **Outgoing arc**: An arc pointing away from a node.
- **Parent**: A node that has at least one child.
- **Child**: A node that has at least one parent.
- **Siblings**: Nodes that have the same parent.
OutLine

• Binary search versus tree structure
• **Binary search tree and its implementation**
  - insertion
  - traversal
  - delete
• Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
• Pitfall: stack limit of recursive call
Binary Search Tree (BST)

- Collection of data elements (data storage)
  a binary tree in which for each node $x$:
  value in left child of $x$ $\leq$ value in $x$ $\leq$ value in right child of $x$

- Basic operations (methods)
  - construct an empty BST
  - determine if BST is empty
  - search the BST for a given item
  - Insert a new item in the BST and maintain BST property
  - delete an item from the BST and maintain BST property
  - Traverse the BST and visit each node exactly once. At least one of the traversals, called an inorder traversal, must visit the values in the nodes in ascending order
Variant of BST

- **Treap**: a binary search tree that orders the nodes by adding a random priority attribute to a node, as well as a key. The nodes are ordered so that the keys form a binary search tree and the priorities obey the max heap order property.
- **red-black tree**: a type of self-balancing binary search tree, a data structure used in computer science, typically used to implement associative arrays.
- **Heap**: a specialized tree-based data structure that satisfies the heap property: if $B$ is a child node of $A$, then $\text{key}(A) \geq \text{key}(B)$.
- **AVL tree**: a self-balancing binary search tree.
- **B-tree**: a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and filesystems.
- threaded binary tree: possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.
Requirement of BST

- **treeEle**: data type
- type of physical storage: linked-list
- ordered mechanism: depends on `treeEle`
- pointer to `root` node

Methods of structure `BST`

- `BST* BST_init( void )`
- `int empty( BST* )`
- `int search( BST*, treeEle )`
- `void insert( BST*, treeEle )`
- `void remove( BST*, treeEle )`
- `void traverse( BST* )`
BST.h

```c
#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

typedef int treeEle; // integer binary tree

typedef struct BinNode {
    treeEle data;
    BinNode *left;
    BinNode *right;
} BinNode;

typedef BinNode* BinNodePtr;

typedef struct {
    BinNodePtr root;
} BST;

// allocate a new BinNode with data = val
BinNodePtr newBinNode( treeEle val );

// construct an empty BST
BST* BST_init( void );

// return 1 if BST is empty
// 0 otherwise
int empty( BST* );

// return 1 if item is in the BST
// 0 otherwise
int search( BST* tree, treeEle item );

// insert a new item in the BST and maintain BST property
void insert( BST* tree, treeEle item );

// traverse BST: inorder LMD
void traverse_inorder( BST* tree );
    void traverse_inorder_aux( BinNodePtr node );
#endif // BINARY_SEARCH_TREE
```

Linked-List BST: header file

Type of physical storage: linked-List

pointer to root node

constructor of tree node (leaf node)

Methods of structure BST
BST method: constructor (建構子)

BST.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "BST.h"

// allocate a new BinNode with data = val
BinNodePtr newBinNode( treeEle  \text{ val } )
{
    BinNodePtr node = (BinNodePtr) malloc(sizeof(BinNode));
    assert(node);
    node->data = \text{ val };
    node->left = NULL;
    node->right = NULL;
    return node;
}

// construct an empty BST
BST* BST_init( void )
{
    BST* tree = (BST*) malloc(sizeof(BST));
    assert(tree);
    tree->root = NULL;  \text{ empty tree}
    return tree;
}
```

Construct leaf node

Data encapsulation: user does not see function `newBinNode`
**BST method: binary search**

**BST.cpp**

```c
// return 1 if BST is empty
// 0 otherwise
int empty( BST *tree )
{
    assert( tree );
    return (NULL == tree->root);
}

// return 1 if value is in the BST
// 0 otherwise
int search( BST *tree, treeEle item )
{
    assert( tree );
    BinNodePtr locPtr = tree->root;
    int found = 0;
    while(1) {
        if ( found || (NULL == locPtr) ) { break; }
        if ( item < locPtr->data ){
            locPtr = locPtr->left;
        }else if ( item > locPtr->data ){
            locPtr = locPtr->right;
        }else{
            found = 1;
        }
    }
    return found;
}
```

[Diagram of binary search tree]

- `item < data` (go left)
- `item > data` (go right)
- `left-subtree`
- `right-subtree`
OutLine

• Binary search versus tree structure
• Binary search tree and its implementation
  - insertion
  - traversal
  - delete
• Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
• Pitfall: stack limit of recursive call
BST method: insert “64” into tree

64 > 49, descend to right subtree
BST method: insert “64” into tree

64 < 66, descend to left subtree

64 > 62, descend to right subtree

“64” is NOT in the tree
BST method: insert “64” into tree

- **Step 1:** locate where a given item is to be inserted and set its parent node to pointer `parent`.
- **Step 2:** construct a leaf node with data = “64” and attach to node pointed by pointer, `parent`.

New BinNode
BST method: insert

```cpp
// insert a new item in the BST and maintain BST property
void insert( BST *tree, treeEle item )
{
    assert( tree );
    BinNodePtr locPtr = tree->root;
    BinNodePtr parent = NULL;
    int found = 0;
    while(1) {
        if ( found || (NULL == locPtr) ) { break ; }
        parent = locPtr;
        if ( item < locPtr->data ){
            locPtr = locPtr->left ;
        }else if ( item > locPtr->data ){
            locPtr = locPtr->right ;
        }else{
            found = 1 ;
        }
    }
    if ( found ){
        printf("Item already in the tree \n");
    }else{
        locPtr = newNode( item ) ;
        if ( NULL == parent ){ // empty tree
            tree->root = locPtr ;
        }else if ( item < parent->data ){ // insert to left child
            parent->left = locPtr ;
        }else{ // // insert to right child
            parent->right = locPtr ;
        }
    }
}
```

**Question:** why need we to compare item and parent->data again in step 2?
Outline

- Binary search versus tree structure
- **Binary search tree and its implementation**
  - insertion
  - traversal
  - delete
- Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
- Pitfall: stack limit of recursive call
Recursive definition of a binary tree

- A binary tree is either empty or consists of a node called the root, which has pointers to two disjoint binary subtrees called the left subtree and right subtree.

**BST.cpp**

```c
void traverse_inorder( BST *tree )
{
    assert( tree );
    traverse_inorder_aux( tree->root );
    printf(\n"n");
}

void traverse_inorder_aux( BinNodePtr node )
{
    if ( NULL == node ) { return ; }

    traverse_inorder_aux( node->left ) ;
    printf("%d ", node->data ) ;
    traverse_inorder_aux( node->right ) ;
}
```

- **In-order traversal**
  - Traverse the left subtree
  - Visit the root and process its content
  - Traverse the right subtree

Termination condition
Inorder traversal

Here root means staring node of any tree

output

(1) goto left subtree of node 49

(2) goto left subtree of node 28

(3) goto left subtree of node 13
Inorder traversal

(4) root is NULL, output 13
goto right subtree of node 13

(5) root is NULL, all children of node 13 have been visited,
go back to node 28

(6) output node 28,
goto right subtree of node 28

(7) goto left subtree of node 35
Inorder traversal

(8) root is NULL, output 35, goto right subtree of node 35

(9) root is NULL, all children of node 35 have been visited, go back to node 28

(10) All children of node 28 have been traversed, go back to node 49

(11) left-subtree of node 49 have been traversed, output 49 and goto right subtree
(12) goto left subtree of node 66

(13) goto left subtree of node 62

(14) root is NULL, output 62, goto right subtree of node 62

(15) All children of node 62 have been visited, go back to node 66

(16) Let subtree of node 66 is visited, output 66 and goto right subtree of node 66
Inorder traversal

(17) goto left subtree of node 80

(18) root is NULL, output 80 and
goto right subtree of node 80

(19) All children of node 80 have
been visited,
go back to node 66

(20) All children of node 66 have
been visited,
go back to node 49
Inorder traversal

(21) All children of node 49 have been visited, terminate

Inorder in BST is ascending order, why?
Driver for Inorder traversal

main.cpp

```c
#include <stdio.h>
#include "BST.h"

int main( int argc, char *argv[])
{
    BST* tree = BST_init();  // 1
    if (empty(tree)) {
        printf("Tree is empty \n");
    }
    insert( tree, 49 );
    insert( tree, 28 );
    insert( tree, 13 );
    insert( tree, 35 );
    insert( tree, 66 );
    insert( tree, 62 );
    insert( tree, 80 );
    traverse_inorder( tree );  // 3
    if (search(tree, 80)){
        printf("80 is in the tree\n");
    } else if (!search(tree, 12)){
        printf("12 is NOT in the tree\n");
    } return 0 ;
}
```

```
Tree is empty
13 28 35 49 62 56 80
80 is in the tree
12 is NOT in the tree
Press any key to continue.
```
Driver for Inorder traversal

insert(tree, 66)

insert(tree, 62)

insert(tree, 80)
Exercise

- Implement integer BST with methods `newBinNode`, `BST_init`, `empty`, `search`, `insert` as we discuss above and write a method (function) to show configuration of BST as follows.

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>left node</th>
<th>right node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804b888</td>
<td>49</td>
<td>0x804b898</td>
<td>0x804b8c8</td>
</tr>
<tr>
<td>0x804b898</td>
<td>28</td>
<td>0x804b8a8</td>
<td>0x804b8b8</td>
</tr>
<tr>
<td>0x804b8a8</td>
<td>13</td>
<td>(nil)</td>
<td>(nil)</td>
</tr>
<tr>
<td>0x804b8b8</td>
<td>35</td>
<td>(nil)</td>
<td>(nil)</td>
</tr>
<tr>
<td>0x804b8c8</td>
<td>66</td>
<td>0x804b8d8</td>
<td>0x804b8e8</td>
</tr>
<tr>
<td>0x804b8d8</td>
<td>62</td>
<td>(nil)</td>
<td>(nil)</td>
</tr>
<tr>
<td>0x804b8e8</td>
<td>80</td>
<td>(nil)</td>
<td>(nil)</td>
</tr>
</tbody>
</table>
Exercise

- Use recursive call to implement methods `search` and `insert`.
- Write a method to compute maximum depth of a BST.

What is topology of a BST created by inserting 13, 28, 35, 49, 62, 66, 80 in turn.
- Can you modify an unbalanced BST into a balanced one?
OutLine

• Binary search versus tree structure
• **Binary search tree and its implementation**
  - insertion
  - traversal
  - delete
• Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
• Pitfall: stack limit of recursive call
Delete a node \( x \) from BST \[1\]

case 1: \( x \) is a leaf node
Delete a node $x$ from BST

case 2: $x$ has one child

```plaintext
G
/    |
F    J
/    /|
A    H  O
/    /  |
C    I   M
/    /  |
B    K   N
   /    |
  D    L
```

```plaintext
Delete
```

```
G
/    |
F    J
/    /|
A    H  O
/    /  |
C    I   M
/    /  |
B    K   N
   /    |
  D    L
```

```plaintext
case 2: x has one child
```

```
G
/    |
F    J
/    /|
A    H  O
/    /  |
C    I   M
/    /  |
B    K   N
   /    |
  D    L
```

```plaintext
Delete
```

```
G
/    |
F    J
/    /|
A    H  O
/    /  |
C    I   M
/    /  |
B    K   N
   /    |
  D    L
```

```plaintext
case 2: x has one child
```
Delete a node $x$ from BST

Case 3: $x$ has two children

Replace $x$ with its inorder successor $xsucc$
Delete a node $x$ from BST

- delete $xsucc$
BST method: remove item

```c
void remove(BST *tree, treeEle item)
{
    int found;
    BinNodePtr x, parent;
    found = search2( tree, item, &x, &parent);
    if (!found)
    {
        printf("Item is not in the BST \n");
        return;
    }
    if ( (NULL != x->left) && (NULL != x->right) )
    {
        // x has two children
        // find x's inorder successor and its parent
        BinNodePtr xsucc = x->right;
        parent = x;
        while( NULL != xsucc->left )
        {
            parent = xsucc;
            xsucc = xsucc->left;
        }
        // move content of xsucc to x and change x to
        // point to successor which will be deleted
        x->data = xsucc->data;
        x = xsucc;
    } // if x has two children
    // proceed with case where node x has 0/1 child
    BinNodePtr subtree = x->left;
    if ( NULL == subtree )
    {
        subtree = x->right;
    }
    if ( NULL == parent )
    {
        tree->root = subtree;
    } else if ( x == parent->left )
    {
        parent->left = subtree;
    } else
    {
        parent->right = subtree;
    }
    free(x);
}
```

// locates node containing an item and its parent
```c
int search2( BST *tree, treeEle item,
              BinNodePtr *locPtr, BinNodePtr *parent )
{
    assert( tree );
    *locPtr = tree->root;
    *parent = NULL;
    int found = 0;
    while(1) {
        if ( !found || (NULL == locPtr) ) { break ; }
        if ( item < (*locPtr)->data ){
            *parent = *locPtr;
            *locPtr = (*locPtr)->left ;
        } else if ( item > (*locPtr)->data ){
            *parent = *locPtr;
            *locPtr = (*locPtr)->right ;
        } else{
            Found = 1 ;
        }
    }
    return found ;
}
```
Exercise

- Implement method \textit{remove} and write a driver to test it, you can use following BST as test example. Note: you need to test all boundary cases.

- Use recursive call to implement methods \textit{remove}.
Exercise

- Construct following expression tree (note that you may need general binary tree, not BST) and show its configuration.

- Show result of pre-order (prefix), in-order (infix) and post-order (postfix) respectively.

\[(a + b) \times (c / (d - e))\]
OutLine

• Binary search versus tree structure
• Binary search tree and its implementation
  - insertion
  - traversal
  - delete
• Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
• Pitfall: stack limit of recursive call
Convert RPN expression to expression tree

15 + 841 − − x

Create leaf node 1 and push address onto stack

5 + 841 − − x

Create leaf node 5 and push address onto stack

+841 − − x

Create node “+” and pop 5, 1 from stack as its children.

841 − − x

Create leaf node 8 and push address onto stack
Convert RPN expression to expression tree [2]

<table>
<thead>
<tr>
<th>expression</th>
<th>stack</th>
<th>comments</th>
<th>Binary tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>41 - - x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>top</td>
<td>Create leaf node 4 and push address onto stack</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 - - x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>top</td>
<td>Create leaf node 1 and push address onto stack</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-- x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>top</td>
<td>Create node “-” and pop 1, 4 from stack as its children.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>top</td>
<td>Push node ‘-’ onto stack</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Convert RPN expression to expression tree [3]

<table>
<thead>
<tr>
<th>expression</th>
<th>stack</th>
<th>comments</th>
<th>Binary tree</th>
</tr>
</thead>
</table>
| − ×        | +     | Create node “−” and pop “−”, 8 from stack as its children. | $egin{array}{c} + \\
\downarrow \\
1 \quad 5 \\
\downarrow \\
* \\
\downarrow \\
- \\
\downarrow \\
+ \quad 5 \\
\downarrow \\
8 \quad 8 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\downarrow \\
- \quad 1 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\end{array}$ |
| −          | -     | Push node “−” onto stack | $egin{array}{c} - \\
\downarrow \\
+ \quad 1 \\
\downarrow \\
5 \quad 5 \\
\downarrow \\
8 \quad 8 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\end{array}$ |
| ×          | +     | Create node “×” and pop “−”, “+” from stack as its children. | $egin{array}{c} * \\
\downarrow \\
+ \quad 1 \\
\downarrow \\
5 \quad 5 \\
\downarrow \\
8 \quad 8 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\end{array}$ |
| ×          | −     | Push node “×” onto stack | $egin{array}{c} - \\
\downarrow \\
+ \quad 1 \\
\downarrow \\
5 \quad 5 \\
\downarrow \\
8 \quad 8 \\
\downarrow \\
- \quad 4 \\
\downarrow \\
- \quad 1 \\
\end{array}$ |

Only one address on the stack, this address is root of the tree
Exercise

• Depict flow chart of “convert RPN expression to expression tree”.
• Write program to do “convert RPN expression to expression tree”, you can use following expression tree as test example.
• Use above binary tree to evaluate result (stack free, just traverse the binary tree).

Infix: \((1 + 5) \times (8 - (4 - 1))\)

Postfix: 1 5 + 8 4 1 − − ×

Parenthesis free
OutLine

- Binary search versus tree structure
- Binary search tree and its implementation
  - insertion
  - traversal
  - delete
- Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
- Pitfall: stack limit of recursive call
Stack allocation in VC2005

- A function’s prolog (prolog code sequence 起始設定) is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.
- The parameter area is always at the bottom of the stack, so that it will always be adjacent to the return address during any function call.
- The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function Types for a certain class of frame functions.
- When you define a local variable, enough space is allocated on the stack frame to hold the entire variable, this is done by compiler.
- Frame variables are automatically deleted when they go out of scope. Sometimes, we call them automatic variables.
Stack frame by g++

```cpp
int foo( int level, int a, int b );

int main( int argc, char* argv[] )
{
    int level = 3;
    int a = 2;
    int b = 3;
    foo( level, a, b );
    return 0;
}

int foo( int level, int a, int b )
{
    int x;
    x = a + b;
    if ( x >= level ) { return x; }
    return b * foo( level - 1, a-1, b-1 );
}
```

caller: 呼叫者, 如 main
callee: 被呼叫者, 如 foo

Low address

<table>
<thead>
<tr>
<th>local variables of callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
</tr>
</tbody>
</table>

Current base pointer

<table>
<thead>
<tr>
<th>base pointer of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack order</td>
</tr>
</tbody>
</table>

4byte

<table>
<thead>
<tr>
<th>return address of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
</tr>
</tbody>
</table>

4byte

<table>
<thead>
<tr>
<th>function Parameter (right to left)</th>
</tr>
</thead>
<tbody>
<tr>
<td>high address</td>
</tr>
</tbody>
</table>

ebp

0xbfffed04

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
</table>

0xbfffed08

0xbfffed0c

0xbfffed10

0xbfffed14

0xbfffed18

0xbfffed38

0x80484fc

level = ebp[8]
a = ebp[12]
b = ebp[16]
```c
int foo( int level, int a, int b )
{
    int x;
    x = a + b;
    if ( x == level ) { return x; }
    return b * foo( level - 1, a - 1, b - 1 );
}

int main( int argc, char* argv[] )
{
    int level = 3;
    int a = 2;
    int b = 3;
    foo( level, a, b );
    return 0;
}
```
Actions to call a function

- Caller push parameters of callee to stack
- Caller execute command `call`, for example “call _Z3fooii”.
  - push return address (address of caller) to stack
  - program counter points to function code address
- In callee
  - push old `ebp` (base pointer of caller) to stack
  - copy `esp` to `ebp` (ex: `movl %esp, %ebp`)
  - reserve enough space for local variables
- When function return to caller
  - callee move `sp (stack pointer)` to return address
  - callee execute command `ret`, and then program counter points to return address
  - caller pop base pointer to restore original status
Cost to call a function

- Function calls (including parameter passing and placing object’s address on the stack)
- Preservation of caller’s stack frame
- Return-value communication
- Old stack-frame restore
- Return (give program control back to caller)

- recursive call is easy to implement and code size is minimum, however we need to pay a little overhead. That’s why we do not like recursive call when dealing with computational intensive task.

**Exercise:** write quick sort with recursive version and non-recursive version, then compare performance between them.
Exercise

• Modify following code to show address of function parameter, local variable and content of return address, base pointer. Use “g++ -O0” to compile your code on workstation and check configuration of stack frame.
• What is configuration of stack frame using icpc –O0 ?
• What is configuration of stack frame in VC6.0 ?
• Is configuration of stack frame the same for each execution? Why?
• What’s size of function prolog for compiler g++, icpc and vc6?

```c
int foo( int level, int a, int b )
{
    int x ;
    x = a * b ;
    if ( 0 >= level ) { return x ; }
    return b * fon( level - 1, a-1, b-1 ) ;
}
```

```c
int main( int argc, char* argv[])
{
    int level = 3 ;
    int a = 2 ;
    int b = 3 ;
    foo( level, a, b ) ;
    return 0 ;
}
```

```c
int foo( int level, int a, int b )
{
    int level = 3 ;
    int a = 2 ;
    int b = 3 ;
    foo( level, a, b ) ;
    return 0 ;
}
```
Stack limit

- In RedHat 9, 32-bit machine, default stack size is 8MB. Use command “ulimit -a” to show this information.

- Visual studio C++ 6.0, default stack size is 1MB
Test stack limit in VC6.0

```c
#include <stdio.h>
#define MAXBUFFER 1024

void foo( int level )
{
    int argc, argv[];
    return 0;
}

void foo( int level )
{
    char word[MAXBUFFER]; // 1kB buffer in stack frame
    if ( level <= 0 ) { return ; }
    printf("word[last] = %d, level = %d\n", word[MAXBUFFER-1], level);
    foo(level-1);
}

word[last] = ? level = 118
word[last] = ? level = 117
word[last] = ? level = 116
word[last] = ? level = 115
word[last] = ? level = 114
word[last] = ? level = 113
word[last] = ? level = 112
word[last] = ? level = 111
word[last] = ? level = 110
word[last] = ? level = 109
word[last] = ? level = 108
word[last] = ? level = 107
word[last] = ? level = 106
word[last] = ? level = 105
word[last] = ? level = 104
word[last] = ? level = 103
word[last] = ? level = 102
word[last] = ? level = 101
word[last] = ? level = 100
word[last] = ? level = 99
word[last] = ? level = 98
word[last] = ? level = 97
word[last] = ? level = 96
word[last] = ? level = 95
Press any key to continue
```

Recursive call

Level number cannot reach 1 since stack overflow
modify stack limit in VC6.0
Exercise

• Write driver to test stack limit in VC6.0 and modify stack size in project setting dialog, does it work?

• Use the same driver, test stack limit on workstation with compiler g++ and icpc respectively. Is stack size independent of compiler?

• If we modify function foo such that local variable word is of no use what’s stack size on workstation?

```c
void foo(int level)
{
    char word[MAXBUFFER]; // 1kB buffer in stack frame
    if ( 0 <= level ) { return ; }
    printf("level = %d\n", level);
    foo( level-1 ) ;
}
```

Local variable word is of no use.