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

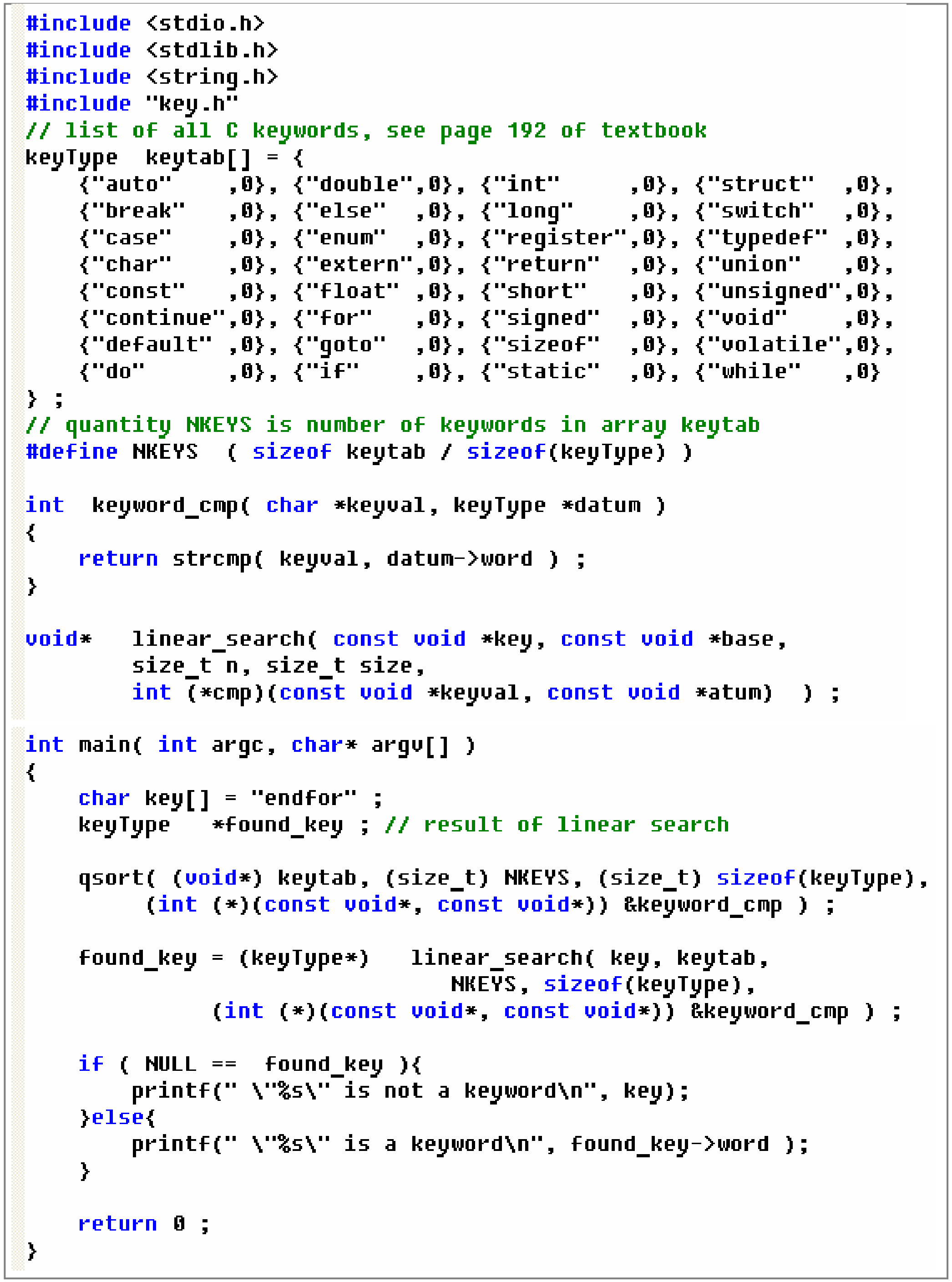
- Data type of ***key*** and ***base*** are immaterial, we only need to provide comparison operator. In other words, framework of linear search is independent of comparison operation.

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
```

User-defined comparison operation

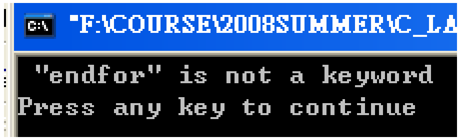
linear search for structure-array



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "key.h"
// list of all C keywords, see page 192 of textbook
keyType  keytab[] = {
    {"auto"    ,0}, {"double",0}, {"int"     ,0}, {"struct"  ,0},
    {"break"   ,0}, {"else"  ,0}, {"long"    ,0}, {"switch"  ,0},
    {"case"    ,0}, {"enum"  ,0}, {"register",0}, {"typedef" ,0},
    {"char"    ,0}, {"extern",0}, {"return"  ,0}, {"union"   ,0},
    {"const"   ,0}, {"float" ,0}, {"short"   ,0}, {"unsigned",0},
    {"continue",0}, {"for"   ,0}, {"signed"  ,0}, {"void"    ,0},
    {"default" ,0}, {"goto"  ,0}, {"sizeof"  ,0}, {"volatile",0},
    {"do"      ,0}, {"if"    ,0}, {"static"  ,0}, {"while"   ,0}
} ;
// quantity NKEYS is number of keywords in array keytab
#define NKEYS  ( sizeof keytab / sizeof(keyType) )

int  keyword_cmp( char *keyval, keyType *datum )
{
    return strcmp( keyval, datum->word ) ;
}

void*   linear_search( const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *atum)  ) ;

int main( int argc, char* argv[] )
{
    char key[] = "endfor" ;
    keyType   *found_key ; // result of linear search

    qsort( (void*) keytab, (size_t) NKEYS, (size_t) sizeof(keyType),
         (int (*)(const void*, const void*)) &keyword_cmp ) ;

    found_key = (keyType*)   linear_search( key, keytab,
                                NKEYS, sizeof(keyType),
              (int (*)(const void*, const void*)) &keyword_cmp ) ;

    if ( NULL ==  found_key ){
        printf(" \"%s\" is not a keyword\n", key);
    }else{
        printf(" \"%s\" is a keyword\n", found_key->word );
    }

    return 0 ;
}
```

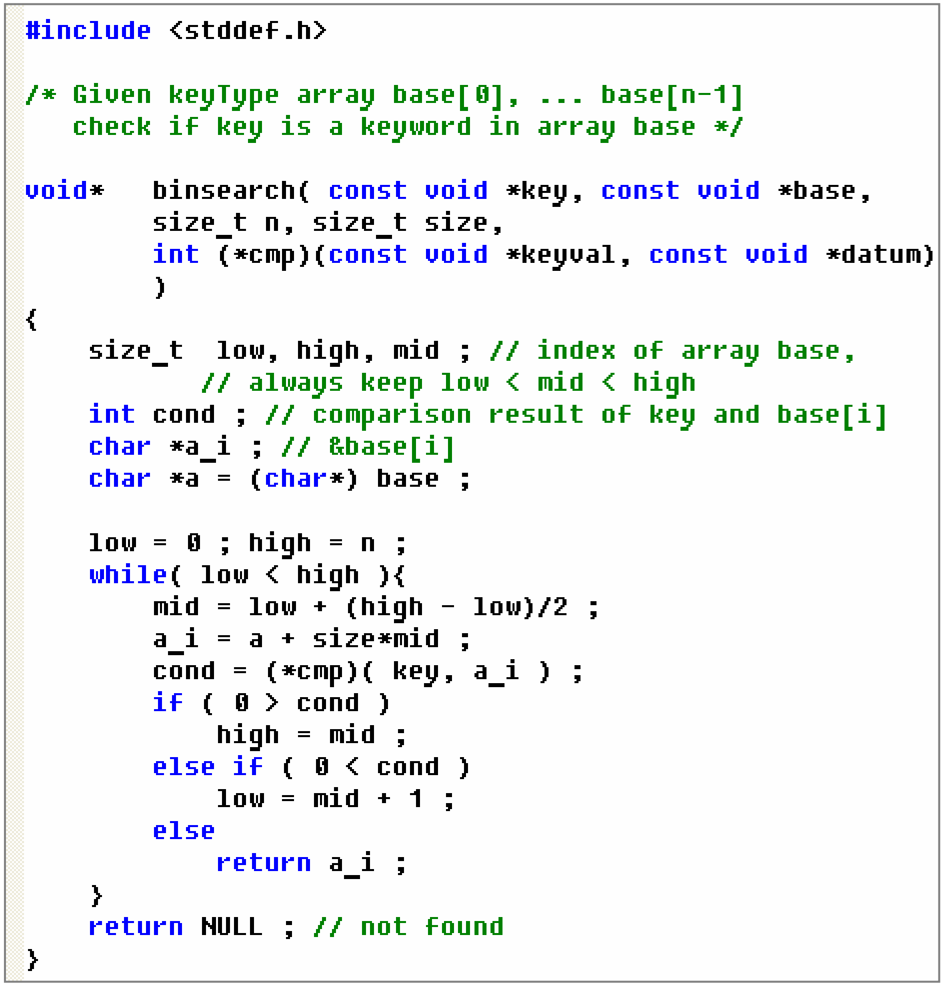
1. search ***key*** must be consistent with ***keyval*** in comparison operator, say ***key*** and ***keyval*** have the same data type, pointer to content of search key

2. ***keytab[i]*** must be consistent with ****found_key***, they must be the same type and such type has ***sizeof(keyType)*** bytes

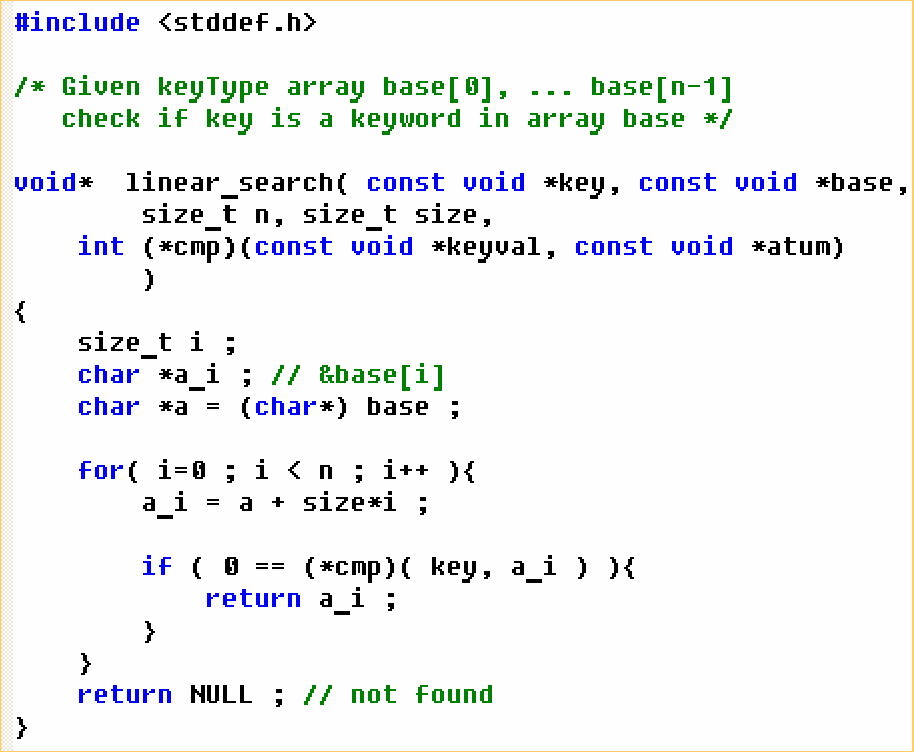


```
 "endfor" is not a keyword
Press any key to continue
```

binary search in chapter 6



```
#include <stddef.h>

/* Given keyType array base[0], ... base[n-1]
   check if key is a keyword in array base */

void*   binsearch( const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *datum)
        )
{
    size_t  low, high, mid ; // index of array base,
           // always keep low < mid < high
    int cond ; // comparison result of key and base[i]
    char *a_i ; // &base[i]
    char *a = (char*) base ;

    low = 0 ; high = n ;
    while( low < high ){
        mid = low + (high - low)/2 ;
        a_i = a + size*mid ;
        cond = (*cmp)( key, a_i ) ;
        if ( 0 > cond )
            high = mid ;
        else if ( 0 < cond )
            low = mid + 1 ;
        else
            return a_i ;
    }
    return NULL ; // not found
}
```

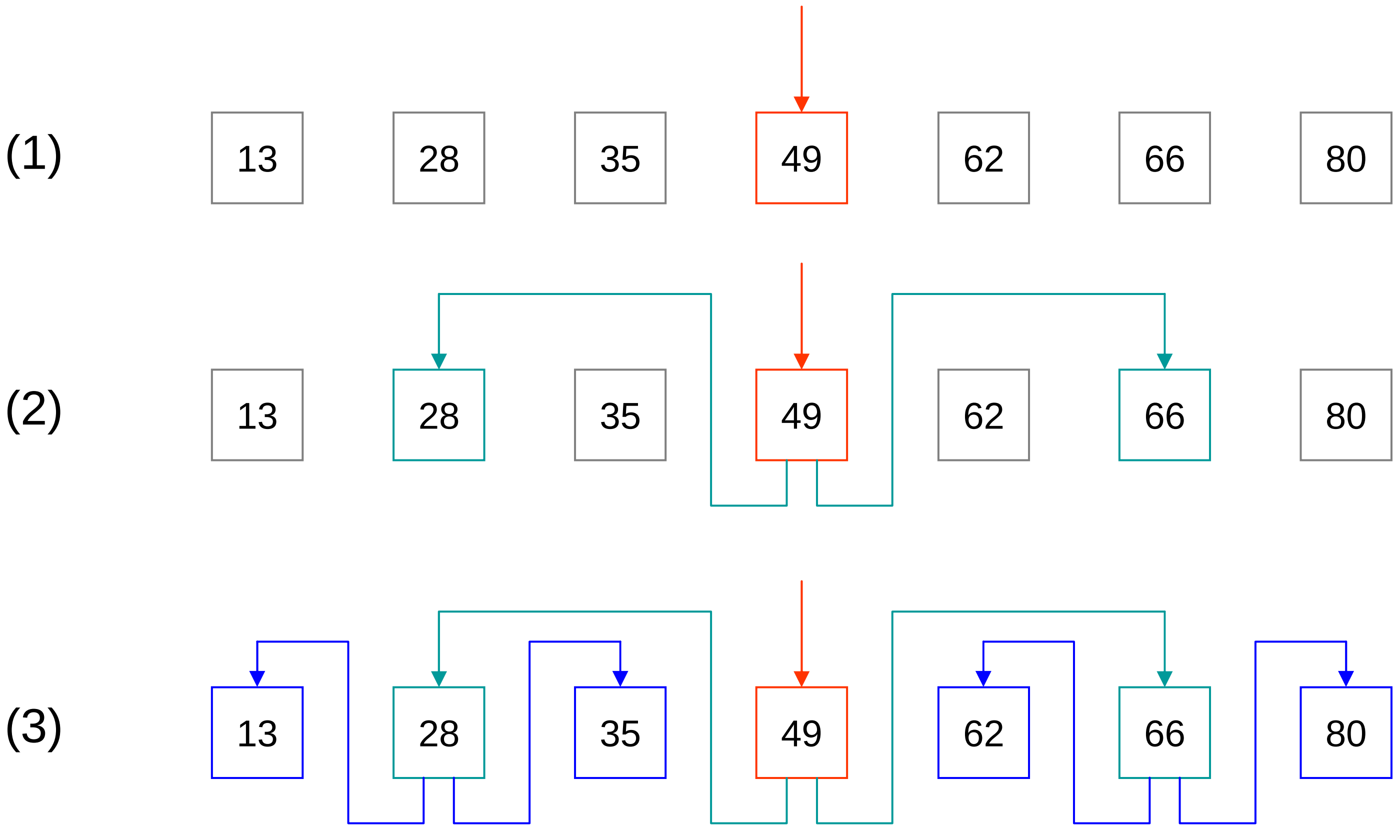


```
#include <stddef.h>

/* Given keyType array base[0], ... base[n-1]
   check if key is a keyword in array base */

void*  linear_search( const void *key, const void *base,
        size_t n, size_t size,
    int (*cmp)(const void *keyval, const void *atum)
        )
{
    size_t i ;
    char *a_i ; // &base[i]
    char *a = (char*) base ;

    for( i=0 ; i < n ; i++ ){
        a_i = a + size*i ;

        if ( 0 == (*cmp)( key, a_i ) ){
            return a_i ;
        }
    }
    return NULL ; // not found
}
```

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.

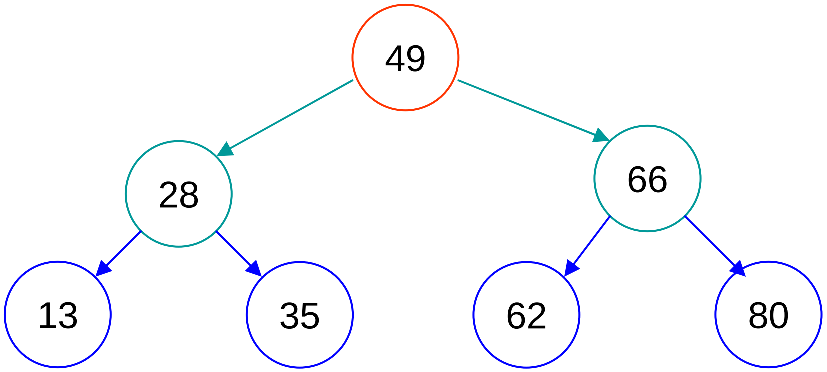
step-by-step of binary search [1]

13	28	35	49	62	66	80

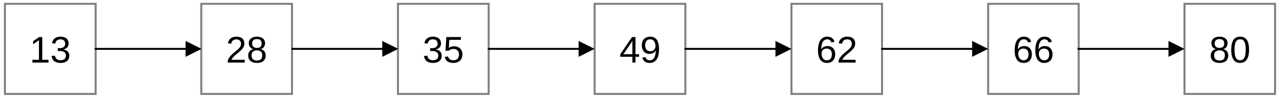


step-by-step of binary search [2]

Equivalent tree structure



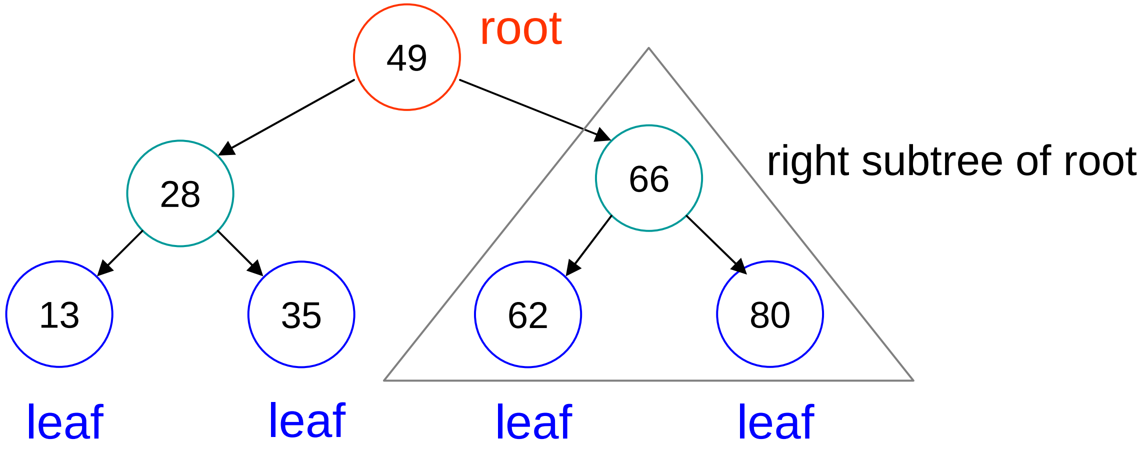
Question: Does binary-search work on sorted Linked-List?



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 [2]



root

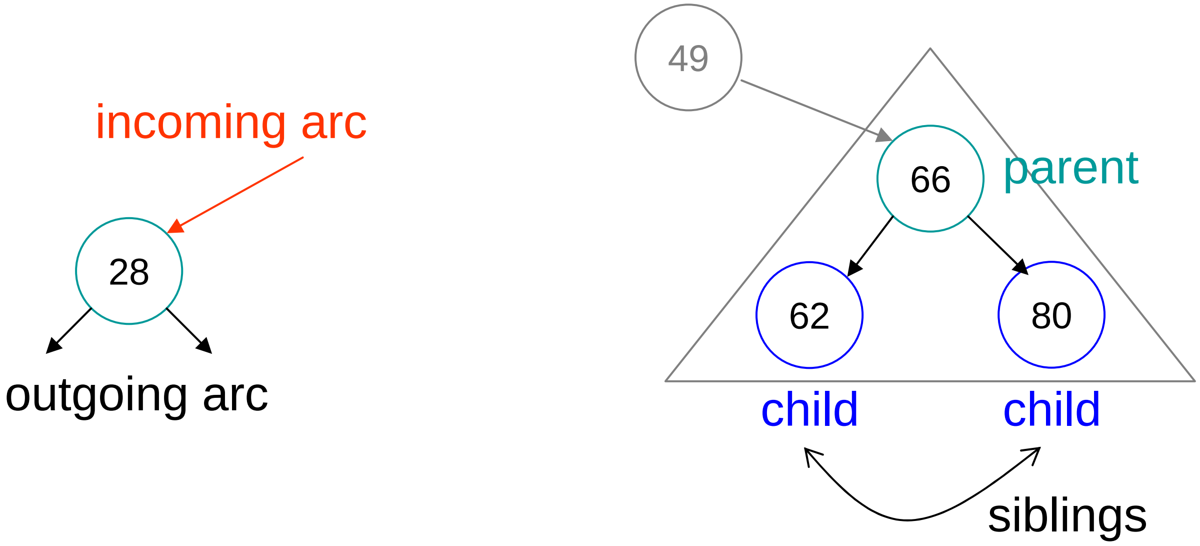
right subtree of root

leaf

leaf

leaf

leaf



incoming arc

outgoing arc

parent

child

child

siblings

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 <= value in x <= 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 key(*A*) ≥ 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

integrate into
structure ***BST***

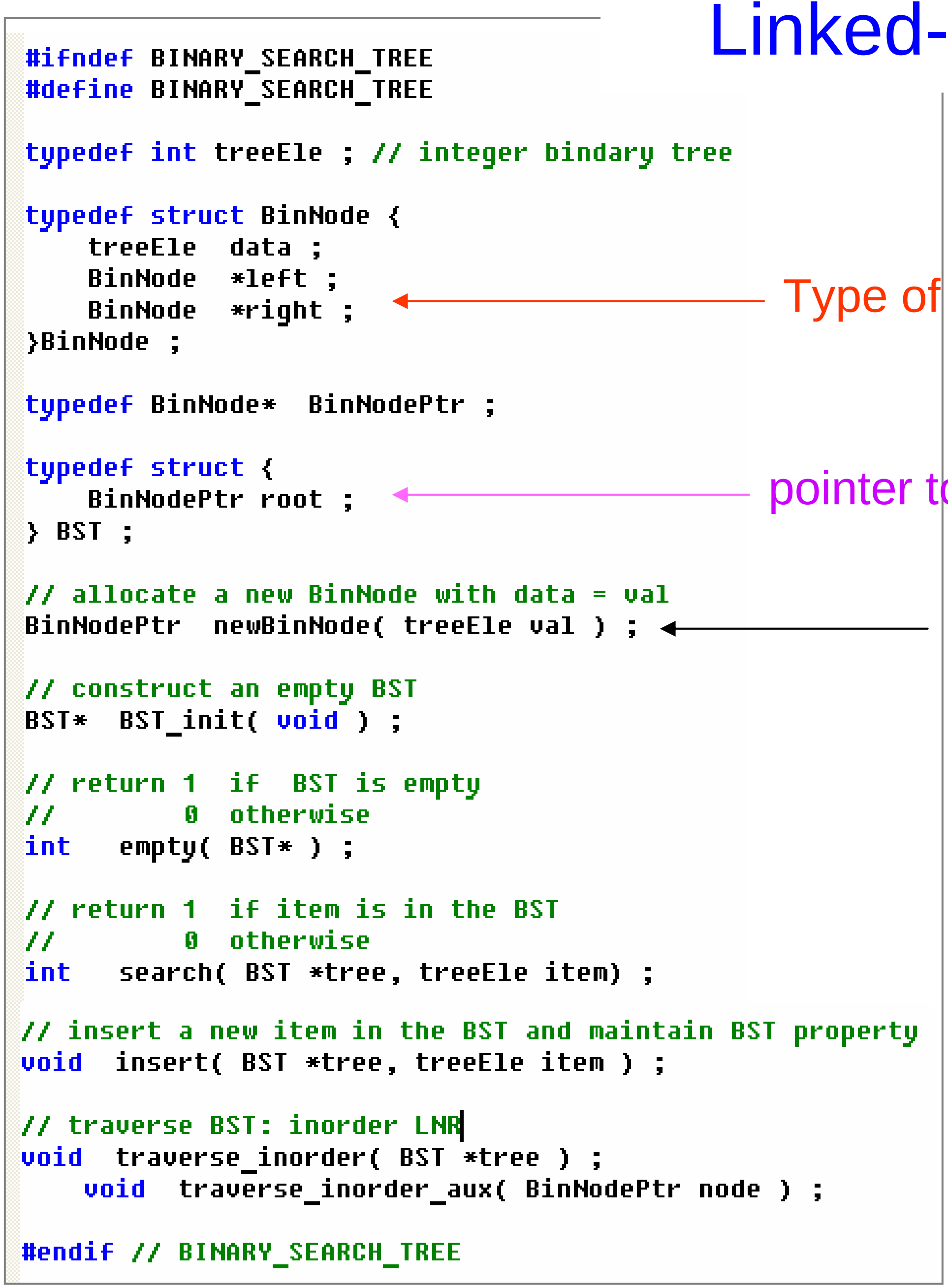
- ***treeEle***: data type
- type of physical storage: linked-list
- ordered mechanism: depends on *treeEle*
- pointer to *root* node

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**)

Linked-List BST: header file

BST.h



```
#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

typedef int treeEle ; // integer bindary 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 LNR
void  traverse_inorder( BST *tree ) ;
    void  traverse_inorder_aux( BinNodePtr node ) ;

#endif // BINARY_SEARCH_TREE
```

Type of physical storage: linked-List

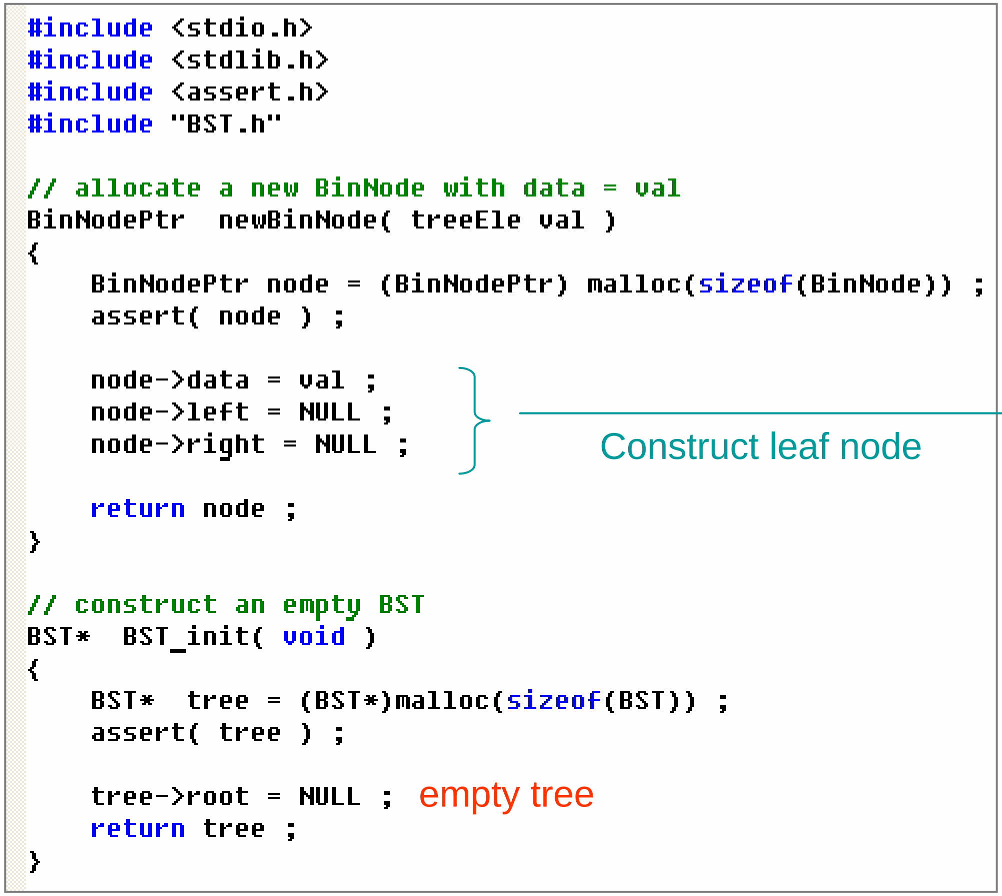
pointer to *root* node

constructor of tree node (leaf node)

Methods of structure *BST*

BST method: constructor (建構子)

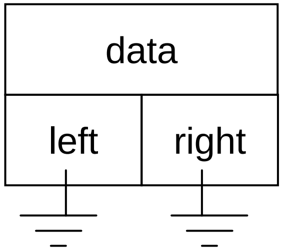
BST.cpp



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

// allocate a new BinNode with data = val
BinNodePtr  newBinNode( treeEle val )
{
    BinNodePtr node = (BinNodePtr) malloc(sizeof(BinNode)) ;
    assert( node ) ;

    node->data = 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 ;
    return tree ;
}
```

empty tree

Construct leaf node



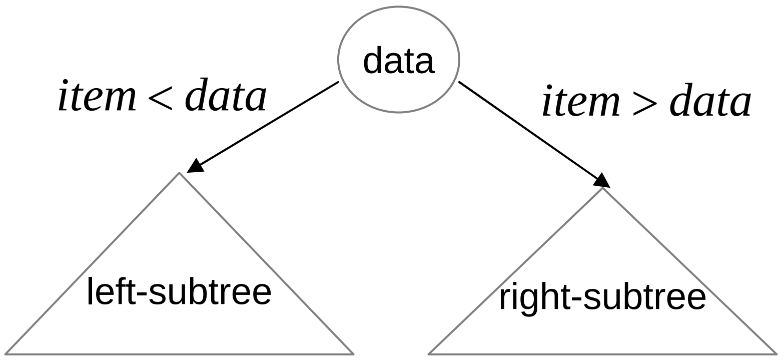
Data encapsulation: user does not ***see*** function *newBinNode*

BST method: binary search

BST.cpp

```
// 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 ;
}
```

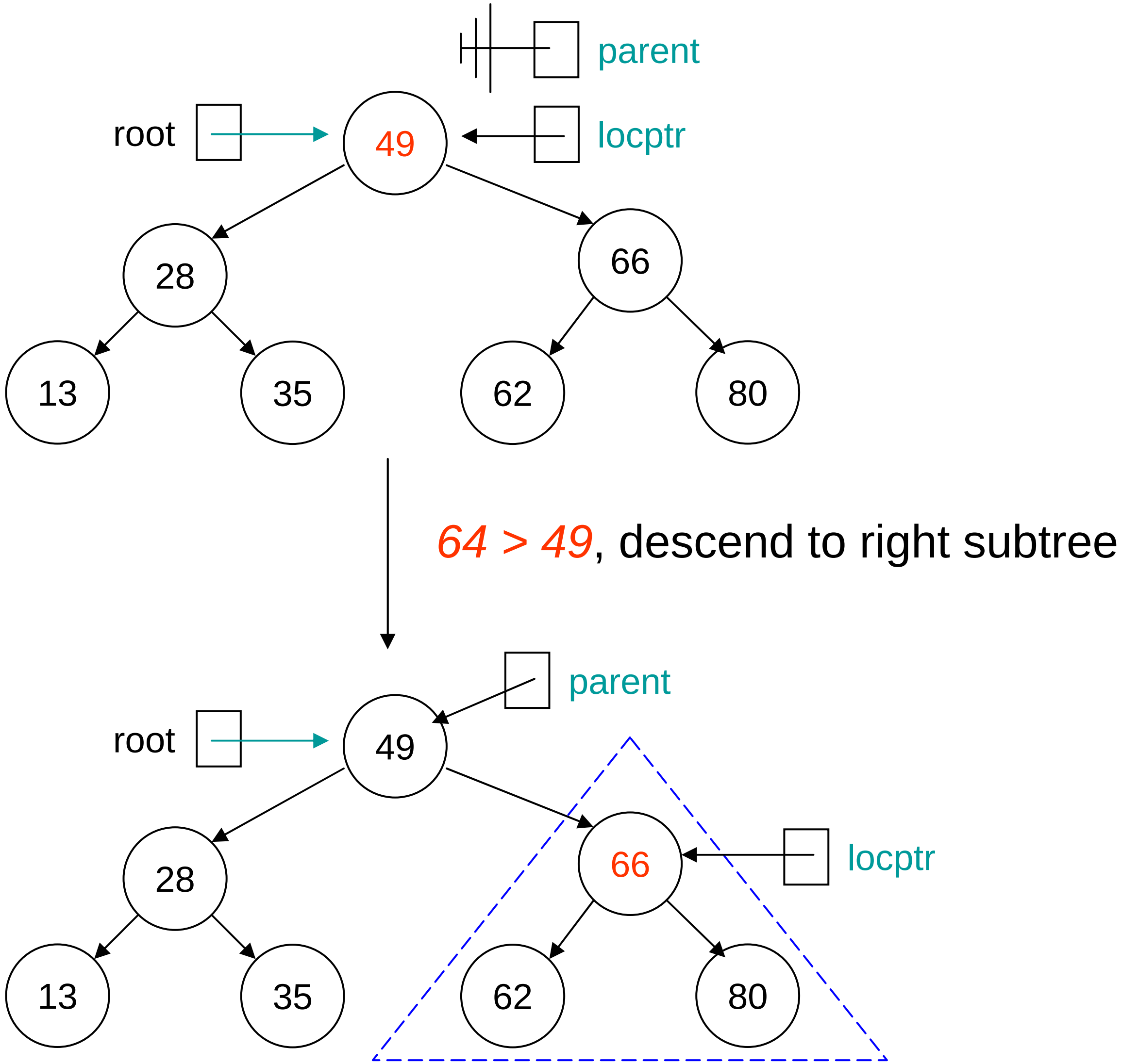
binary search



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

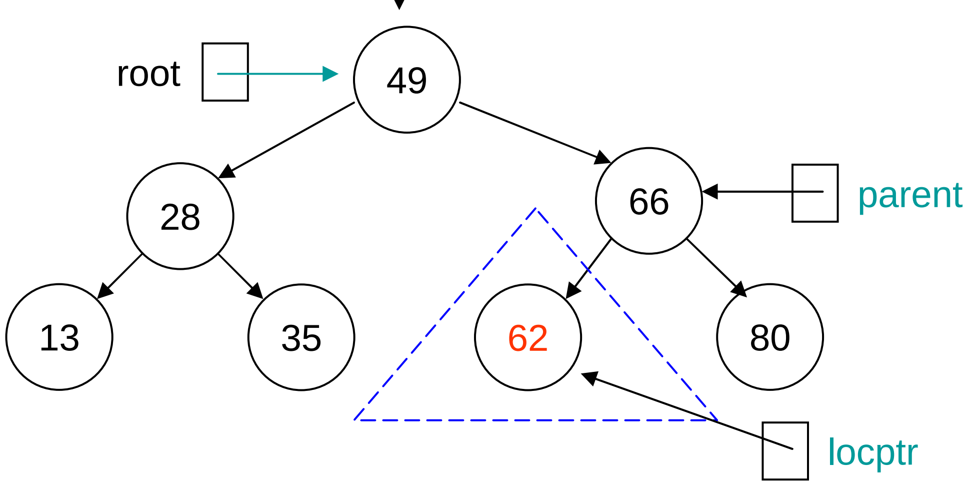
BST method: insert “64” into tree [1]



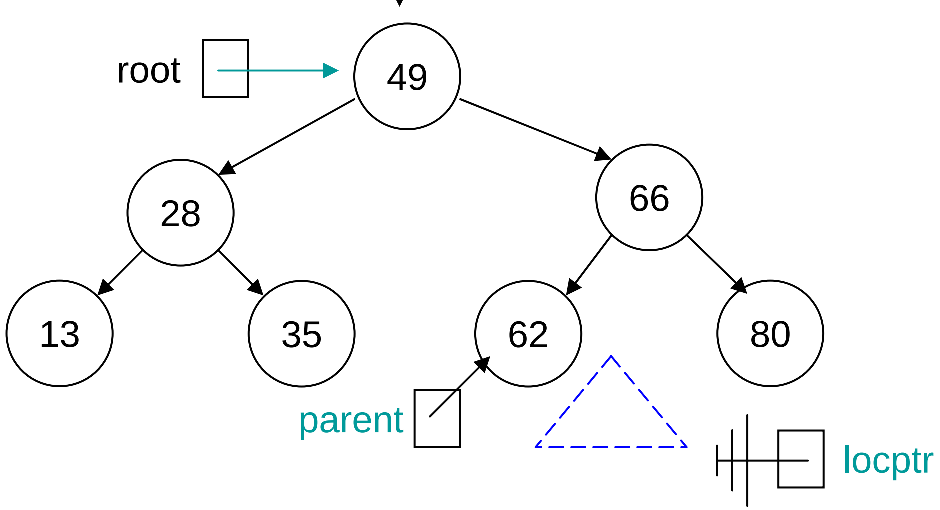
64 > 49, descend to right subtree

BST method: insert “64” into tree [2]

64 < 66, descend to left subtree

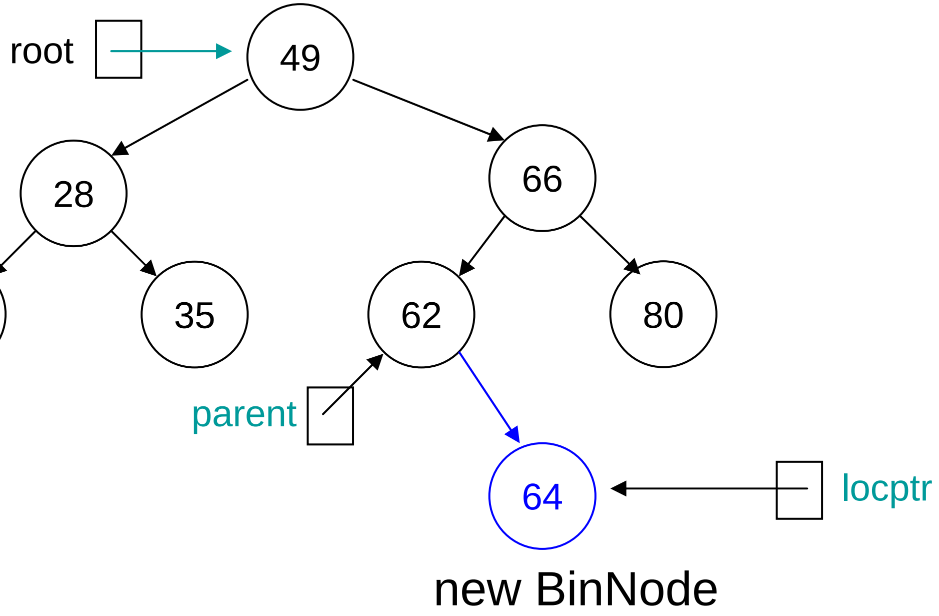


64 > 62, descend to right subtree



“64” is *NOT* in the tee

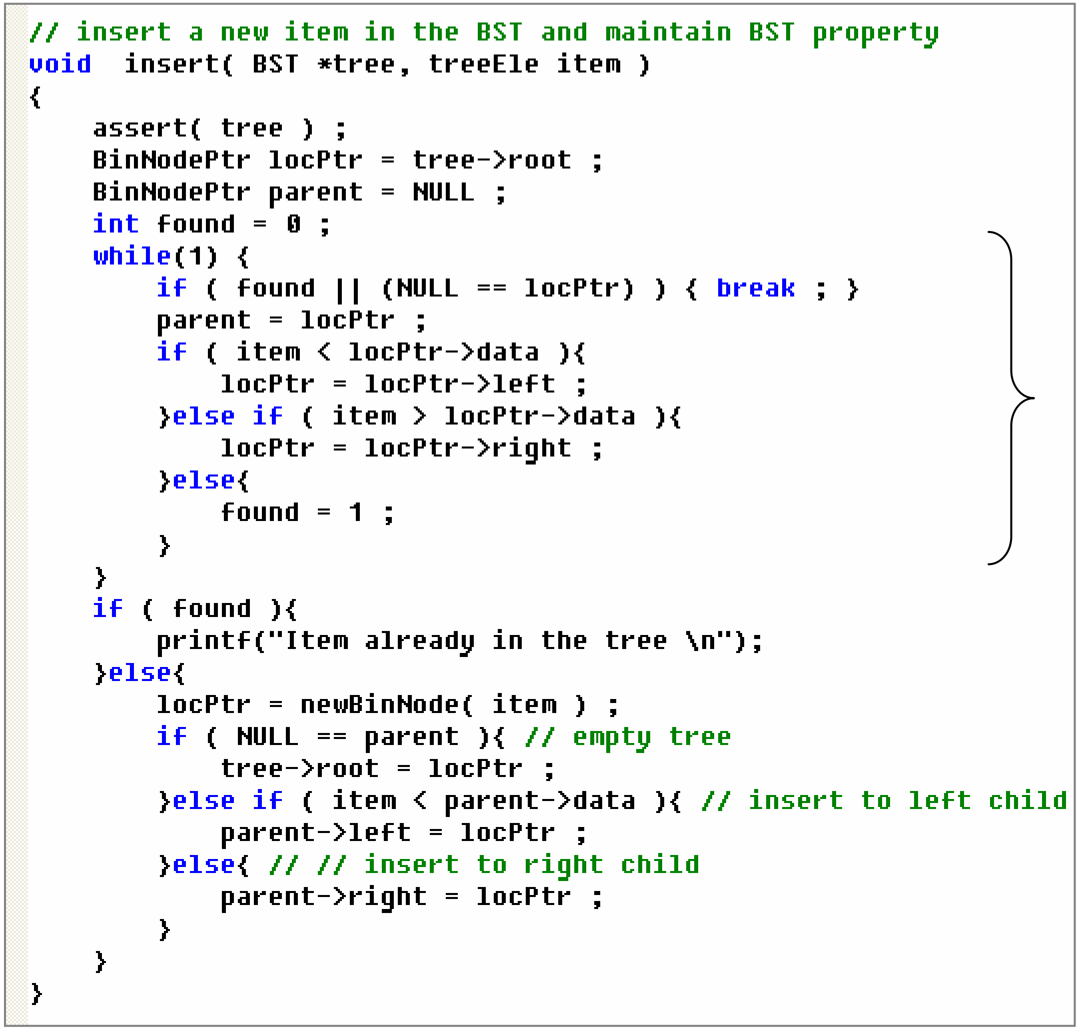
BST method: insert “64” into tree [3]



- ***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*.

BST method: insert [4]

BST.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 = newBinNode( 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 ;
        }
    }
}
```

step 1: locate parent node of target data

step 2: create leaf node of target data and attach to parent node

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

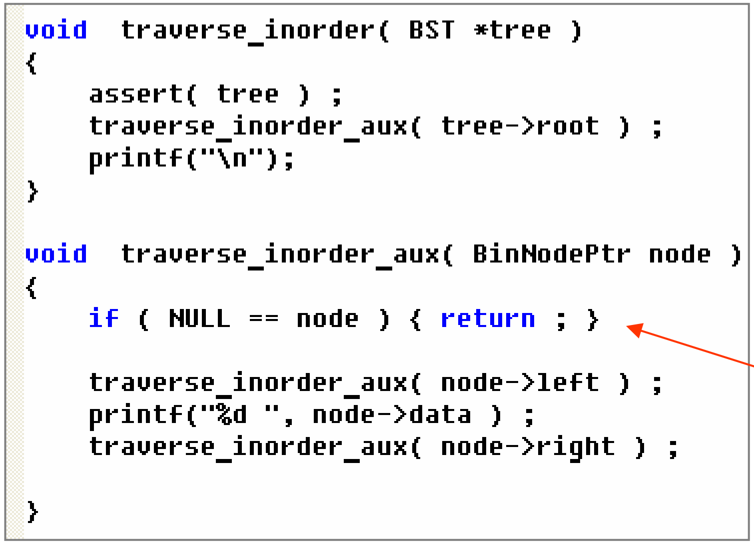
OutLine

- 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



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

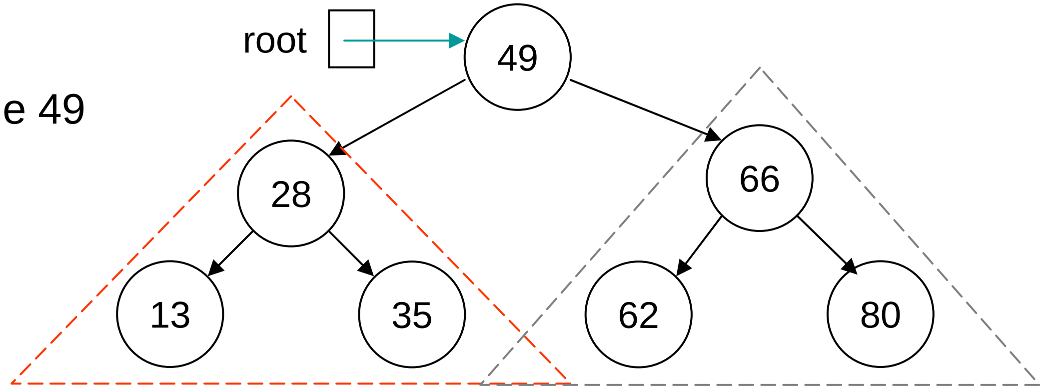
Termination condition

Inorder traversal [1]

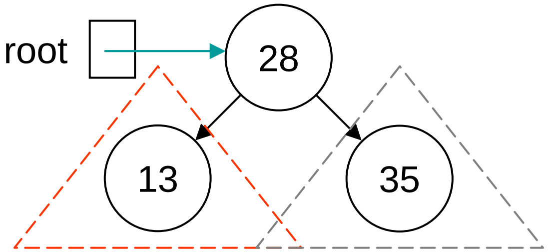
Here ***root*** means staring node of any tree

output

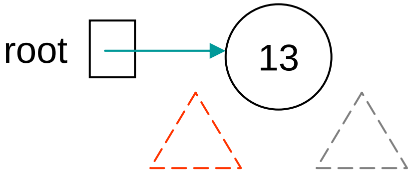
(1) goto left subtree of node 49



(2) goto left subtree of node 28



(3) goto left subtree of node 13

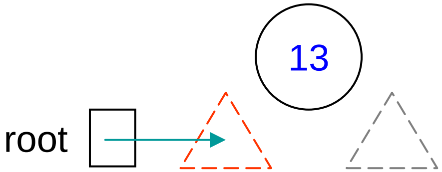


Inorder traversal [2]

output

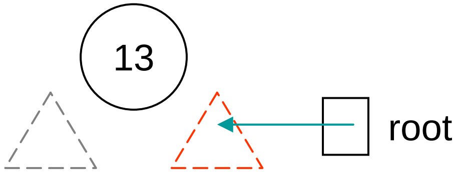
(4) root is NULL, output 13

goto right subtree of node 13



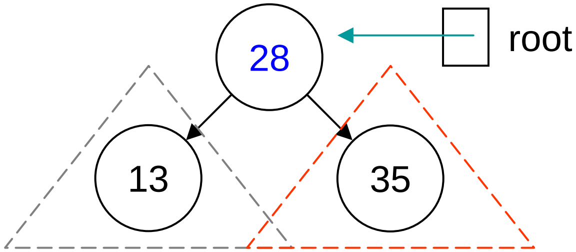
13

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



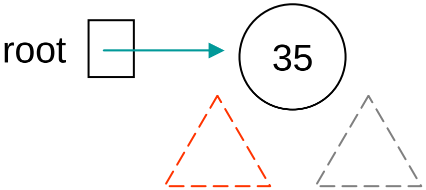
13

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



13,28

(7) goto left subtree of node 35

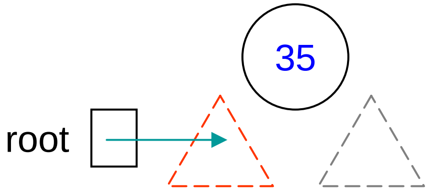


13, 28

Inorder traversal [3]

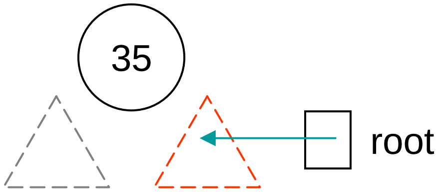
output

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



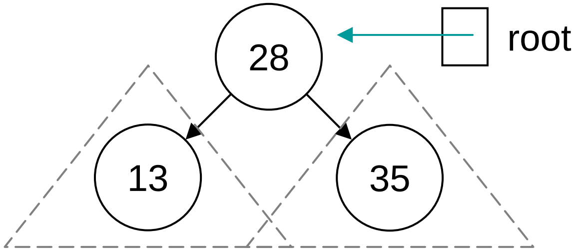
13, 28, 35

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



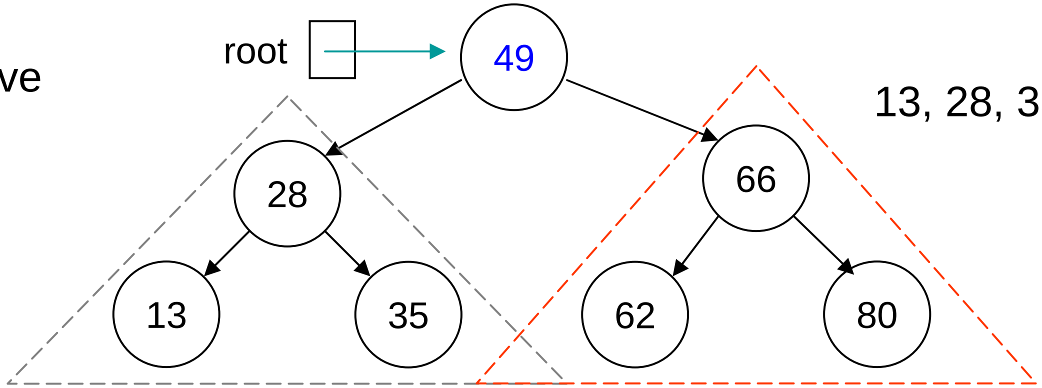
13, 28, 35

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



13, 28, 35

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

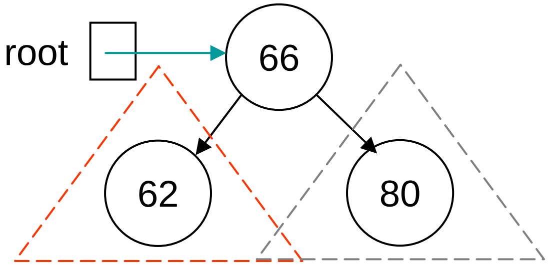


13, 28, 35, 49

Inorder traversal [4]

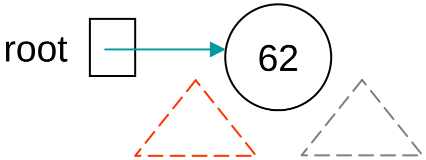
output

(12) goto left subtree of node 66



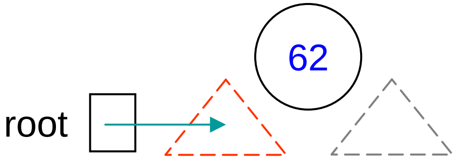
13, 28, 35, 49

(13) goto left subtree of node 62



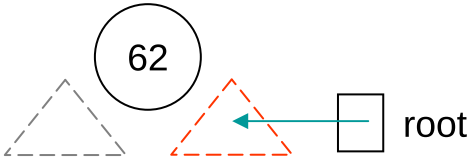
13, 28, 35, 49

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



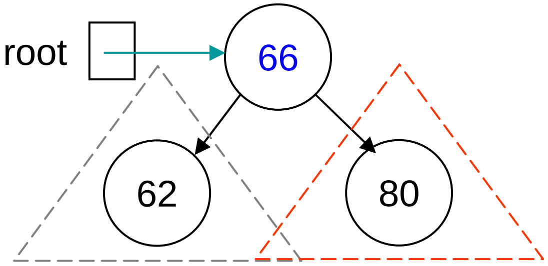
13, 28, 35, 49, 62

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



13, 28, 35, 49, 62

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

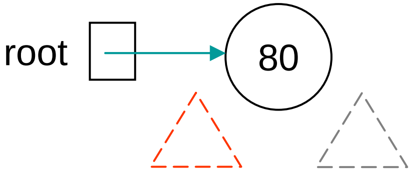


13, 28, 35, 49, 62, 66

Inorder traversal [5]

output

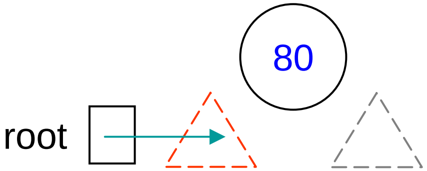
(17) goto left subtree of node 80



13,28,35,49,62,66

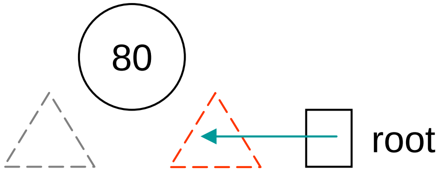
(18) root is NULL, output 80 and

goto right subtree of node 80



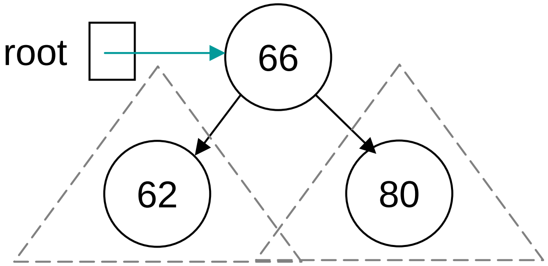
13,28,35,49,62,66,80

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



13,28,35,49,62,66,80

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

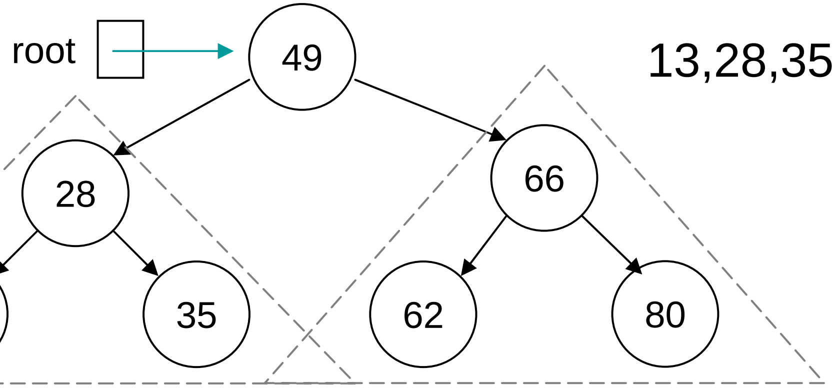


13,28,35,49,62,66,80

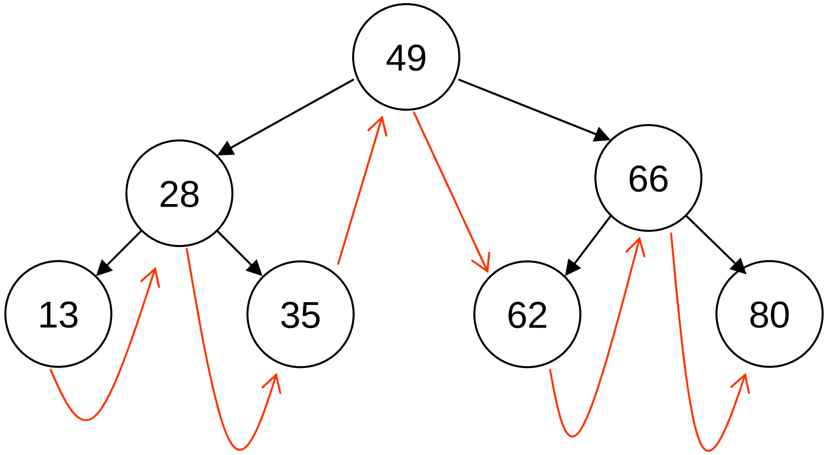
Inorder traversal [6]

output

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



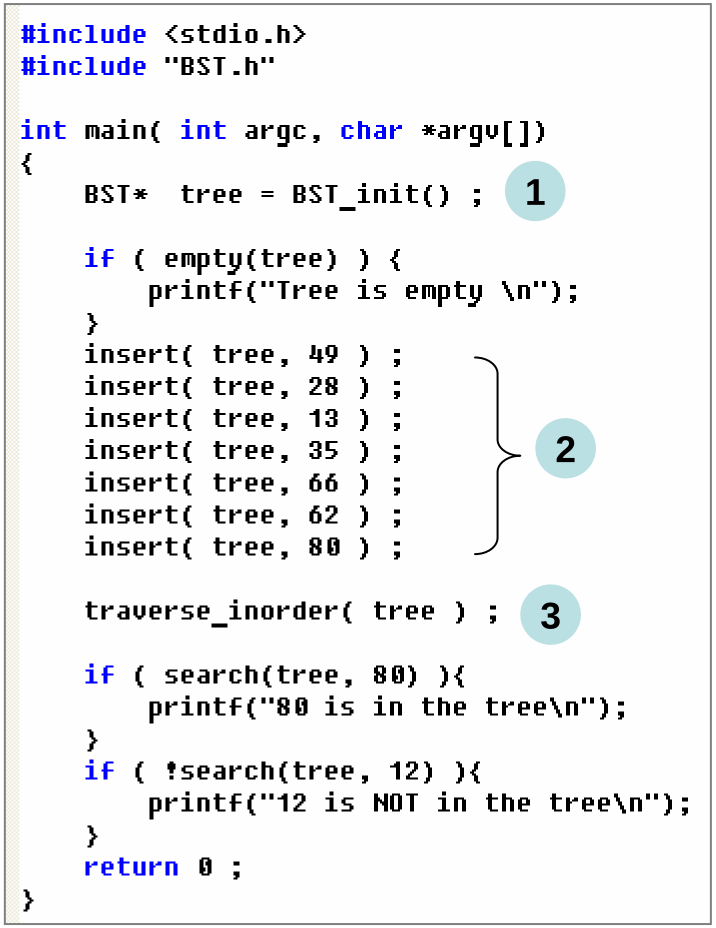
13,28,35,49,62,66,80



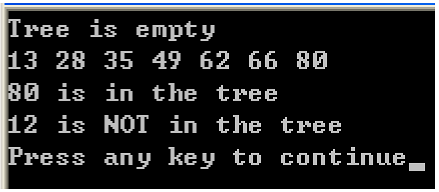
Inorder in BST is ascending order, why?

Driver for Inorder traversal [1]

main.cpp

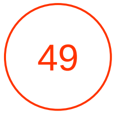


```
#include <stdio.h>
#include "BST.h"

int main( int argc, char *argv[])
{
    BST*  tree = BST_init() ;

    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 ) ;

    if ( search(tree, 80) ){
        printf("80 is in the tree\n");
    }
    if ( !search(tree, 12) ){
        printf("12 is NOT in the tree\n");
    }
    return 0 ;
}
```

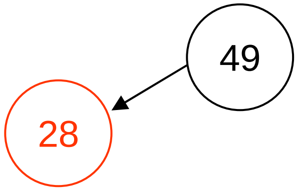


```
Tree is empty
13 28 35 49 62 66 80
80 is in the tree
12 is NOT in the tree
Press any key to continue_
```

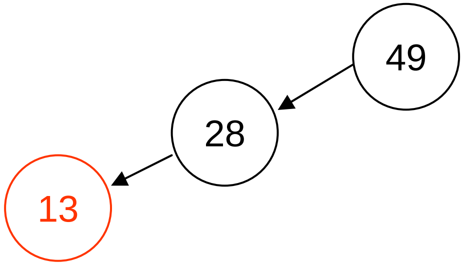
insert(tree,49)



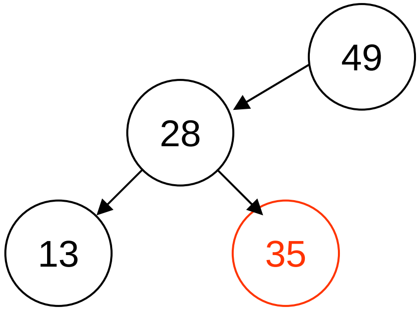
insert(tree,28)



insert(tree,13)

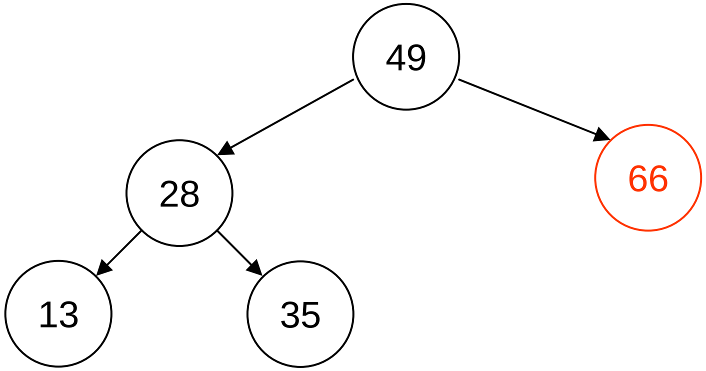


insert(tree,35)

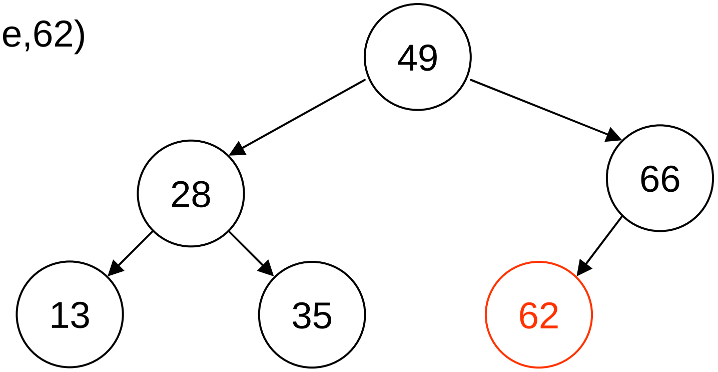


Driver for Inorder traversal [2]

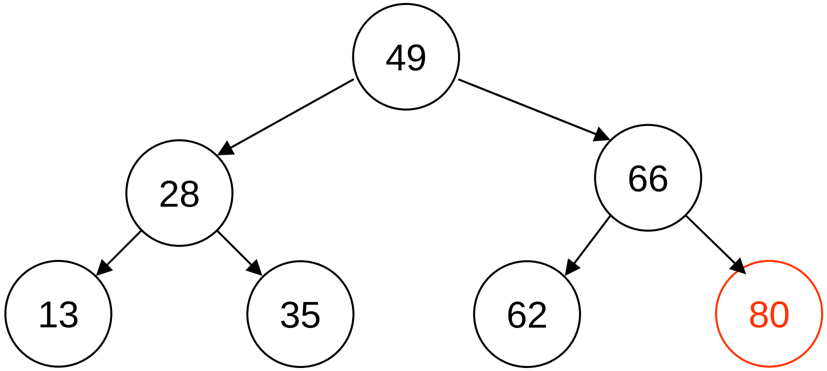
insert(tree,66)



insert(tree,62)

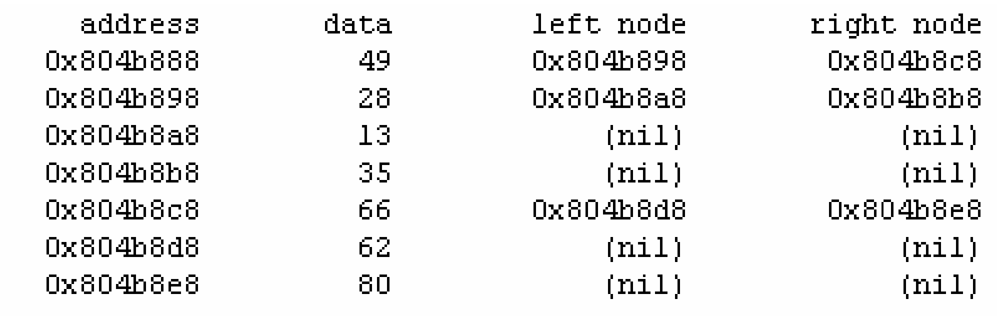


insert(tree,80)

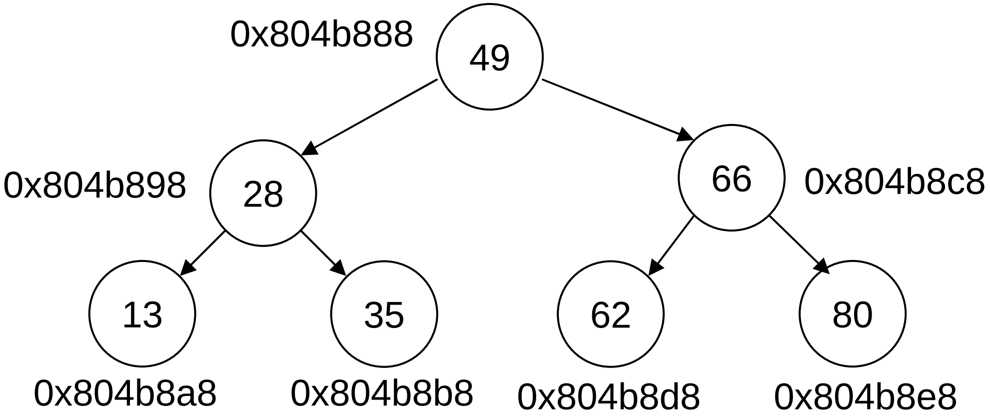


Exercise

- 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.

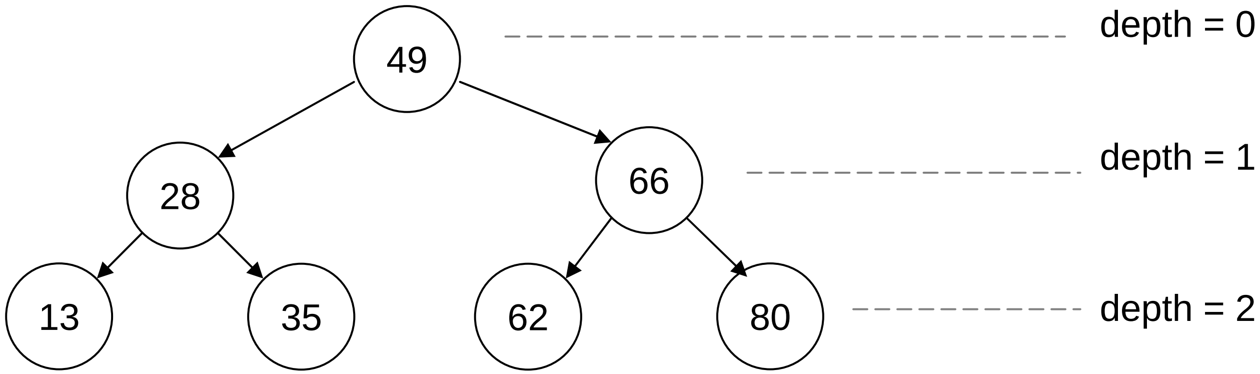


```
   address          data          left node          right node
0x804b888            49          0x804b898           0x804b8c8
0x804b898            28          0x804b8a8           0x804b8b8
0x804b8a8            13              (nil)               (nil)
0x804b8b8            35              (nil)               (nil)
0x804b8c8            66          0x804b8d8           0x804b8e8
0x804b8d8            62              (nil)               (nil)
0x804b8e8            80              (nil)               (nil)
```



Exercise

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



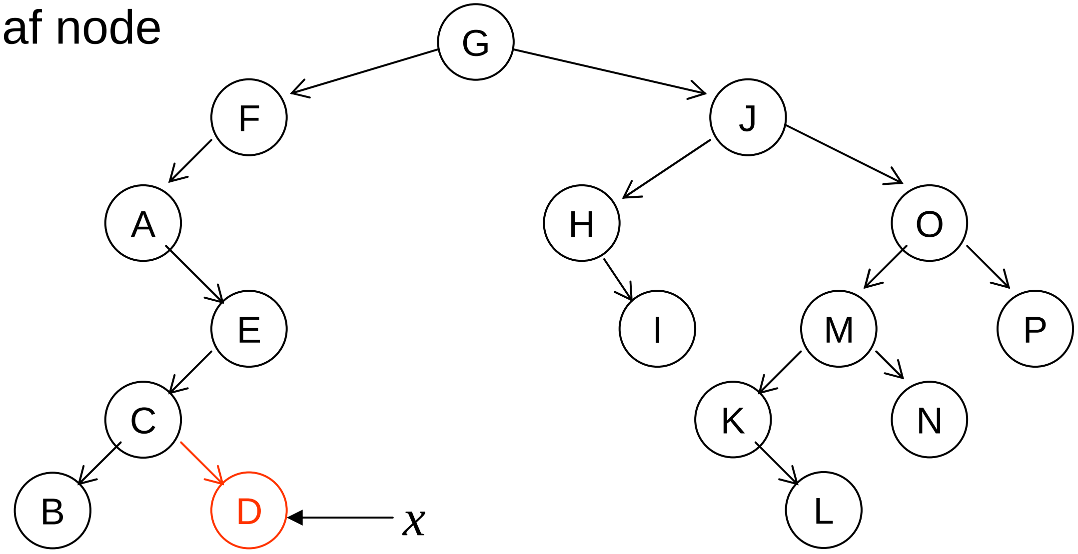
- What is topology of a BST created by inserting 13, 28, 35, 49, 62, 66, 80 in turn.
- Can you modify an unbalanced BST into a balanced one?

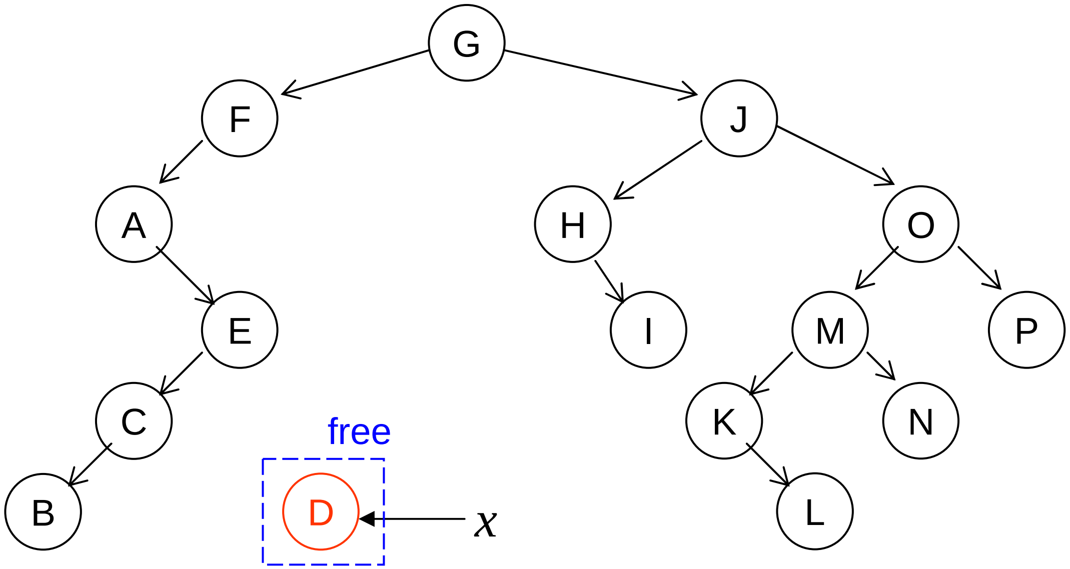
OutLine

- 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]

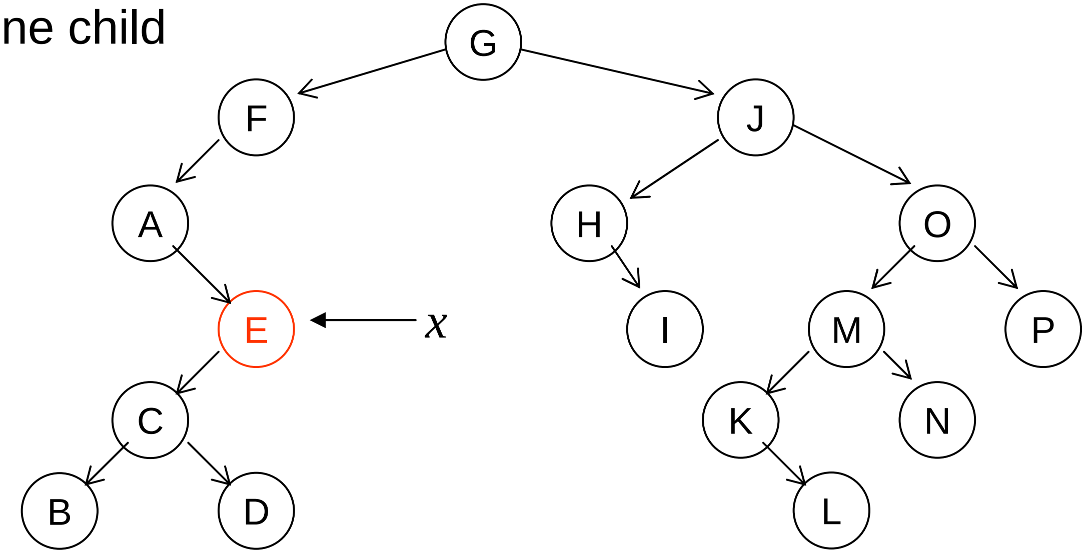
case 1: x is a leaf node

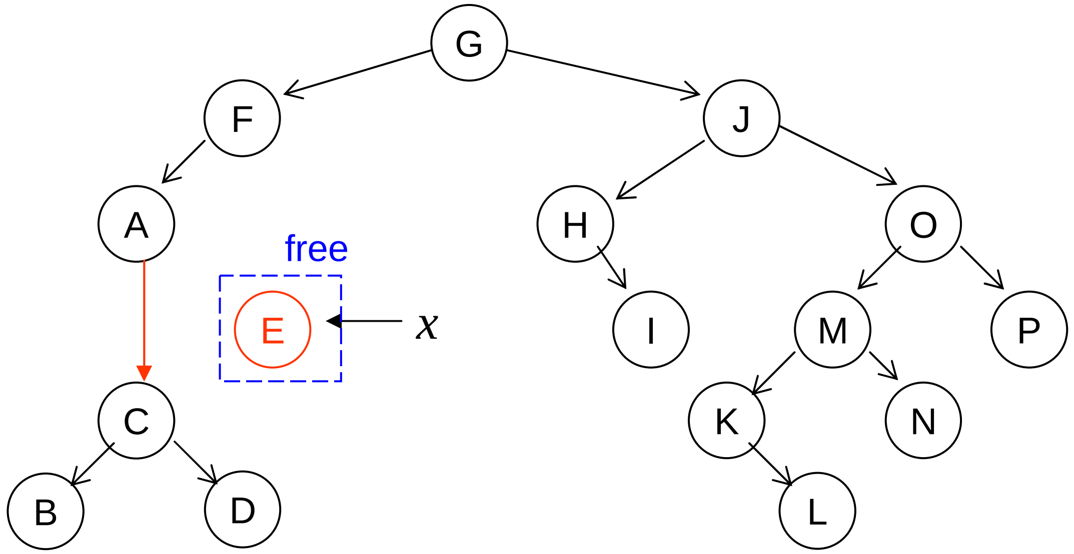




Delete a node x from BST [2]

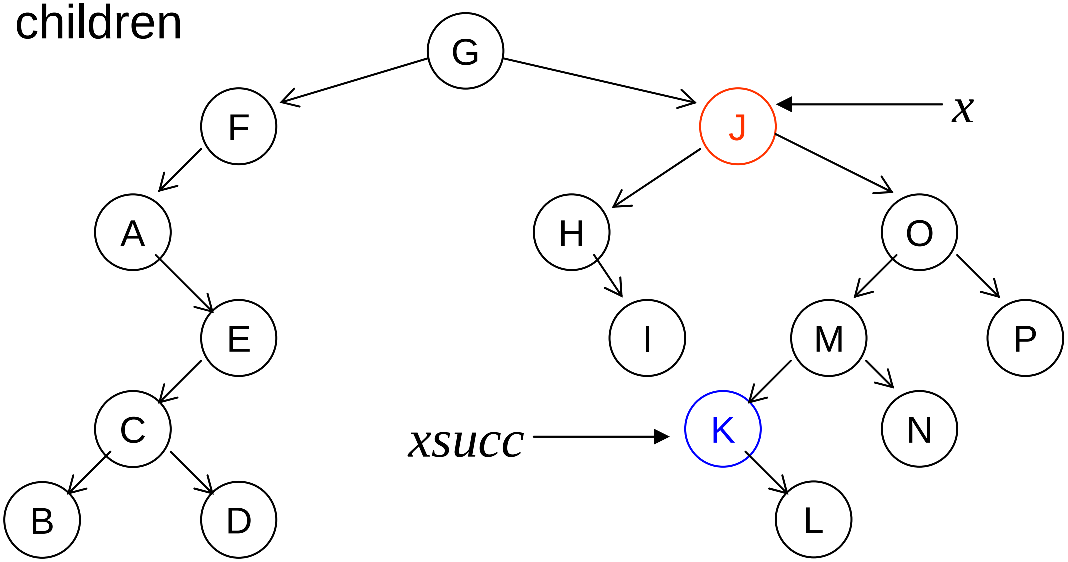
case 2: x has one child



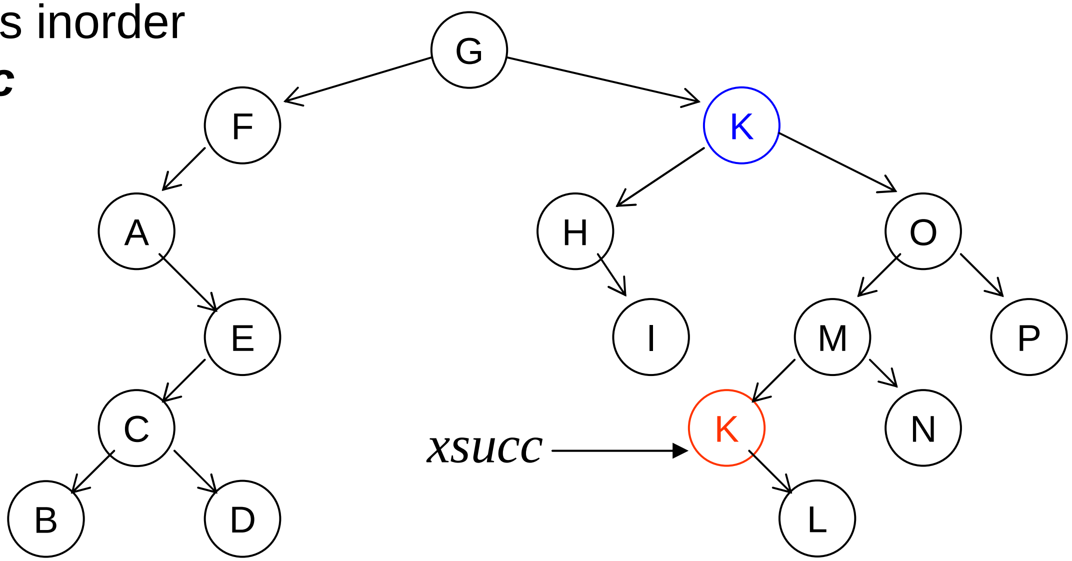


Delete a node x from BST [3]

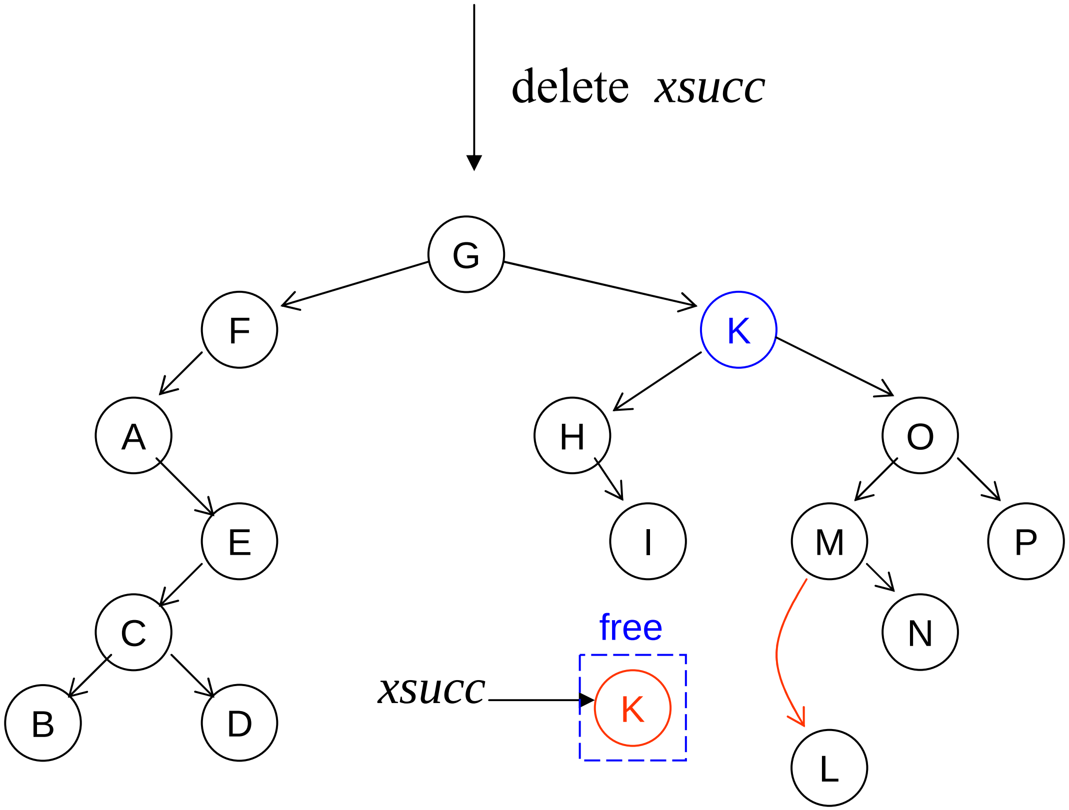
case 3: x has two children



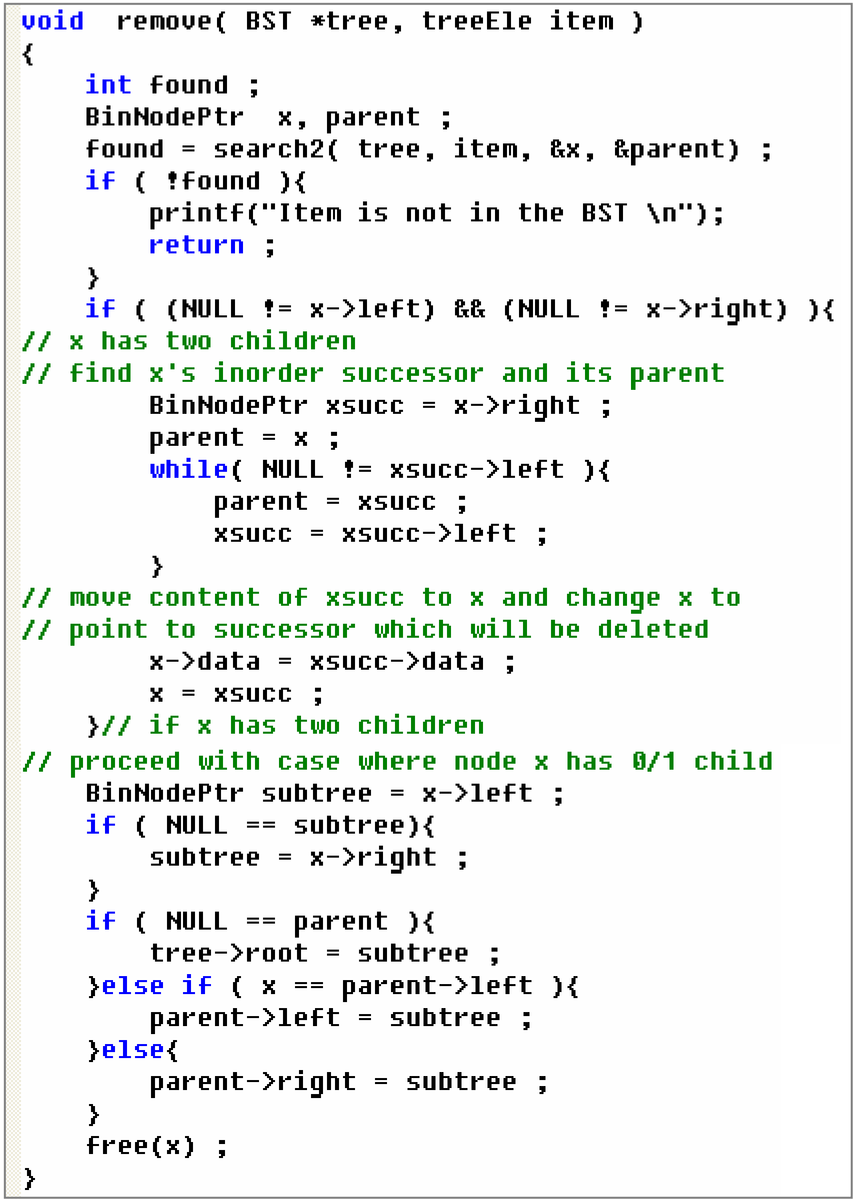
Replace x with its inorder successor ***xsucc***



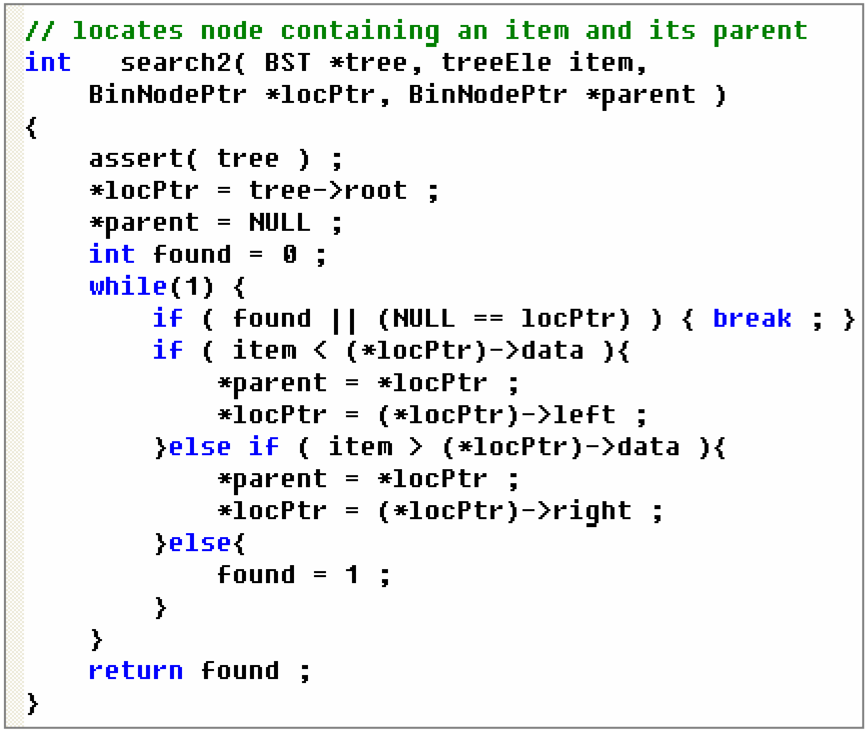
Delete a node x from BST [4]



BST method: remove item



```
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) ;
}
```

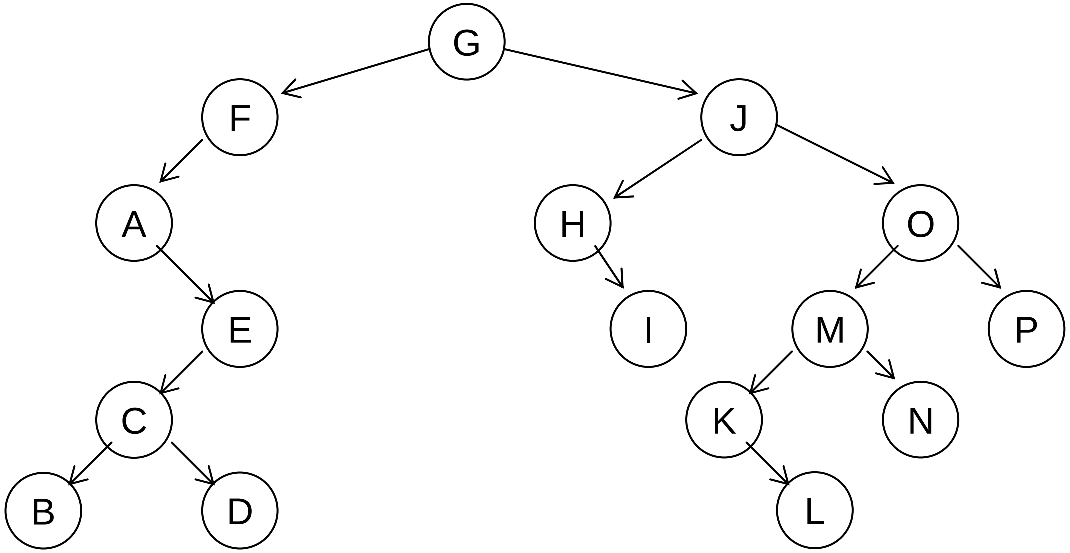


```
// locates node containing an item and its parent
int   search2( BST *tree, treeEle item,
    BinNodePtr *locPtr, BinNodePtr *parent )
{
    assert( tree ) ;
    *locPtr = tree->root ;
    *parent = NULL ;
    int found = 0 ;
    while(1) {
        if ( found || (NULL == locPtr) ) { break ; }
        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 *remove* and write a driver to test it, you can use following BST as test example.
  Note: you need to test all boundary cases

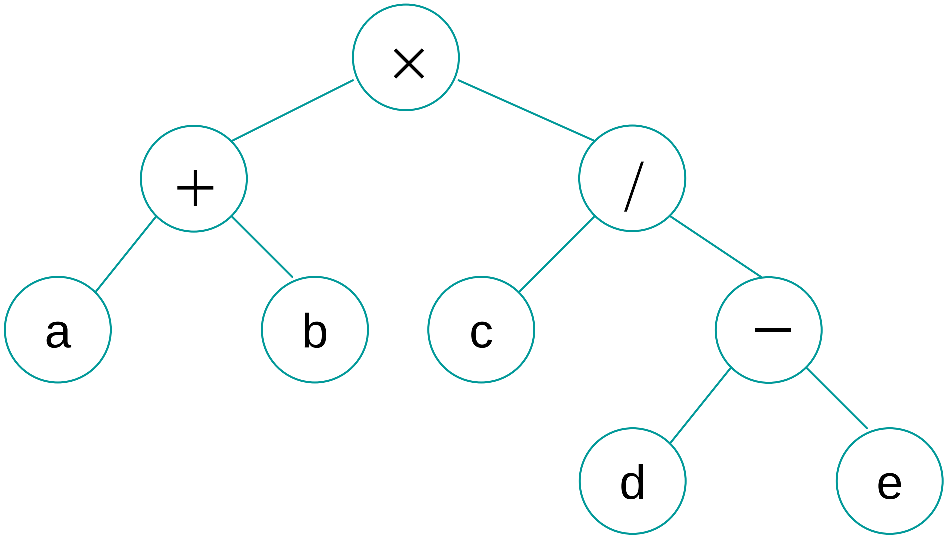
- Use recursive call to implement methods *remove.*



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.

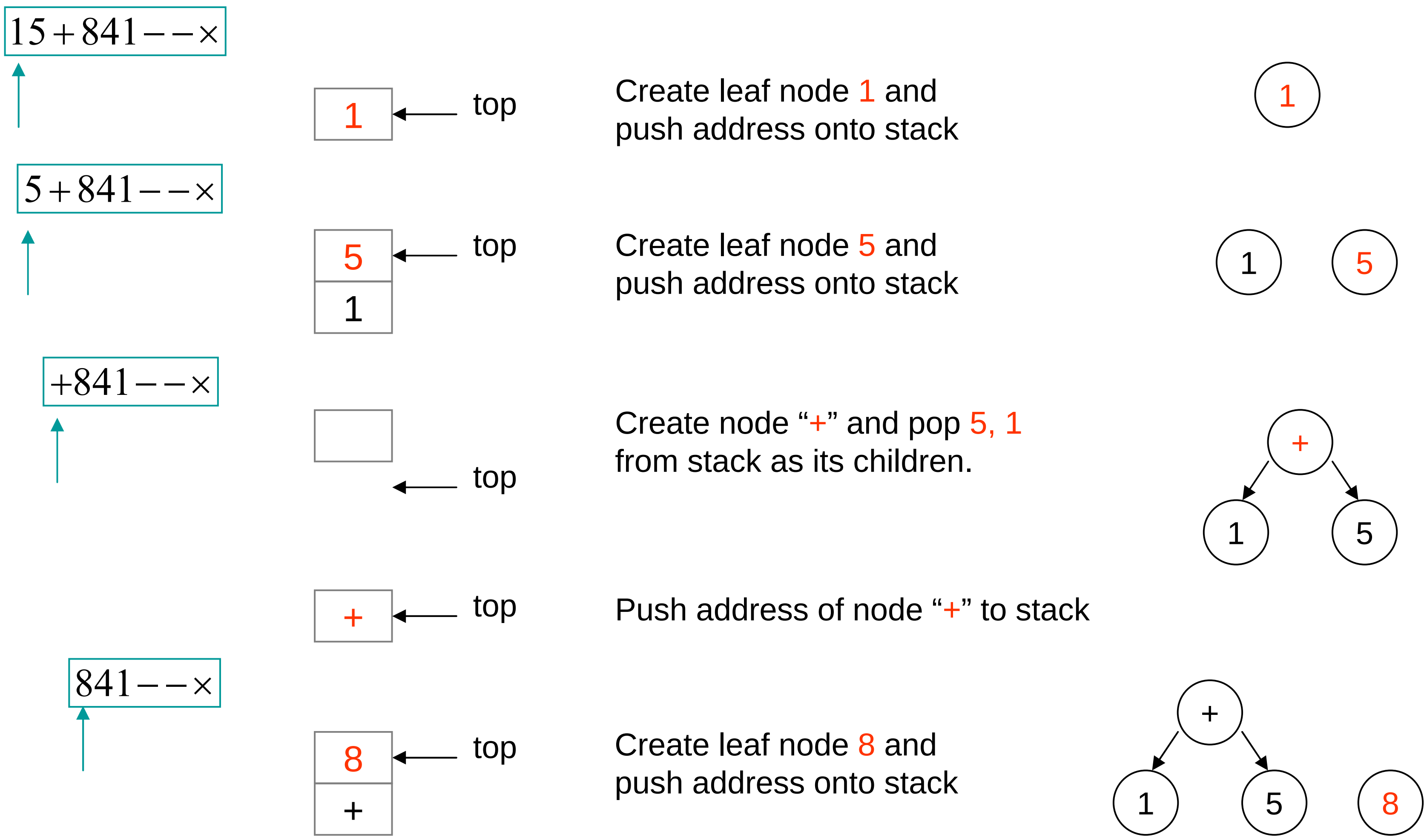
$$(a+b)\times(c/(d-e))$$



OutLine

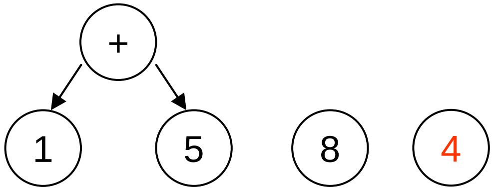
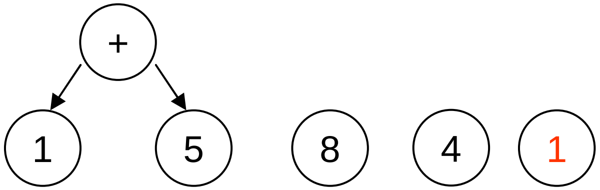
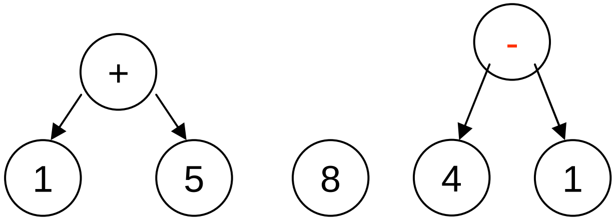
- Binary search versus tree structure
- Binary search tree and its implementation
  - insertion
  - traversal
  - delete
- Application: expression tree
  - convert RPN to binary tree
  - evaluate expression tree
- Pitfall: stack limit of recursive call

Convert RPN expression to expression tree [1]



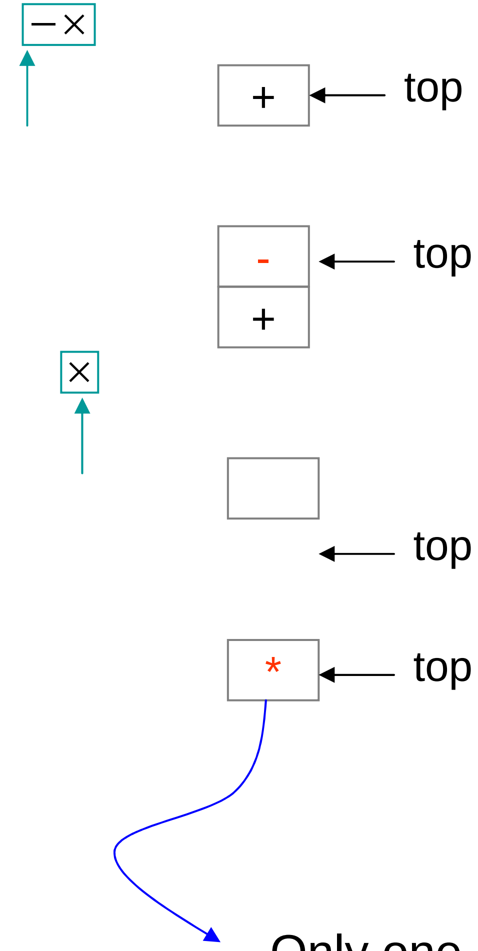
expression	stack	comments	Binary tree
$15+841--\times$	1 ← top	Create leaf node 1 and push address onto stack	1
$5+841--\times$	5 ← top 1	Create leaf node 5 and push address onto stack	1 5
$+841--\times$	← top	Create node “+” and pop 5, 1 from stack as its children.	+ 1 5
	+ ← top	Push address of node “+” to stack	
$841--\times$	8 ← top +	Create leaf node 8 and push address onto stack	+ 1 5 8

Convert RPN expression to expression tree [2]

expression	stack	comments	Binary tree
41−−×	4 ← top 8 +	Create leaf node 4 and push address onto stack	
1−−×	1 ← top 4 8 +	Create leaf node 1 and push address onto stack	
−−×	8 ← top +	Create node "-" and pop 1, 4 from stack as its children.	
	- ← top 8 +	Push node '-' onto stack	

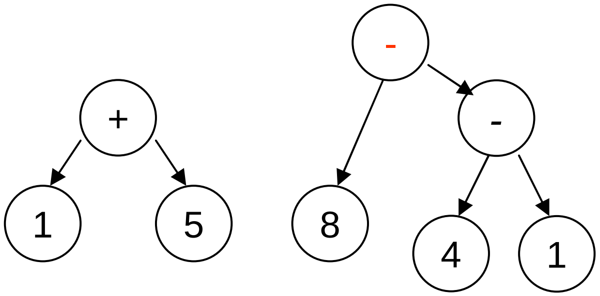
Convert RPN expression to expression tree [3]

expression stack comments Binary tree



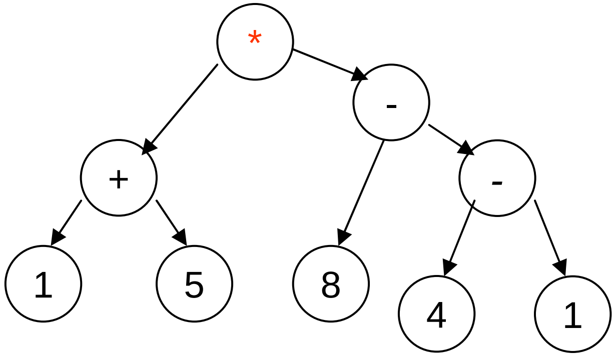
Create node "-" and pop "-", 8 from stack as its children.

Push node "-" onto stack



Create node "*" and pop "-", "+" from stack as its children.

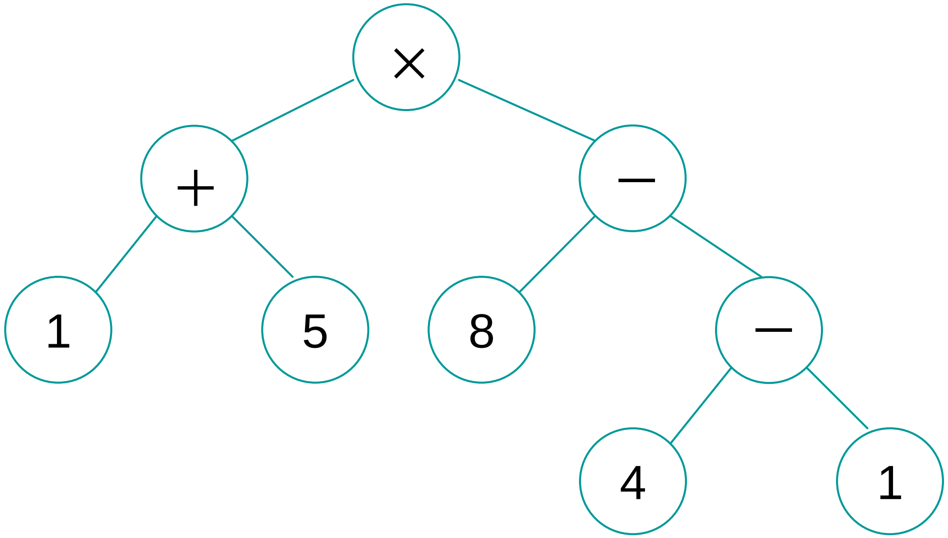
Push node "*" onto stack



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).



$$\text{infix: } (1+5)\times\left(8-(4-1)\right)$$

$$postfix: \ 15+841--\times$$

parenthesis free

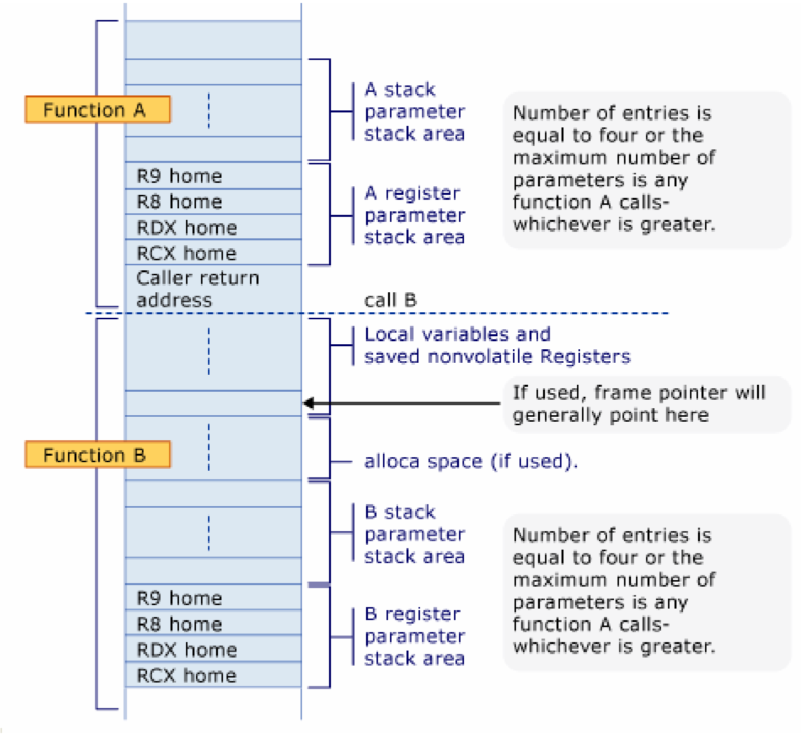
OutLine

- 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

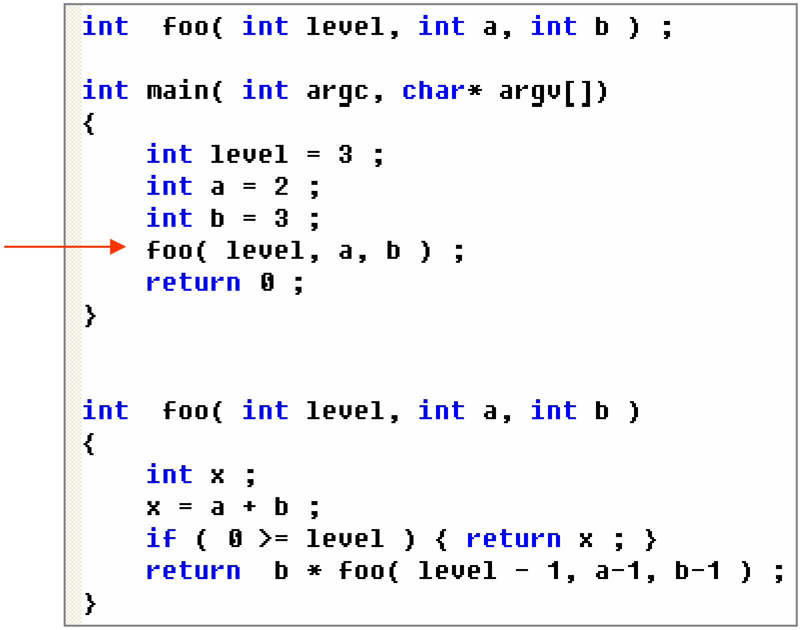
- A function‘s prolog (prolog code sequence 起始設定) is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.
- The parameter area is always at the bottom of the stack, so that it will always be adjacent to the return address during any function 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 variabels are automatically deleted when they go out of scope. Sometimes, we call them **automatic variables**.



Stack frame by g++

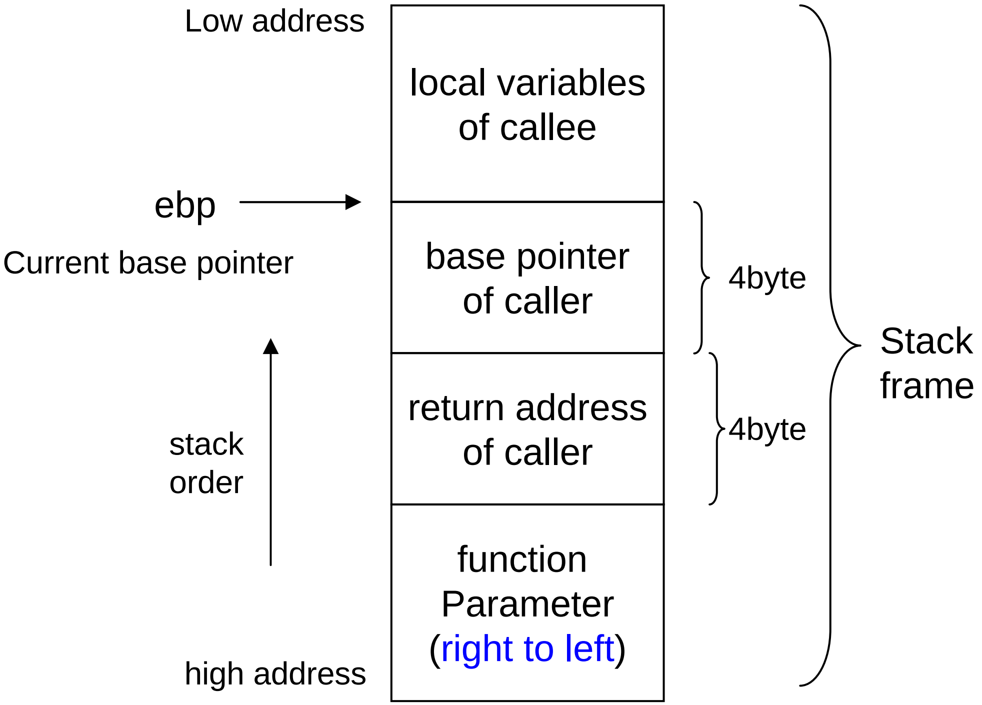
g++ -O0 main.cpp



```
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 ( 0 >= 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

ebp

Current base pointer

stack order

high address

local variables of callee

base pointer of caller

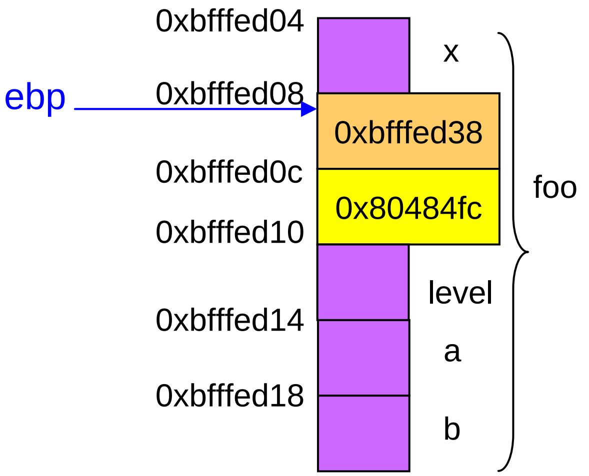
return address of caller

function Parameter (right to left)

4byte

4byte

Stack frame



0xbfffed04

ebp

0xbfffed08

0xbfffed0c

0xbfffed10

0xbfffed14

0xbfffed18

x

0xbfffed38

0x80484fc

level

a

b

foo

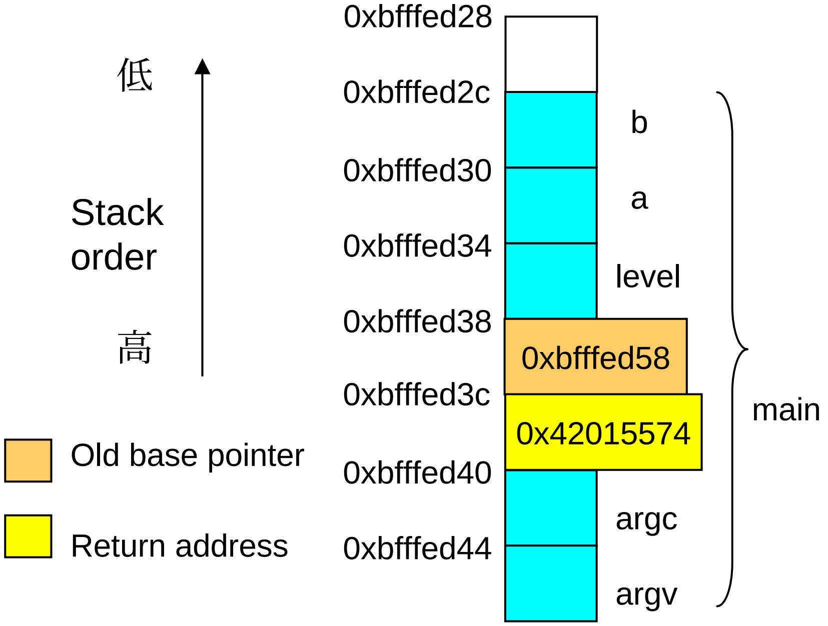
x = ebp[-4]

level = ebp[8]

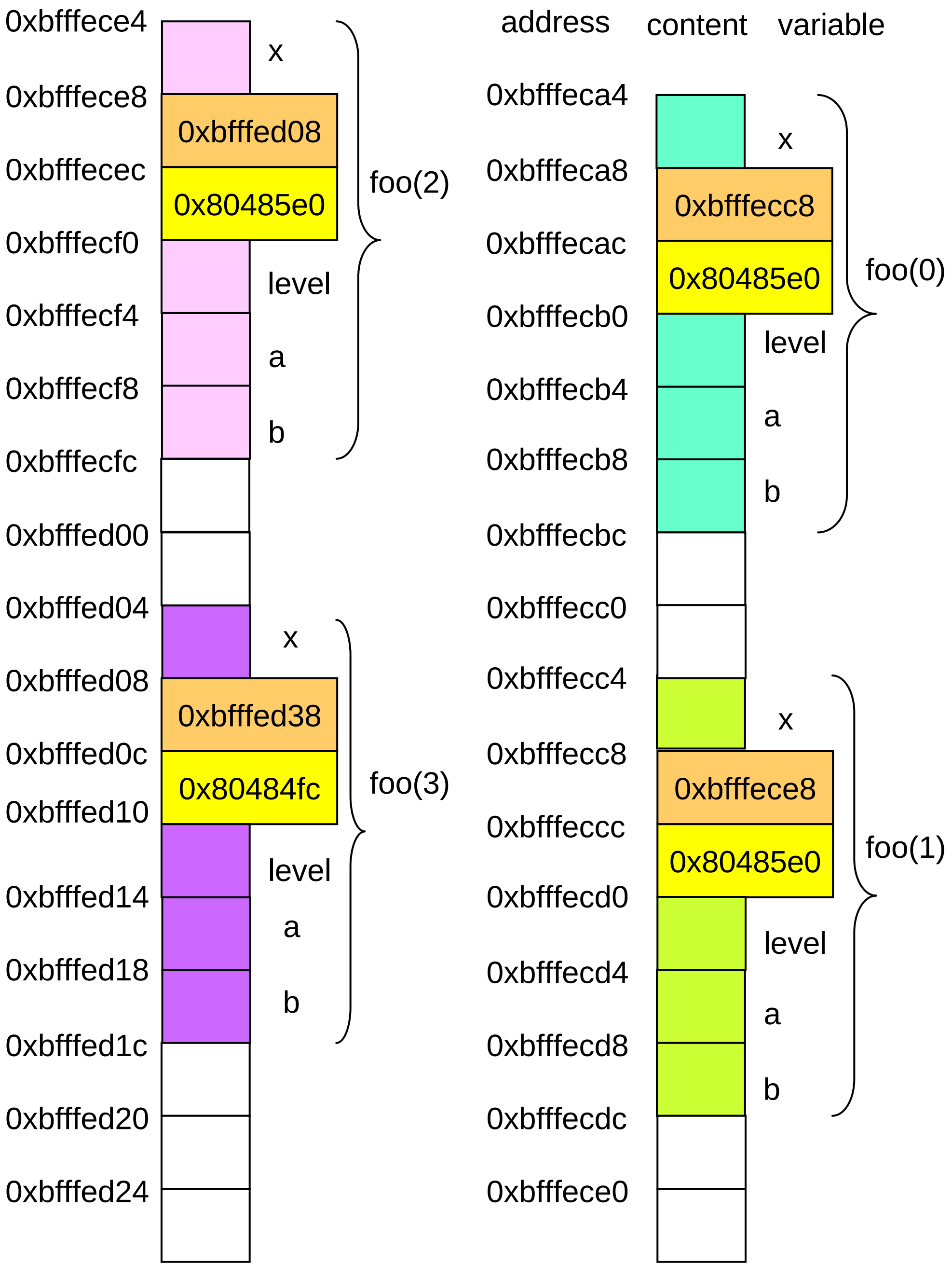
a = ebp[12]

b = ebp[16]

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



address	content	variable
0xbfffed28		
0xbfffed2c		b
0xbfffed30		a
0xbfffed34		level
0xbfffed38	0xbfffed58	
0xbfffed3c	0x42015574	
0xbfffed40		argc
0xbfffed44		argv



address	content	variable
0xbfffece4		x
0xbfffece8	0xbfffed08	
0xbfffecec	0x80485e0	
0xbfffecf0		level
0xbfffecf4		a
0xbfffecf8		b
0xbfffecfc		
0xbfffed00		
0xbfffed04		x
0xbfffed08	0xbfffed38	
0xbfffed0c	0x80484fc	
0xbfffed10		level
0xbfffed14		a
0xbfffed18		b
0xbfffed1c		
0xbfffed20		
0xbfffed24		

address	content	variable
0xbfffeca4		x
0xbfffeca8	0xbfffecc8	
0xbfffecac	0x80485e0	
0xbfffecb0		level
0xbfffecb4		a
0xbfffecb8		b
0xbfffecbc		
0xbfffecc0		
0xbfffecc4		x
0xbfffecc8	0xbfffece8	
0xbfffeccc	0x80485e0	
0xbfffecd0		level
0xbfffecd4		a
0xbfffecd8		b
0xbfffecdc		
0xbfffece0		

Actions to call a function

- Caller push parameters of callee to stack
- Caller execute command *call*, for example “call _Z3fooiii”.
  - 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

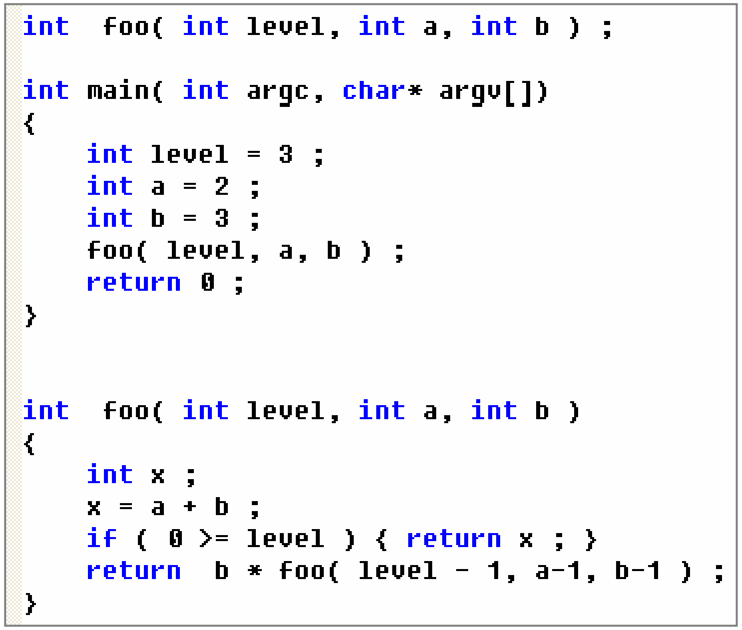
- Function calls (including parameter passing and placing object's address on the stack)
- Preservation of caller's stack frame
- Return-value communication
- Old stack-frame restore
- Return (give program control back to caller)

- 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

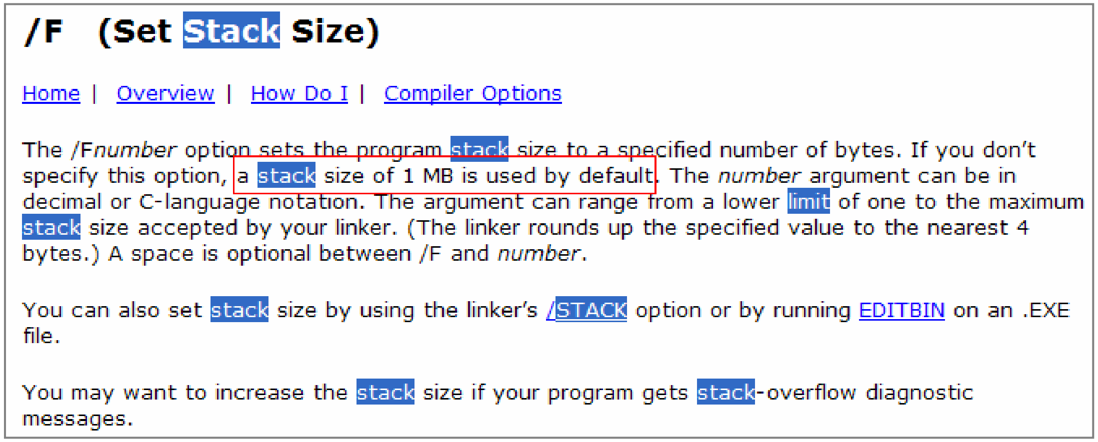
- Modify following code to show address of function parameter, local variable and content of return address, base pointer.
  Use “g++ -O0” to compile your code on workstation and check configuration of stack frame.
- What is configuration of stack frame using icpc –O0 ?
- What is configuration of stack 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?



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

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



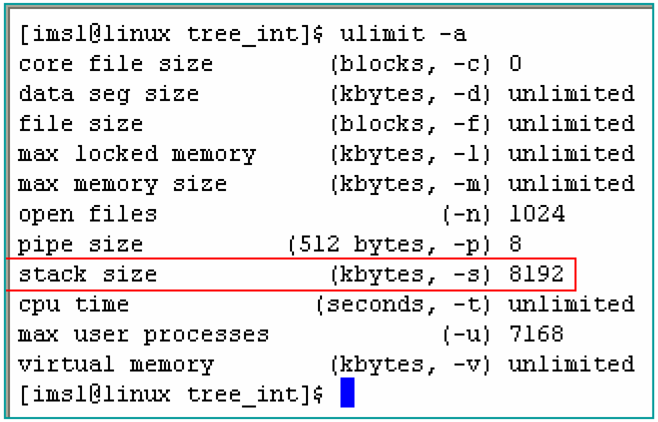
/F (Set Stack Size)

Home | Overview | How Do I | Compiler Options

The /F*number* option sets the program stack size to a specified number of bytes. If you don’t specify this option, a stack size of 1 MB is used by default. The *number* argument can be in decimal or C-language notation. The argument can range from a lower limit of one to the maximum stack size accepted by your linker. (The linker rounds up the specified value to the nearest 4 bytes.) A space is optional between /F and *number*.

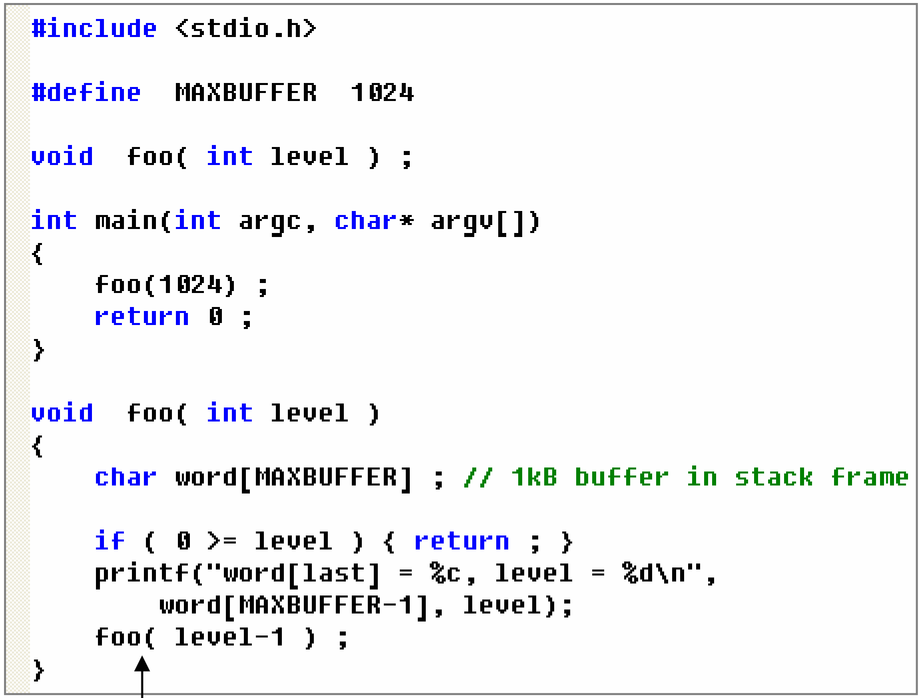
You can also set stack size by using the linker’s /STACK option or by running EDITBIN on an .EXE file.

You may want to increase the stack size if your program gets stack-overflow diagnostic messages.



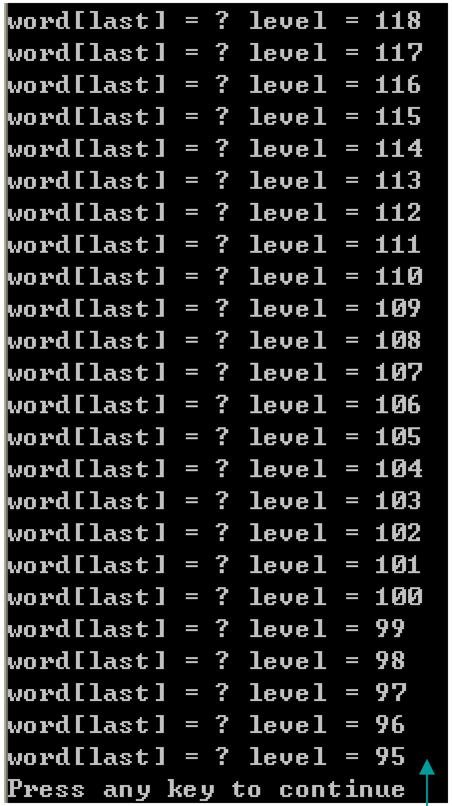
```
[imsl@linux tree_int]$ ulimit -a
core file size        (blocks, -c) 0
data seg size         (kbytes, -d) unlimited
file size             (blocks, -f) unlimited
max locked memory     (kbytes, -l) unlimited
max memory size       (kbytes, -m) unlimited
open files                    (-n) 1024
pipe size          (512 bytes, -p) 8
stack size            (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes            (-u) 7168
virtual memory        (kbytes, -v) unlimited
[imsl@linux tree_int]$
```

Test stack limit in VC6.0



```
#include <stdio.h>

#define  MAXBUFFER  1024

void  foo( int level ) ;

int main(int argc, char* argv[])
{
    foo(1024) ;
    return 0 ;
}

void  foo( int level )
{
    char word[MAXBUFFER] ; // 1kB buffer in stack frame

    if ( 0 >= level ) { return ; }
    printf("word[last] = %c, level = %d\n",
        word[MAXBUFFER-1], level);
    foo( level-1 ) ;
}
```

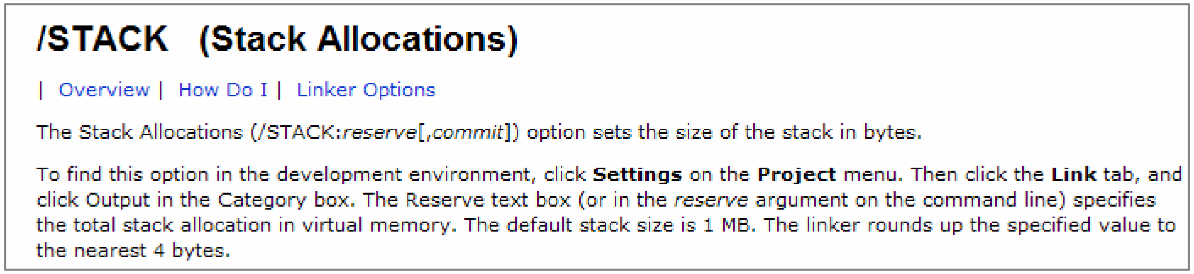
Recursive call



```
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
```

Level number cannot reach 1 since stack overflow

modify stack limit in VC6.0

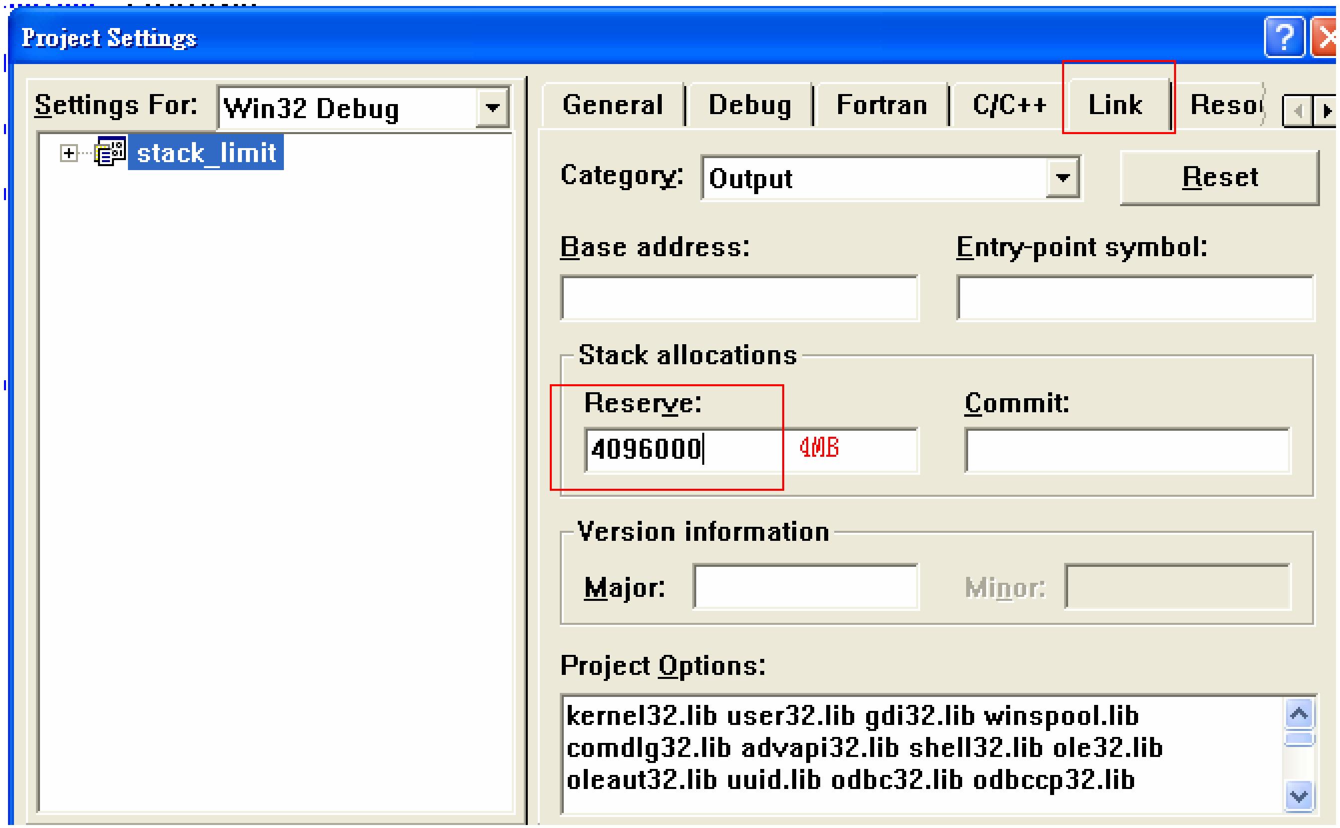


/STACK (Stack Allocations)

| Overview | How Do I | Linker Options

The Stack Allocations (/STACK:*reserve*[,*commit*]) option sets the size of the stack in bytes.

To find this option in the development environment, click **Settings** on the **Project** menu. Then click the **Link** tab, and click Output in the Category box. The Reserve text box (or in the *reserve* argument on the command line) specifies the total stack allocation in virtual memory. The default stack size is 1 MB. The linker rounds up the specified value to the nearest 4 bytes.



Settings For: Win32 Debug

stack_limit

Category: Output

Reserve: 4096000

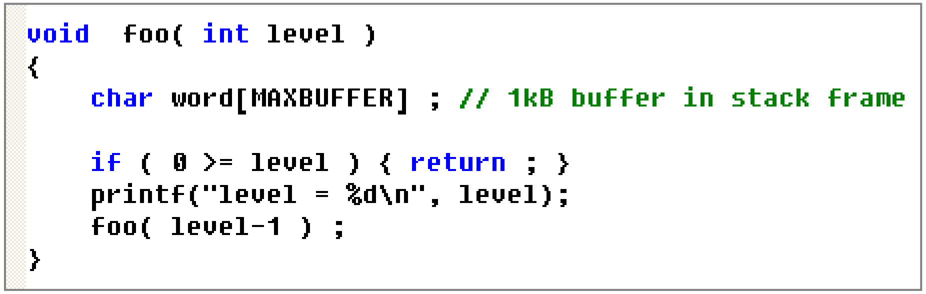
4MB

Project Options:

kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib

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?



```
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.