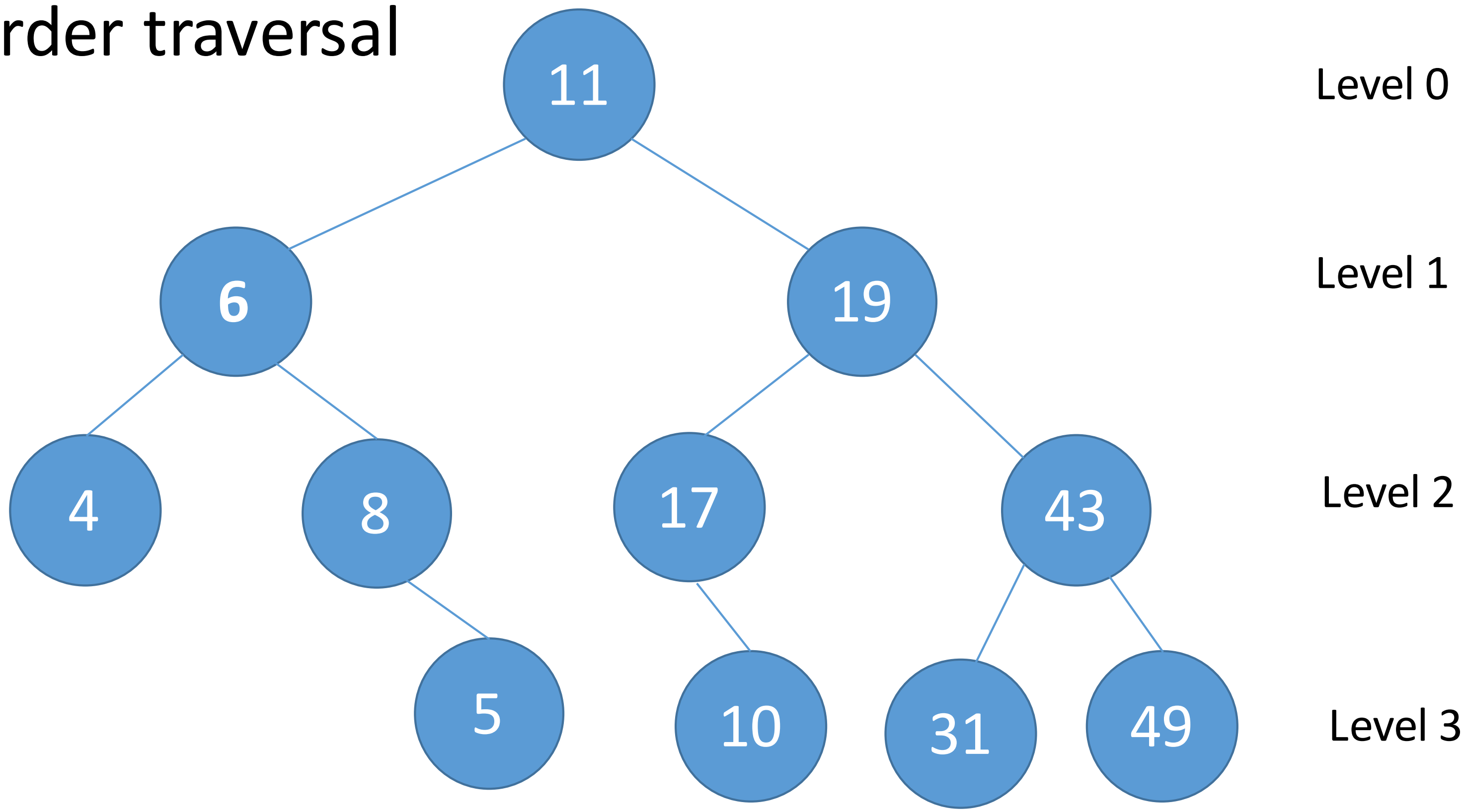
ECE264: Advanced C Programming

Summer 2019

Week 7: Binary Tree Traversal (contd.), Binary Search Trees, Misc. topics (const, variadic functions, macros, bitwise operations, bit fields), Parallel programming using threads

Breadth First Traversal (of a tree)

- Level order traversal



- 11, 6, 19, 4, 8, 17, 43, 5, 10, 31, 49

Breadth first traversal (of a tree)

```
void LOT(Node * root) {
 Queue q;
 push(&q, root);
 while (IsEmpty(&q) == false) {
   Node* headNode = Dequeue(&q)
   print(headNode->val)
   Enqueue(&q, headNode->leftChild);
   Enqueue(&q headNode->rightChild);
 }
}
```

Depth first traversal (of a tree) – iterative code

- Recall Preorder, Inorder, and Postorder were written as recursive codes

```
Preorder(Node* n) {
if(n->val == NULL)
      return;
print(n->val)
Preorder(n->leftChild);
Preorder(n->rightChild);
}
```

```
Inorder(Node* n) {
if(n->val == NULL)
      return;
Inorder(n->leftChild);
print(n->val)
Inorder(n->rightChild);
}
```

```
PostOrder(Node* n) {
if(n->val == NULL)
      return;
Postorder(n->leftChild);
Postorder(n->rightChild);
print(n->val)
}
```

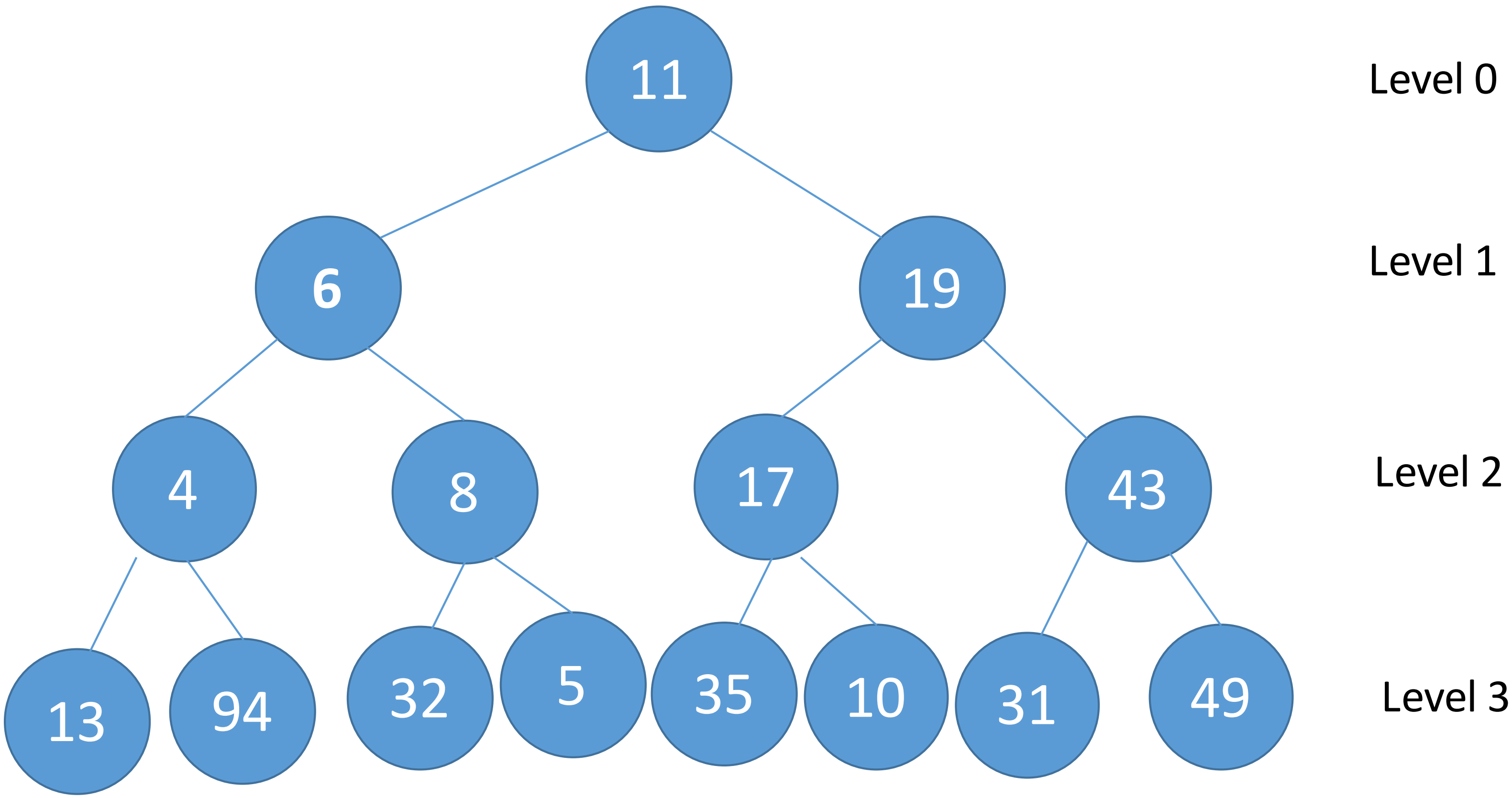
```
void Preorder(Node * root) {
 stack s;
 push(&s, root);
 while (IsEmpty(&s) == false) {
   Node* topNode = Pop(&s)
   print(topNode->val)
   Push(&q, topNode->rightChild);
   Push(&q topNode->leftChild);
 }
}
```

Exercise

What data structure do you need to use for writing an iterative code of Postorder traversal?

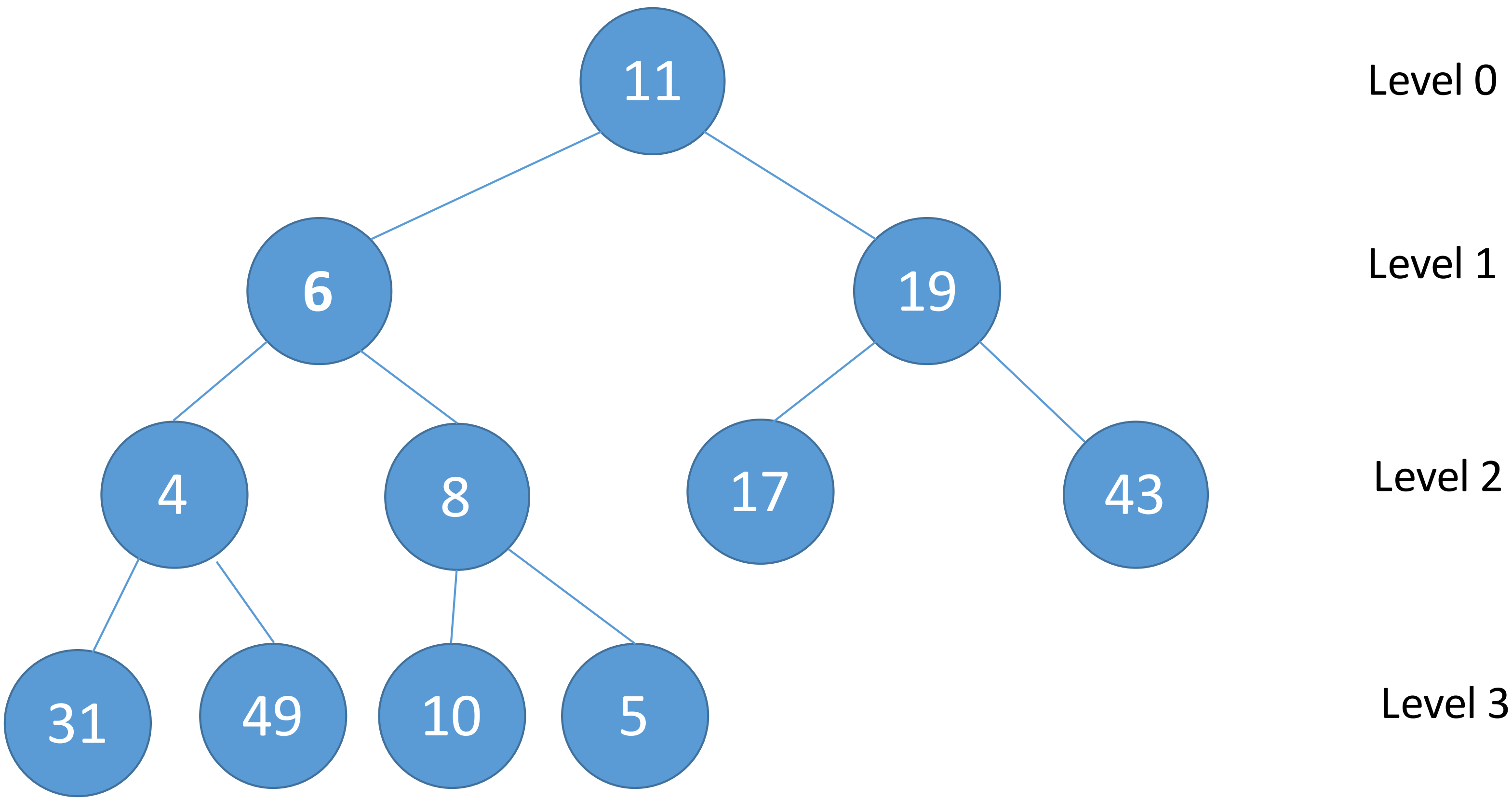
Full Binary Tree

- Every node except leaf has two children



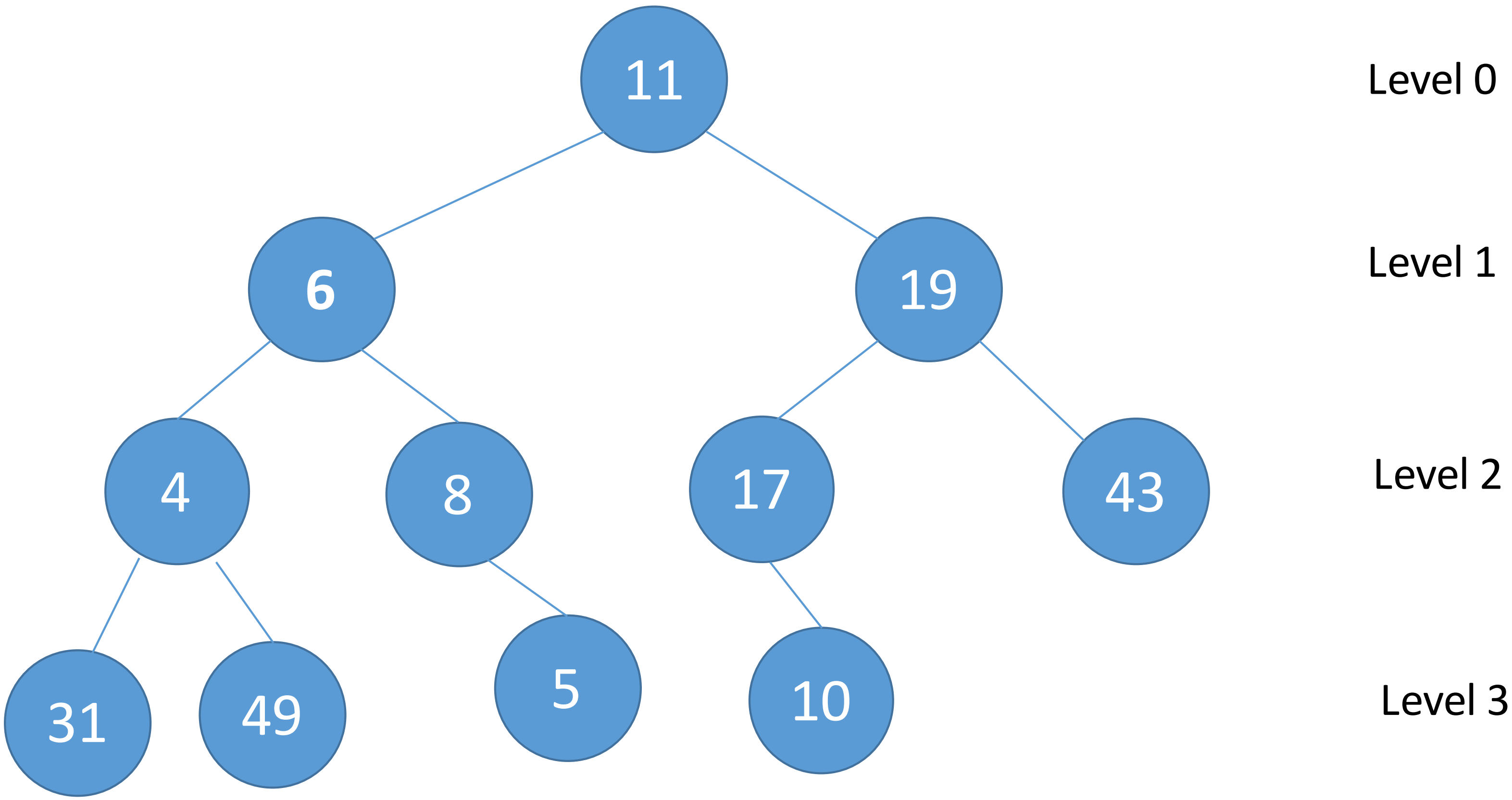
Complete Binary Tree

- Every level except the last is filled and all nodes at the last level are as far left as possible



Exercise

- Complete or Full ?

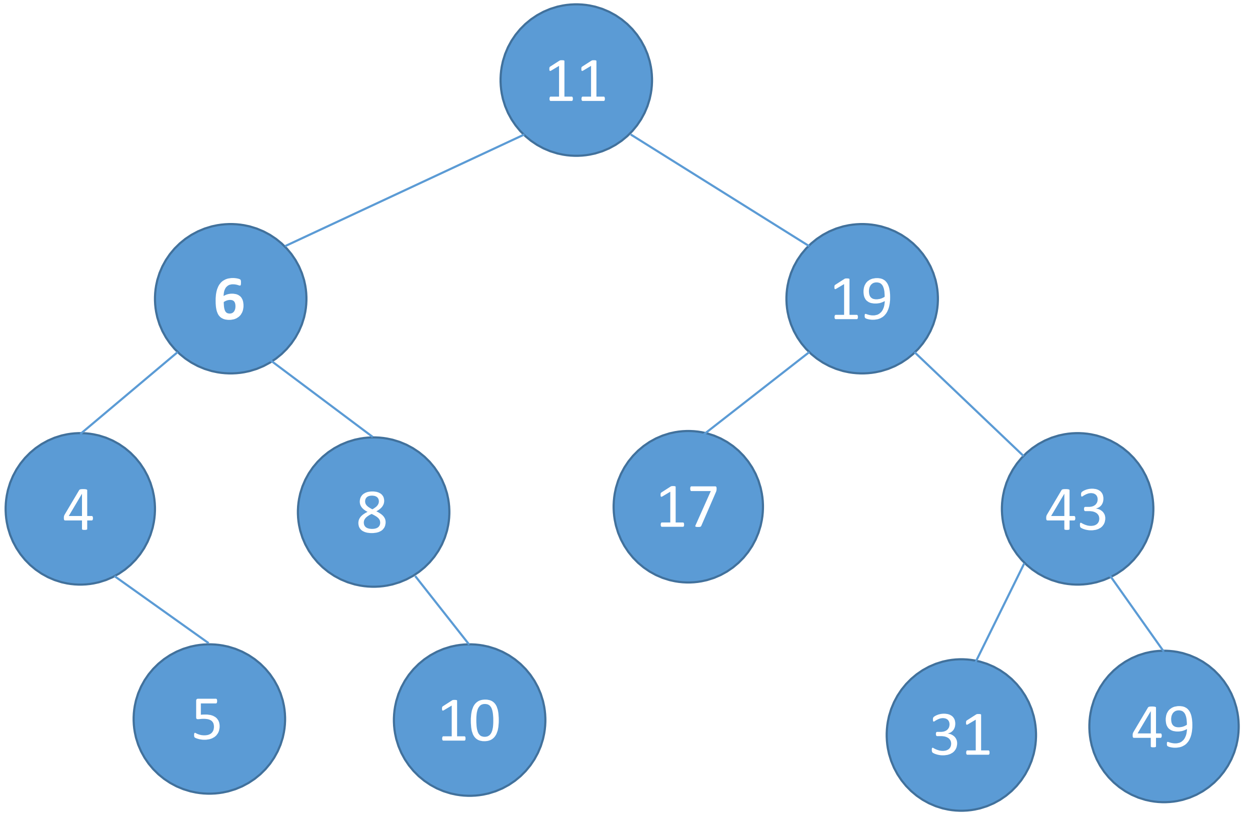


Binary Search Trees (BST)

- For efficient sorting, searching, retrieving
- BST Property:
    - Keys in left subtree are lesser than parent node key
    - Keys in right subtree are greater than parent node key
    - Duplicate keys not allowed

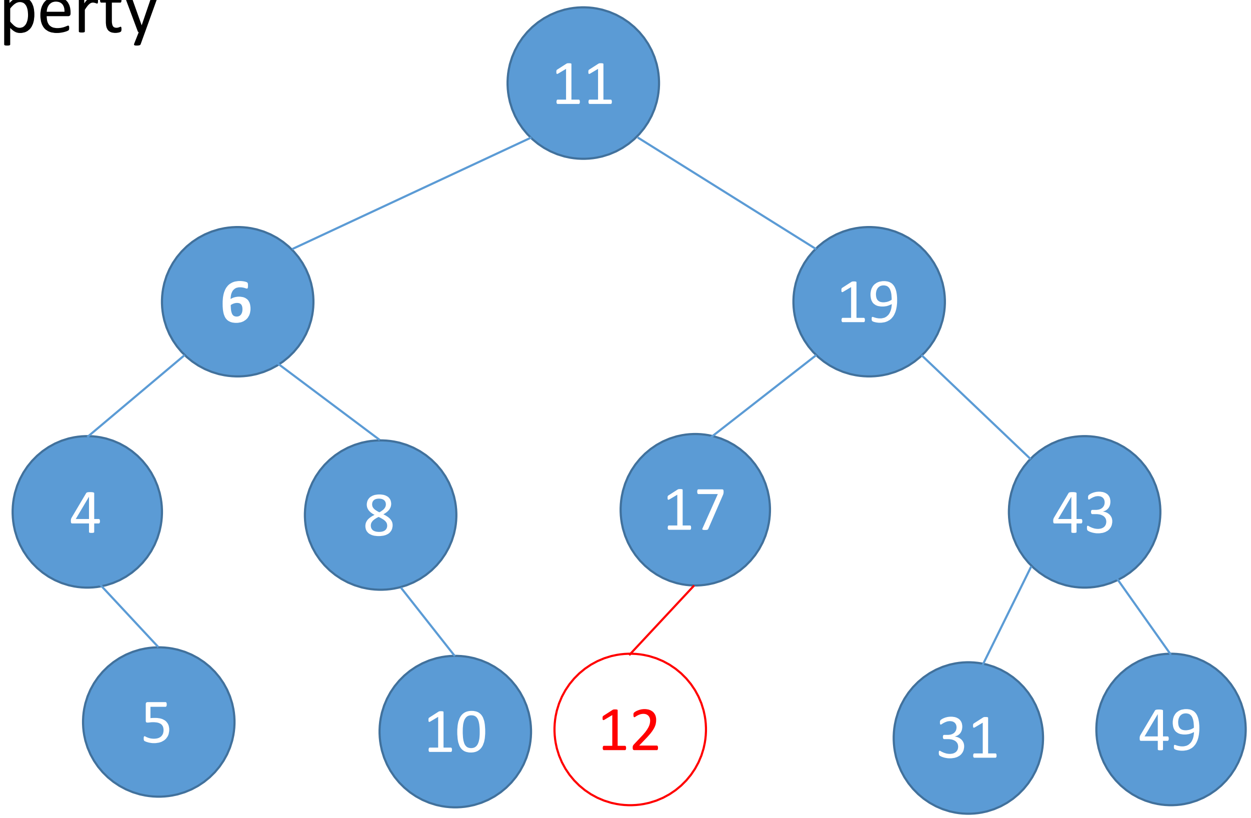
Binary Search Tree

- Example



Binary Search Tree

- Insertion: inserts element without violating the BST property



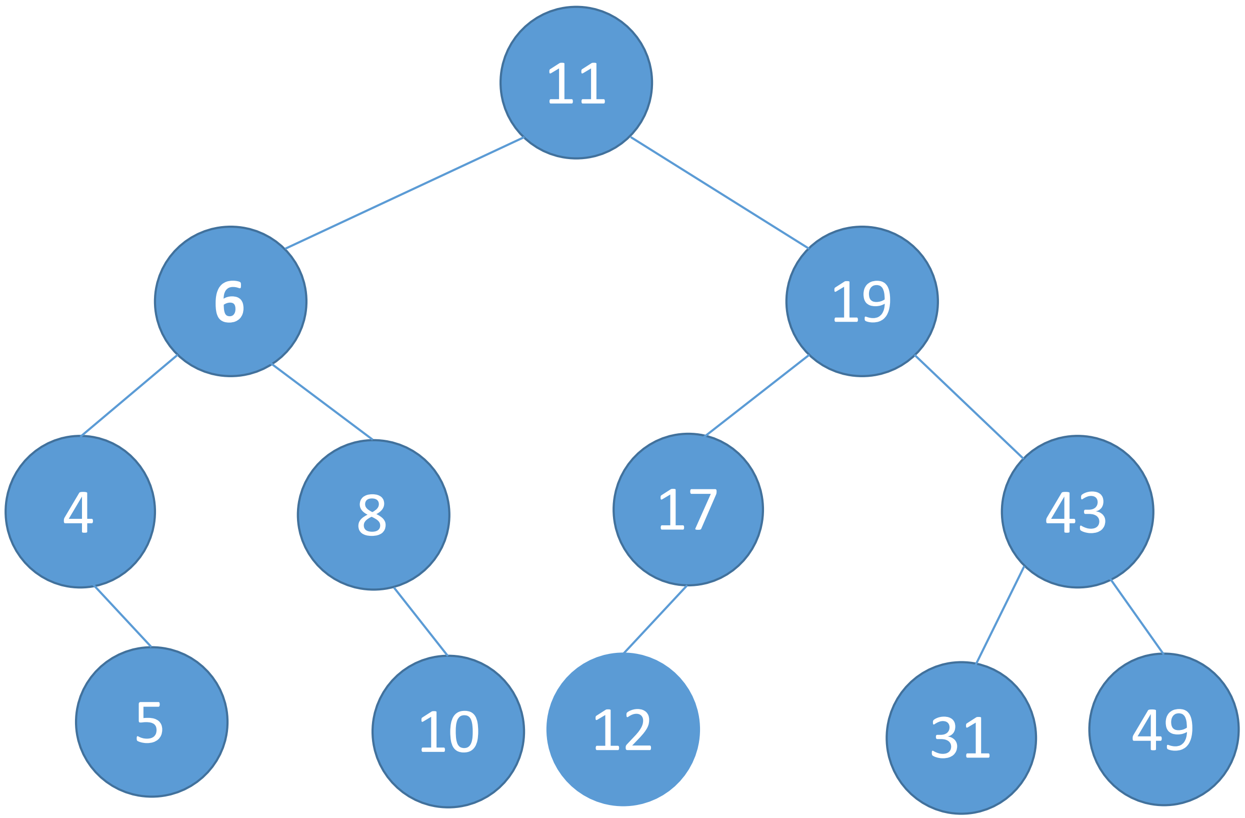
Binary Search Tree

- Insertion

```
bool add(TreeNode **rootPtr, int key) {
    if (*rootPtr == NULL) {
        *rootPtr = buildNode(key);
        return true;
    } else if ((*rootPtr)->val == key) {
        return false;
    } else if ((*rootPtr)->val < key) {
        return add(&((*rootPtr)->right), key);
    } else {
        return add(&((*rootPtr)->left), key);
    }
}
```

Binary Search Tree

- Search: returns true if key exists. False otherwise.



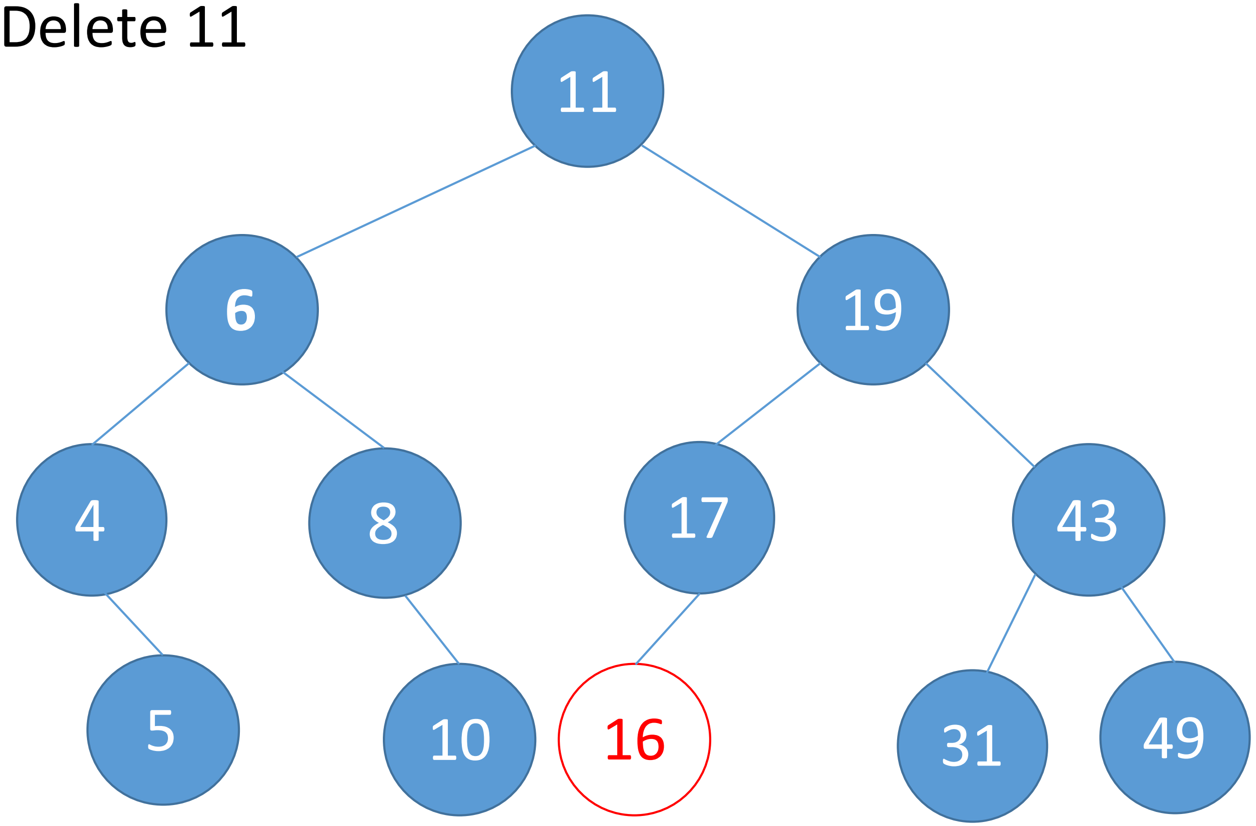
Binary Search Tree

- Search

```
bool Contains(Node* n, int key) {
    if(n == NULL)
        return false;
    if(n->val == key)
        return true;
    else if (n->val > key)
        return Contains(n->leftChild, key);
    else
        return Contains(n->rightChild, key);
}
```

Binary Search Tree

- Removal: remove without violating BST property
  - Delete 11

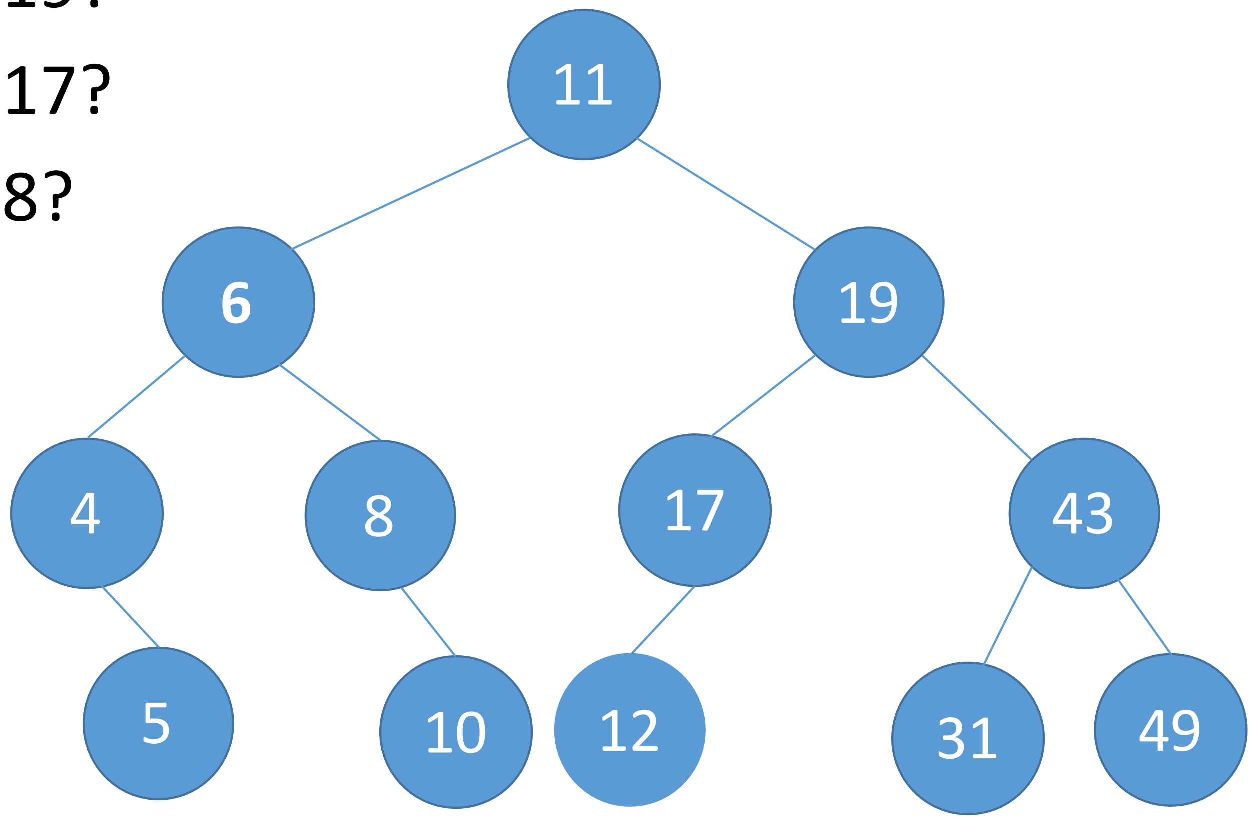


Binary Search Tree

- Removal cases
  - Not in a tree
  - Is a leaf
  - Has one or more children
- Return true if key removed. False otherwise.

Exercise

- Remove 19?
- Remove 17?
- Remove 8?



BST remove node

- Removal code: see bst.c

Applications – parsing of expression trees

- Goal: turn 2 + 3 into 2 3 +
    - We did this using stacks
- We can use binary trees to do the same job
- Binary trees allow us to create a more useful program
    - *earlier we never checked if the input was a valid infix expression*

We can build a basic compiler!

- Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
- *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. (2 + (3 * 7))
    - So can ignore order-of-operations (PEMDAS rule)

Fully-parenthesized expression – definition

- Recursive definition
  1. A number (floating point in our example)
  2. *Open parenthesis* ‘(‘ followed by

     fully-parenthesized expression followed by

     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by

     fully-parenthesized expression followed by

     closed parenthesis ‘)’

Fully-parenthesized expression – notation

1. E -> lit
2. E -> (E op E)

Expression parsing

Parsing is:

1. The process of determining if an expression is a valid fully-parenthesized expression
2. Breaking the expression into components
   - *Why do we need this step?*

     We need not worry if a number has single digit, or multiple digits, or how many blank spaces separate two components etc.

Parsing

Rules: 1) E -> lit 2) E -> (E op E)

- Get the next token
- If the next token is a VAL (matches rule 1), return true.
- If the next token is an LPAREN match all of rule 2:
  - We have already seen the LPAREN, so the next thing we expect to see is a fully-parenthesized expression. We can just call *this same function* recursively to do that! If the recursive call returns true, it means we have found a fully-parenthesized expression
  - The next part of rule 2 is to match an operation, so we grab the next token and see if it is an ADD, SUB, MUL, or DIV. If it is, we continue.
  - Then we call this same function recursively again to find another fully-parenthesized expression.
  - Finally, we grab the next token to see if it is an RPAREN. If it is, we have matched rule 2, and this is a fully-parenthesized expression, so we return true.

Parsing

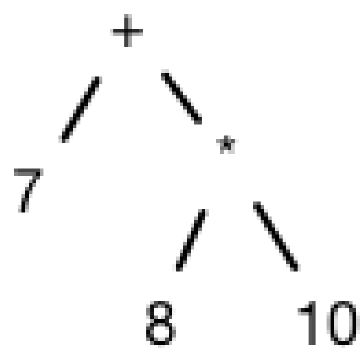
Example of a recursive descent parser

- Can check if an expression is fully parenthesized
- Can’t check if a C program is valid

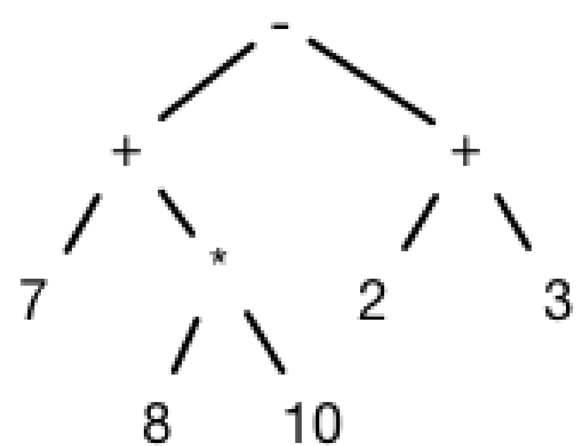
Expression trees

- Each leaf node is a number, non-leaf (interior) node a binary operation.

(7 + (8 * 10))



((7 + (8 * 10)) - (2 + 3))



Building expression trees

- Can build while parsing a fully parenthesized expression

  Via bottom-up building of the tree

- Create subtrees, make those subtrees left- and right-children of a newly created root.

  Modify recursive parser:

  1. If token == VAL, return a pointer to newly created node containing a number
  2. Else
     1. store pointers to nodes that are left- and right- expression subtrees
     2. Create a new node with value = ‘OP’

Expression trees

- Example: (7 + (8 * 10))

Exercise

What traversal order needs to be followed for tree deletion?

Type Qualifiers

- const, volatile, restrict
- Examples:

```
const int x=10; //equivalent to: int const x=10;
volatile int y=0; //eq to: int volatile y;
int *restrict c;
```

Const Qualifier

- The type is a constant (cannot be modified).
- `const` is the keyword

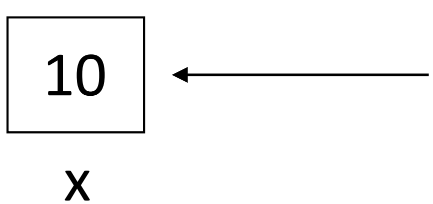
```
const int x=10; //x is a constant integer (hence, in RO
memory). x cannot be modified.
```

- We can also declare a `const` variable as:

```
int const x=10;
```

Const Properties

- Needs to be initialized at the time of definition
- Can’t modify after definition
- `const int x=10;`
  `x=20; //compiler would throw an error`
- `int const x=10;`
  `x=10; //can’t even assign the same value`
- `int const y; //uninitialized const variable y. Useless.`

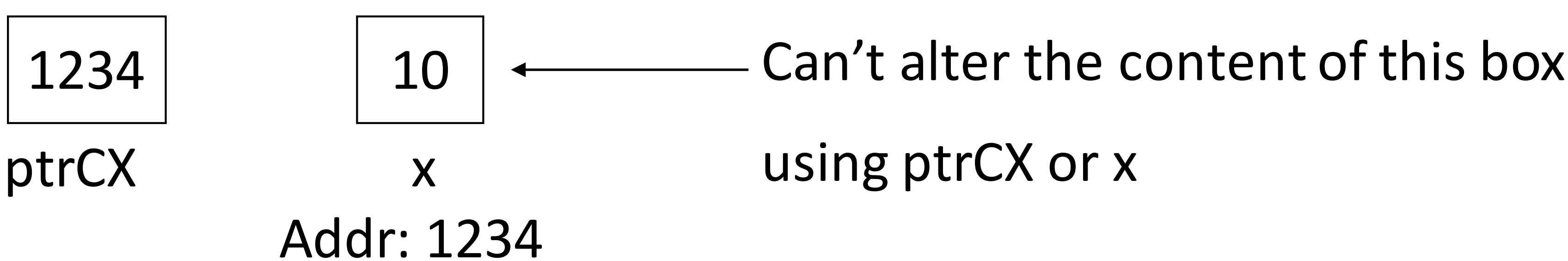


Can’t alter the content of this box

Const Example1 (error)

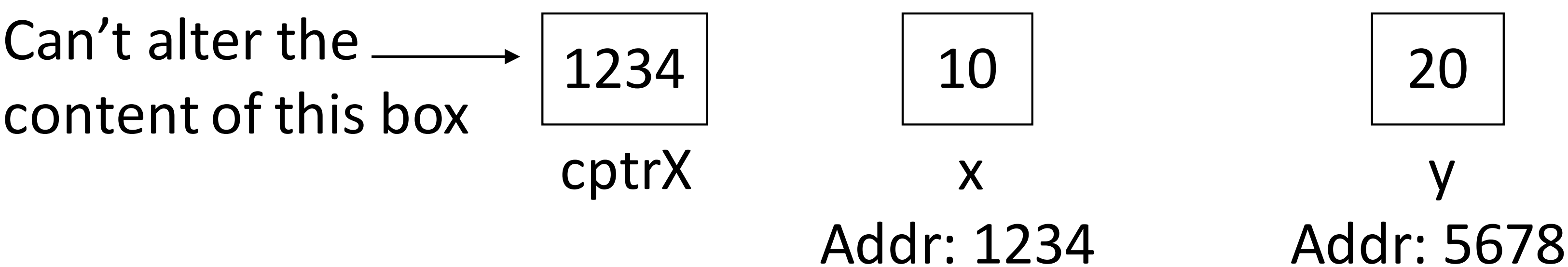
```
/*ptrCX is a pointer to a constant integer. So, can't
modify what ptrCX points to.*/
const int* ptrCX;
int const* ptrCX;

int const x=10;
ptrCX = &x;
*ptrCX = 20; //Error
```



Const Example2 (error)

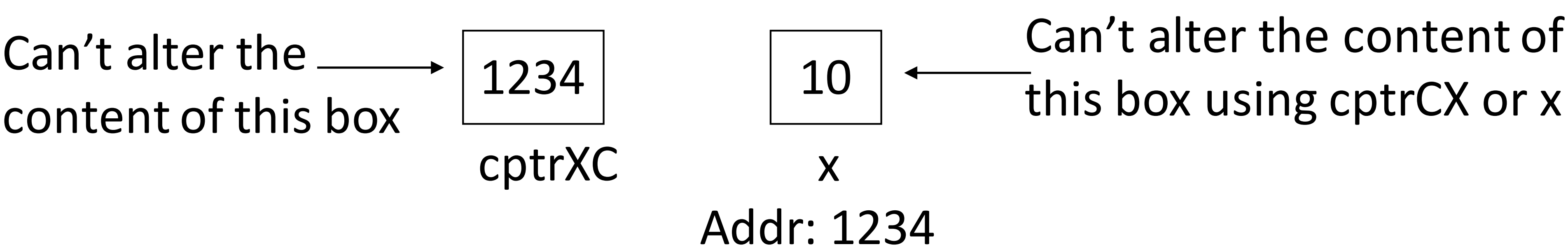
```
/*cptrX is a constant pointer to an integer. So, can't
point to anything else after initialized.*/
int x=10, y=20;
int *const cptrX=&x;
cptrX = &y; //Error
```



Const Example3 (error)

```
/*cptrXC is a constant pointer to a constant integer. So,
can't point to anything else after initialized. Also,
can't modify what cptrXC points to.*/

const int x=10, y=20;
const int *const cptrXC=&x;
int const *const cptrXC2=&x; //equivalent to prev. defn.
cptrXC = &y; //Error
*cptrX = 40; //Error
```



Const Example4 (warning)

```
/*p2x is a pointer to an integer. So, we can use p2x to
alter the contents of the memory location that it points
to. However, the memory location contains read-only data -
cannot be altered. */

const int x=10;
const int *p1x=&x;
int *p2x=&x; //warning
*p2x = 20; //goes through. Might crash depending on memory
location accessed
```

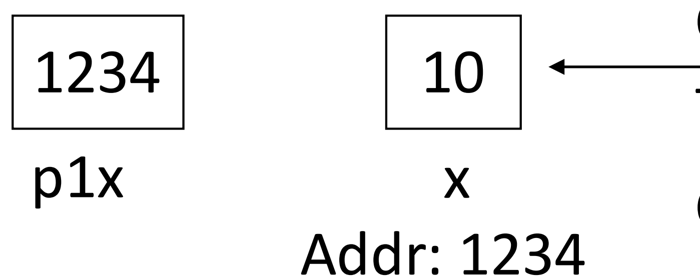
p2x	p1x	x
1234	1234	10
		Addr: 1234

Can't alter the content of this box using p1x or x. Can alter using p2x.

Const Example5 (no warning, no error)

```
/*p1x is a pointer to a constant integer. So, we can't use
p1x to alter the content of the memory location that it
points to. However, the memory location it points to can
be altered (through some other means e.g. using x)*/

int x=10;
const int *p1x=&x;
```



Can't alter the content of this box using p1x.

Can alter using x.

Const Example6 (warning)

```
/*p1x is a constant pointer to an integer. So, we can use
p1x to alter the contents of the memory location that it
points to (and we can't let p1x point to something else
other than x). However, the memory location contains read-
only data - cannot be altered. */

const int x=10;
int *const p1x=&x;//warning
*p1x = 20; //goes through. Might crash depending on memory
location accessed
```

Can't alter the content of this box

1234

p1x

10

x

Addr: 1234

Can't alter the content of this box using x. Can alter using p1x.

Const Example7 (no warning, no error)

```
/*p1x is a pointer to a constant integer. So, we can’t use
p1x to alter the content of the memory location that it
points to. However, the memory location it points to can
be altered (through some other means e.g. using x)*/

int x=10;
const int *const p1x=&x;
```

Can’t alter the content of this box → 1234

p1x

10 ← Can’t alter the content of this box using p1x.

x

Addr: 1234

Can alter using x.

const Case Study - strchr

- `strchr` is a library function that accepts a string and a char and returns a pointer to the first occurrence of the char
  - `char* strchr(const char* str, char c)`
- Returns a pointer to a `char`. So, we could modify the content of the memory location that the return value (a pointer) points to!
  - Exercise: is this an error or warning?

volatile Qualifier

- Hint to the compiler indicating that a variable can change in unexpected ways (is volatile)
  - signals the compiler to not do any optimizations with the variable
  - Example:

```
volatile int* x = (volatile int *)0x1234; //x
is a pointer to a memory location with address
0x1234
```

volatile Qualifier

- Hint to the compiler indicating that a variable can change in unexpected ways (is volatile)
  - signals the compiler to not do any optimizations with the variable
  - Example:

```
volatile int* x = (volatile int *)0x1234; //x
is a pointer to a memory location with address
0x1234
```

- Special memory locations in embedded systems programming are assigned certain addresses
  - Control registers, output buffers, input buffers
- For example, memory location at address 0x1234 could be a control register.
  - We can then access this register as we would access an `unsigned int`:

```
unsigned int *ctrlReg = (unsigned int *) 0x1234;

printf("current val of ctrl reg: %u", ctrlReg);

*ctrlReg=0x00000001; /*setting least significant bit to
indicate that input buffer has some data (we put some
data in input buffer and whoever is interested may
consume it)*/
```

volatile Qualifier

```
unsigned int* ctrlReg = (unsigned int *)0x1234;
while (0 == *ctrlReg) {
//no data in input buffer. do some other work
}
```

sample assembly code (when optimizations turned on):

```
mov ctrlReg, #0x1234
mov a, @ctrlReg
loop:
...
bz loop
```

volatile Qualifier

```
volatile unsigned int* ctrlReg = (volatile unsigned int *)0x1234;

while (0 == *ctrlReg) {

//no data in input buffer. do some other work

}
```

sample assembly code (when optimizations turned on):

```
mov ctrlReg, #0x1234
loop:
mov a, @ctrlReg
...
bz loop
```

restrict Qualifier

- Introduced in C99
- May only be used with pointers
- Tells that the pointer is the only way to access a memory location

```
int * restrict source;
```

Example:

https://en.wikipedia.org/wiki/Restrict

Variadic Functions

- Functions that can take variable number of arguments.
- Examples
  - Concatenating strings str1 + str2 + . . .
  - Adding numbers num1 + num2 + num3 +. . .
  - `printf` and `scanf` functions
- Functions that have indefinite 'arity' – number of operands.

Variadic Functions - Motivation

- Adding two integers
  - `int add2(int num1, int num2)`
- Adding three integers
  - `int add3(int num1, int num2, int num3)`
- Adding ‘N’ integers?
  - `int addN(int count, . . .)`

*Flexibility in programming**

*N numbers can be added in a loop. In this example, we would like to ‘modularize’ our addition.

Variadic Functions - definition

```
int addN(int count, . . .)
```

Fixed parameter

Variable number of parameters (represented as three dots)

Fixed parameter must precede three dots.

Variadic Functions

- Useful macros and types
  1. `va_list` //type to hold the list of arguments
  2. `va_start`
  3. `va_arg`
  4. `va_end`
  5. `va_copy //used to copy arguments`

  Macros for stepping through the list of arguments

- Include `stdarg.h` (`varargs.h` is the older version. Not used anymore)

`va_list`

- Type to hold the variable arguments passed while calling a function
- Example:
  - `va_list nums;`
- Also used as a parameter to other macros used in a variadic function definition

`va_start`

- Macro used to initialize the `va_list` variable
  - `va_start(nums, count);`

Name of the type declared previously using `va_list`

Name of the fixed parameter

- Also used as a parameter to other macros used in a variadic function definition

`va_arg`

- Macro used to step through the argument list

```
va_arg(nums, type);
```

Name of the type declared previously using `va_list`

Name of the data type of the argument `(int, float, etc.)`

- A call to `va_arg` modifies `nums`. Next call returns the next argument in the list
- *Caution: calling this macro more than required number of times will take you past the end of the argument list*

`va_end`

- Macro that must be called whenever `va_list` is used in a function
- Cleanup macro

```
va_end(nums);
```

Name of the type declared previously using `va_list`

Example – Adding N Numbers

```
int addN(int count, . . .) {
   va_list nums;
   int sum=0, i=0;
   va_start(nums, count)
   for(i=0;i<count;i++)
      sum += va_arg(nums, int);
   va_end(nums)
   return sum;
}

int main() {

      printf(“Sum:%d\n”,addN(3,100,101,102));

}
```

Exercise

1. Write a variadic function to find the minimum of N numbers

2. Write your version of the printf function that interprets and prints only integers (%d) and floats (%f). Internally, you can use printf, the built-in function.

```
myprintf("%d%f",x,y) //should print x and y values
myprintf("%cdef") //should print %cdef
```

Variadic Functions - vulnerability

- Format string attacks

```
char* str="ECE";
printf("Hello %s",str);
```

Format parameter

Format string attack

- What you can do:
  - Crash someone's program
  - View stack content
  - Overwrite return address

```
//crashing program
int main() {
   printf(%s%s%s%s%s%s%s%s%s);
}
```

Macros

- We have seen preprocessor macros
  - `#define, #ifdef, #ifndef, #else etc.`

  - `#define MAXNAMELEN 80 //the token MAXNAMELEN is replaced by 80 whenever it appears in a program (during compilation)`

    E.g. `char buf[MAXNAMELEN];` //declares a variable buf and reserves 80 bytes of memory for it.

More #define

- We can pass parameters to #define
- Examples:

```
#define INCREMENT(x) x++
#define ADD(a,b) a+b
#define MAX(a,b) (a >= b)?a:b

 int main() {
    int a=10;
    int b=INCREMENT(a);
    int c=ADD(a,b);
    int maxAC = MAX(a,c);
    printf("a:%d b:%d c:%d max:%d\n",a,b,c,maxAC);
}
```

- Sometimes more efficient than writing functions for smaller tasks
  - Expanded inline – no creation of stack frames

```
#define INCREMENT(a) a++
int main() {
int a=10;
int b=INCREMENT(a);
printf("a:%d b:%d\n",a,b);
}
```

```
#define INCREMENT(a) a++
int main() {
int a=10;
int b=a++;
printf("a:%d b:%d\n",a,b);
}
```

- However, there are side effects

```
#define MUL(x, y) x*y
int main() {
int e=MUL(2+3,4+5);
printf(“e:%d\n”,e);
}
```

```
#define MUL(x, y) x*y
int main() {
int e=2+3*4+5; //not (2+3) * (4+5) as expected
printf(“e:%d\n”,e);
}
```

- Can fix this easily – add parenthesis around parameters

```
#define MUL(x, y) (x)*(y)
```

- Can write multi-line macros using \

```
#define SWAP(x, y, type) { \
type tmp=x;\
x=y;\
y=tmp;\
}

int main() {
int x=10, y=20;
SWAP(x,y, int);
printf(“x:%d y:%d\n”,x,y);
}
```

- Can pass pointers to SWAP

```
#define SWAP(x, y, type) { \
type tmp=x;\
x=y;\
y=tmp;\
}

int main() {
int x=10, y=20;
int *px=&x, *py=&y;
SWAP(px,py, int*);
printf("*px:%d *py:%d\n",*px,*py);
}
```

- However, there is a problem with SWAP

```c
#define SWAP(x, y, type)
{ \
type tmp=x;\
x=y;\
y=tmp;\
}
```

```c
int main() {
int x=10, y=20;
if (x > 5)
{
        int tmp=x;
        x=y;
        y=tmp;
};
else
        printf("Not allowed to swap\n")
}
//Throws compiler error.
```

//SWAP(x, y, int);
expanded

- Solution: enclose SWAP in a `do-while` loop

```
#define do { SWAP(x, y, type) { \
type tmp=x;\
x=y;\
y=tmp;\
} while(0);
```

- Syntax of `do-while`:

```
do {
...
}while(cond);
```

- Consider

```
#define SQUARE(x) x*x
int main() {
      printf(“%d\n”,4/SQUARE(2));
}
```

- How to fix this?
  - “inlining” is an option

`inline` Functions

- C99 introduced them

```
inline int SQUARE(x)
{
    return x*x;
}

int main() {
    printf(“%d\n”,4/SQUARE(2));
}
```

Concatenation and Stringizing

- ## (concatenation) and # (stringizing)

```
#define GETNEWTOKEN(a,b) a##b
#define GETSTR(a) #a
int main() {
 printf(“%d\n”,GETNEWTOKEN(12,34));
//prints 1234
 int i=264;
 printf(“%s\n”,GETSTR(myVal)); //prints “i”
 }
```