5. Introduction to Analysis of Algorithms

Analysis of Algorithms

Performance evaluation in terms of computer resources used:

- memory/disk space (space complexity)
- computing/cpu time (time complexity)

Space Complexity

- memory/disk space used by the algorithm

- Searching
  - Linear Search = n
  - Binary Search = n

- Sorting
  - Bubble Sort = n
  - Insertion Sort = n

Time Complexity

- computing/cpu time used by the algorithm
- Question: How to estimate the time required by a program?

Factors affecting execution/running time:

- computer used
- compiler
- **algorithm used**
- **input to the algorithm**

Two ways to estimate

1. empirical
   – code it up, then run it along with a timer

2. algorithm analysis
   – obtain the expected running time of a pseudocode (or actual code) without actually running it

Empirical Running Time

```
#include <time.h>

clock_t start, stop;
double cpu_time_used;

start = clock();

/* DO THE WORK HERE */


stop = clock();

cpu_time_used = ((double)(stop-start)) / CLOCKS_PER_SEC
```

Algorithm Analysis

- A function that maps problem size(input size) into the time required to solve the problem, T(n)

- $T(n) \approx O(n^2)$

Mathematical Definitions

- O (Big-Oh) - upper bound

  $T(n) = O(f(n))$ if there are constants c and n_0 such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$.

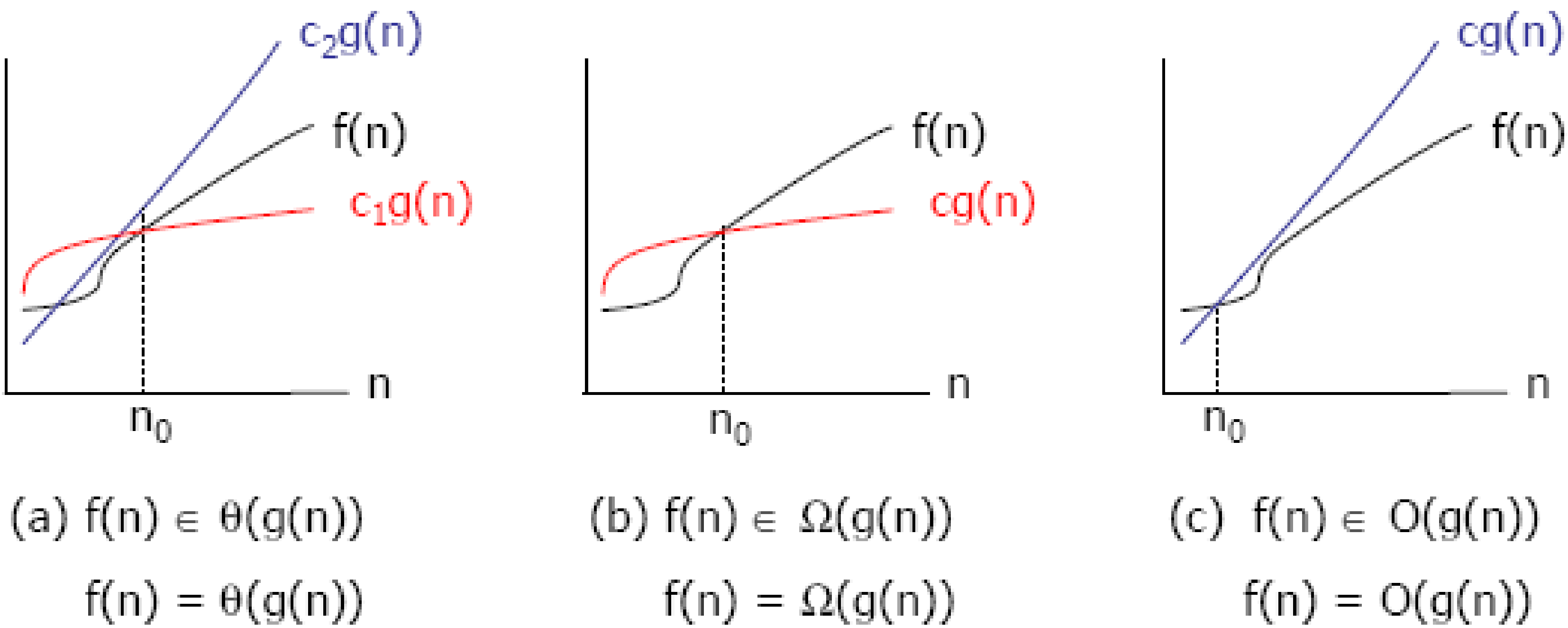
- Ω (Big-Omega) - lower bound

  $T(n) = \Omega(g(n))$ if there are constants c and n_0 such that $T(n) \geq c \cdot g(n)$ when $n \geq n_0$.

- Θ (Theta) - asymptotically tight bound

  $T(n) = \theta(h(n))$ if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

Mathematical Definitions



Reference: Jacildo A.J. Mathematical Foundations. CMSC 142 slides.

Big-Oh notation – upper bound

T(n) = O(f(n)) if there are constants c *and* n_0 such that T(n) ≤ c f(n) when $n \geq n_0$.

Example:

Compare T(n) = 1,000n and f(n) = n^2.

Note: T(n) is larger than f(n) for small values of n, but n^2 grows at a faster rate than T(n).

Thus, T(n) ≈ O(n^2), for $c=1$ and $n_0=1{,}000$ OR $c=10$ and $n_0=100$.

$c \cdot n^2$ is at least as large as $1{,}000n$.

Big-Oh notation – upper bound

When we say $T(n) = O(f(n))$, then

- $f(n)$ is an upper bound on $T(n)$
- $f(n) = \Omega(T(n))$, $T(n)$ is a lower bound on $f(n)$

If $g(n) = 2n^2$, then

- $g(n) = O(n^2)$ or $O(n^3)$ or $O(n^4)$
- but the best is $O(n^2)$ – tight bound

Lower bound vs. Upper bound

- Lower bound analysis
  - are used to describe how fast a given problem can be solved
- Upper bound analysis
  - are used to describe the worst-case performance of an algorithm

Complexity classes

Big-Oh notation	Description (speed of execution)
O(1)	constant
O(log n)	logarithmic
$O(\log^2 n)$	log-squared
O(n)	linear
O(n log n)	
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$, $k \geq 1$	polynomial
$O(a^n)$, $a > 1$	exponential

Growth rate of complexity classes

class	n=2	n=16	n=256	n=1024
1	1	1	1	1
log n	1	4	8	10
n	2	16	256	1024
n log n	2	64	2948	10240
n^2	4	256	65536	1048576
n^3	8	4096	16777216	1.07E+09
2^n	4	65536	1.16E+77	1.8E+308

Reference: Jacildo A.J. Algorithm Analysis Techniques. CMSC 142 slides.

Analysis - Example

- Calculate: $\sum_{i=1}^{n} i^3$

```
int sum(int n) {
   int i, partial_sum;
   partial_sum = 0;
   for (i=1; i<=n; i++)
      partial_sum += i * i * i;
   return partial_sum;
}
```

Analysis - Example

```
int sum(int n) {
   int i, partial_sum;
   partial_sum = 0;                      (1)
   for (i=1; i<=n; i++)                  (2)
      partial_sum += i * i * i;          (3)
   return partial_sum;                   (4)
}
```

- Lines 1 and 4: 1 unit each
- Line 3: 3 units per time executed = 3n
- Line 2: 1 to initialize, n+1 tests, n increments = 2n+2
- Total: 5n+4 , Thus T(n) = O(n)

General Rules

- **Rule 1 — FOR loops**

  The running time of a for loop is at most the running time of the statement inside the for loop (including tests) times the number of iterations.

General Rules

Rule 2 – Nested FOR loops

The total running time of a statement inside a group of nested for loops is the running time of the statement multiplied by the product of the sizes of all the for loops.

Example:

```
for (i=0;  i<n; i++)
      for (j=0; j<n; j++)
               k++;
```

General Rules

Rule 3 – Consecutive statements

The maximum is the one that counts:

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then

$T_1(n) + T_2(n) = max(O(f(n)), O(g(n)))$

Example:

```
for (i=0;  i<n; i++)
      a[i] = 0;
for (i=0;  i<n; i++)
      for (j=0;  j<n; j++)
            a[i] += a[j] + i + j;
```

General Rules

Rule 4 — IF/ELSE:

For the fragment

```
if (cond)
        S1
else
        S2
```

the running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

More Examples

```
sum = 0;
for (i=0; i<n; i++)
    sum++;
```

- $T(n) \approx O(n)$

More Examples

```
sum = 0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        sum++;
```

- $T(n) \approx O(n^2)$

More Examples

```
sum = 0;
for (i=0; i<n; i++)
    for (j=0; j<n*n; j++)
        sum++;
```

- $T(n) \approx O(n^3)$

More Examples

```
sum = 0;
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        sum++;
```

$$\sum_{j=1}^{n-1} j$$

- i=0, 0
- i=1, 1
- i=2, 2
- i=3, 3
- i=n-1, n-1
- $T(n) \approx O(n^2)$

More Examples

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

- $S = 1 + 2 + 3 + .. + (i-1)$
- $S = (n-1) + (n-2) + (n-3) + .. + 1$
- $2S = n + n + n + .. + n$
- $2S = n(n-1)$
- $S = n(n-1)/2$
- $T(n) = O(n^2)$

More Examples

```
sum = 0;
for (i=0; i<n; i++)
    if(i%2==0)
        sum++;
    else
        for (j=0; i<n; i++)
            sum+=2;
```

- $T(n) = O(n^2)$

Search Algorithms

- Linear Search
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)
- Binary Search
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)

Linear Search

Linear Search - list is not necessarily sorted

```
int linear_search(int a[], int n, int x){
    int i;
    for(i=0;i<n;i++)
        if (a[i]==x) break; //found x
    return(i<n); //if x is found i<n else i=n
}
```

a	5	2	1	4	3	
	a[0]	a[1]	a[2]	a[3]	a[4]	n=5

Search Algorithms

- Linear Search
  - Best-case complexity: O(1)
  - Worst-case complexity: O(n)
- Binary Search
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)

Binary Search

- Binary Search - list is sorted

```
int binary_search(int a[], int n, int x){
    int lower,upper, middle;
    lower = 0;
    upper = n-1;
    while(lower <= upper){
        middle = (lower + upper)/2;
        if (x > a[middle]) lower = middle+1;
        else if (x < a[middle] upper = middle - 1;
        else return(1);
    }
    return(0);
}
```

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

25	31	32	43	50
L=5		M=7		U=9

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

25	31	32	43	50
L=5		M=7		U=9

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

25	31	32	43	50
L=5		M=7		U=9

25	31
L=M=5	U=6

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

25	31	32	43	50
L=5		M=7		U=9

25	31
L=M=5	U=6

Binary Search

x=24	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	n=10
a	2	5	12	15	21	25	31	32	43	50	
	L=0				M=4					U=9	

25	31	32	43	50
L=5		M=7		U=9

25	31
L=M=5	U=6

	25
U=4	L=5

Search Algorithms

- Linear Search
  - Best-case complexity: $O(1)$
  - Worst-case complexity: $O(n)$
- Binary Search
  - Best-case complexity: $O(1)$
  - Worst-case complexity: $O(\log n)$

Sorting Algorithms

- Bubble Sort
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)
- Insertion Sort
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)

Bubble Sort

```
void bubble_sort(int a[], int n){
    int i,j;

    for(i=0;i<n-1;i++)
        for(j=1;j<n;j++)
            if (a[j]<a[j-1])
                swap(&a[j],&a[j-1]);
}
```

Bubble Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	5	4	3	2	1	

Bubble Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	5	4	3	2	1	
	4	5	3	2	1	j=1

Bubble Sort (inversely sorted input)

n=5

	a[0]	a[1]	a[2]	a[3]	a[4]	
i=0	5	4	3	2	1	
	4	5	3	2	1	j=1
	4	3	5	2	1	j=2

Bubble Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	5	4	3	2	1	
	4	5	3	2	1	j=1
	4	3	5	2	1	j=2
	4	3	2	5	1	j=3

Bubble Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	5	4	3	2	1	
	4	5	3	2	1	j=1
	4	3	5	2	1	j=2
	4	3	2	5	1	j=3
	4	3	2	1	5	j=4

Bubble Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	5	4	3	2	1	
i=1	4	3	2	1	5	
	3	4	2	1	5	j=1
	3	2	4	1	5	j=2
	3	2	1	4	5	j=3

Bubble Sort (inversely sorted input)

n=5

	a[0]	a[1]	a[2]	a[3]	a[4]
i=0	5	4	3	2	1
i=1	4	3	2	1	5
i=2	3	2	1	4	5
i=3	2	1	3	4	5
	1	2	3	4	5

Bubble Sort (pre-sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=0	1	2	3	4	5	
i=1	1	2	3	4	5	
i=2	1	2	3	4	5	
i=3	1	2	3	4	5	
	1	2	3	4	5	

Sorting Algorithms

- Bubble Sort
  - Best-case complexity: $O(n^2)$
  - Worst-case complexity: $O(n^2)$
- Insertion Sort
  - Best-case complexity: O(?)
  - Worst-case complexity: O(?)

Insertion Sort

```
void insertion_sort(int a[], int n){
    int i,j;

    for(i=1;i<n;i++)
        for(j=i;j>0;j--)
            if (a[j]<a[j-1])
                swap(&a[j-1],&a[j]);
            else
                break;
}
```

Insertion Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=1	5	4	3	2	1	
	4	5	3	2	1	j=1

Insertion Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=1	5	4	3	2	1	
i=2	4	5	3	2	1	
	4	3	5	2	1	j=2
	3	4	5	2	1	j=1

Insertion Sort (inversely sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=1	5	4	3	2	1	
i=2	4	5	3	2	1	
i=3	3	4	5	2	1	
i=4	2	3	4	5	1	
	1	2	3	4	5	

Insertion Sort (pre-sorted input)

	a[0]	a[1]	a[2]	a[3]	a[4]	n=5
i=1	1	2	3	4	5	
i=2	1	2	3	4	5	
i=3	1	2	3	4	5	
i=4	1	2	3	4	5	
	1	2	3	4	5	

Sorting Algorithms

- Bubble Sort
  - Best-case complexity: $O(n^2)$
  - Worst-case complexity: $O(n^2)$
- Insertion Sort
  - Best-case complexity: $O(n)$
  - Worst-case complexity: $O(n^2)$

Tree-based Algorithms

- BST Operations

  Search/Insert:

  – Worst-case complexity: O(?)
- AVL Operations

  Search/Insert

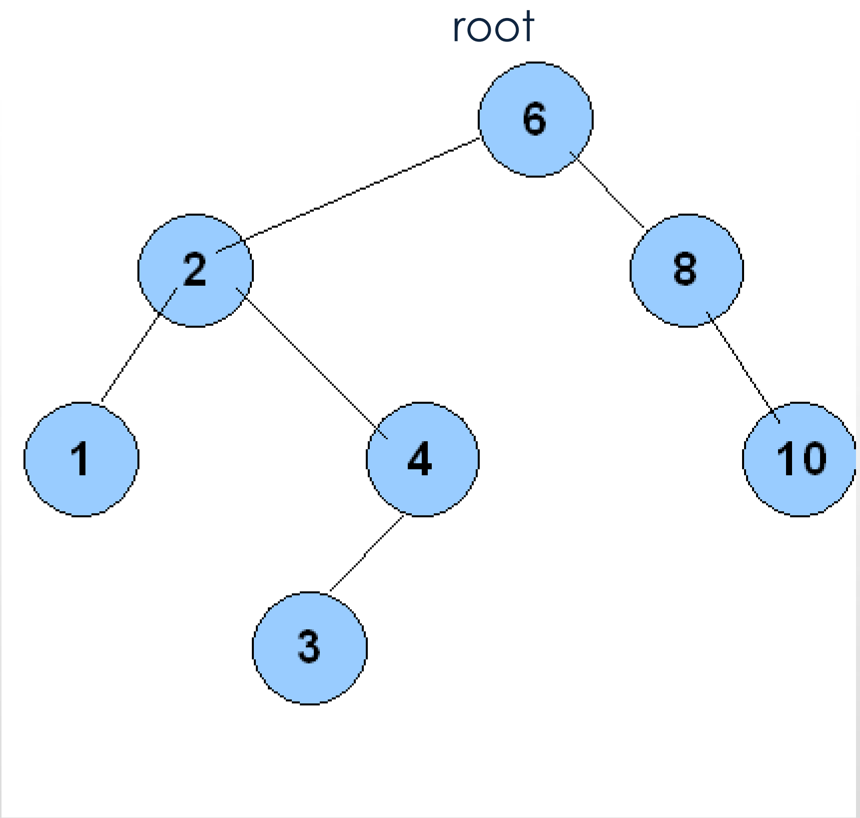
  – Worst-case complexity: O(?)

Search for a key

```
typedef struct node{
   int value;
   struct node *left;
   struct node *right;
   struct node *parent;
}BST;
```

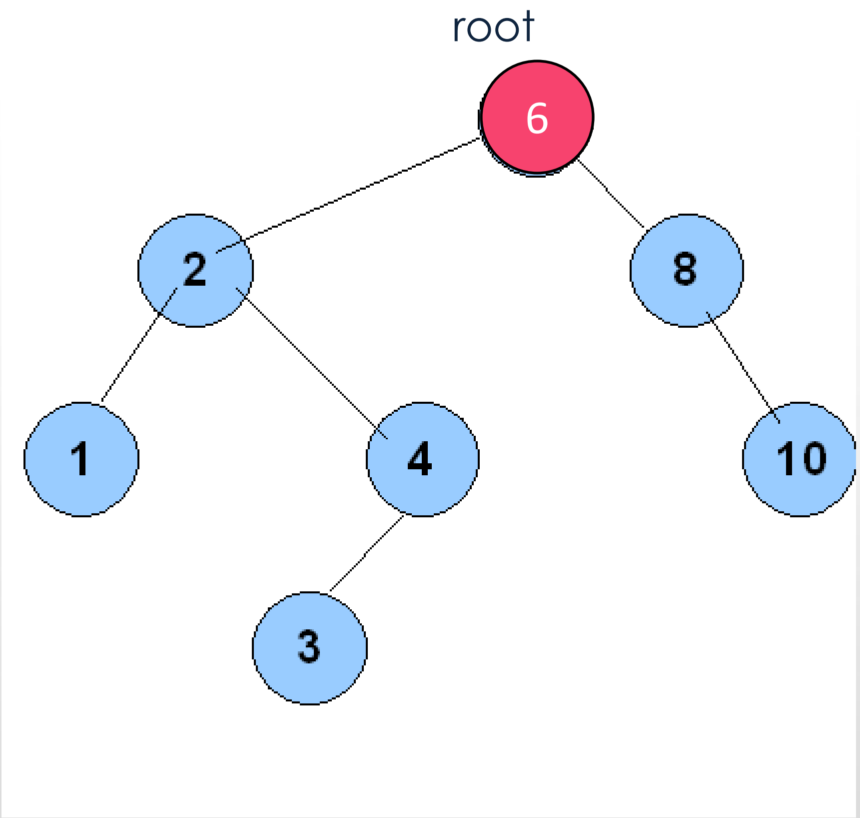
```
BST *search(BST *root, int x){
 BST *temp=root;

 while((temp!=NULL)&&(temp->value!=x)){
   if(x < temp->value)
     temp = temp->left;
   else
     temp = temp->right;
 }
 return temp;
}
```

Search for a key



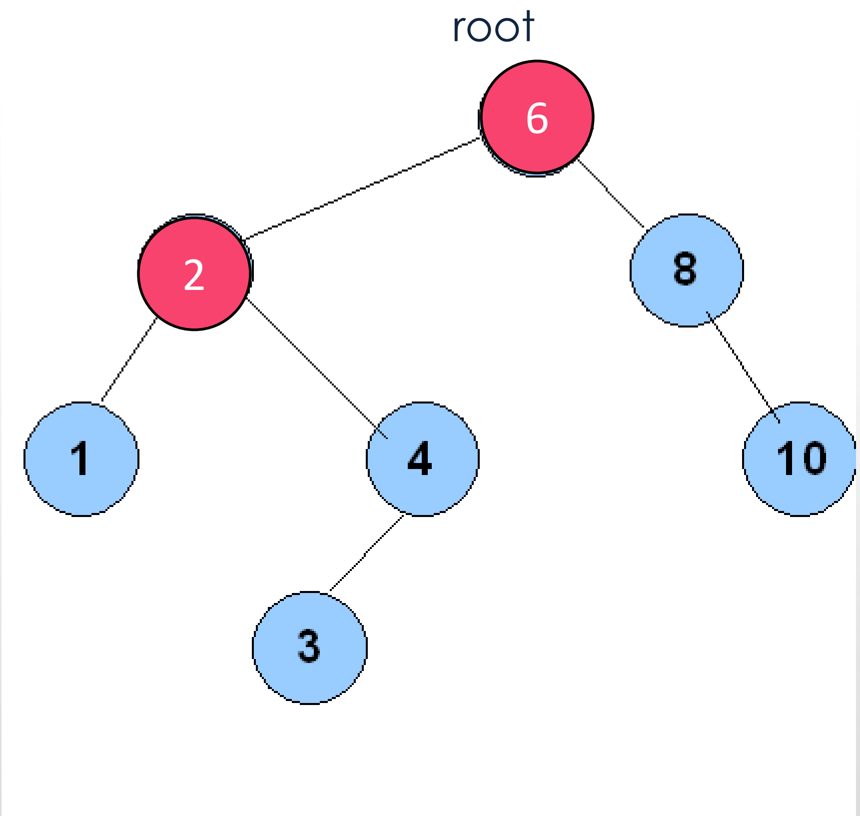
ptr = search(root, 3);

Search for a key



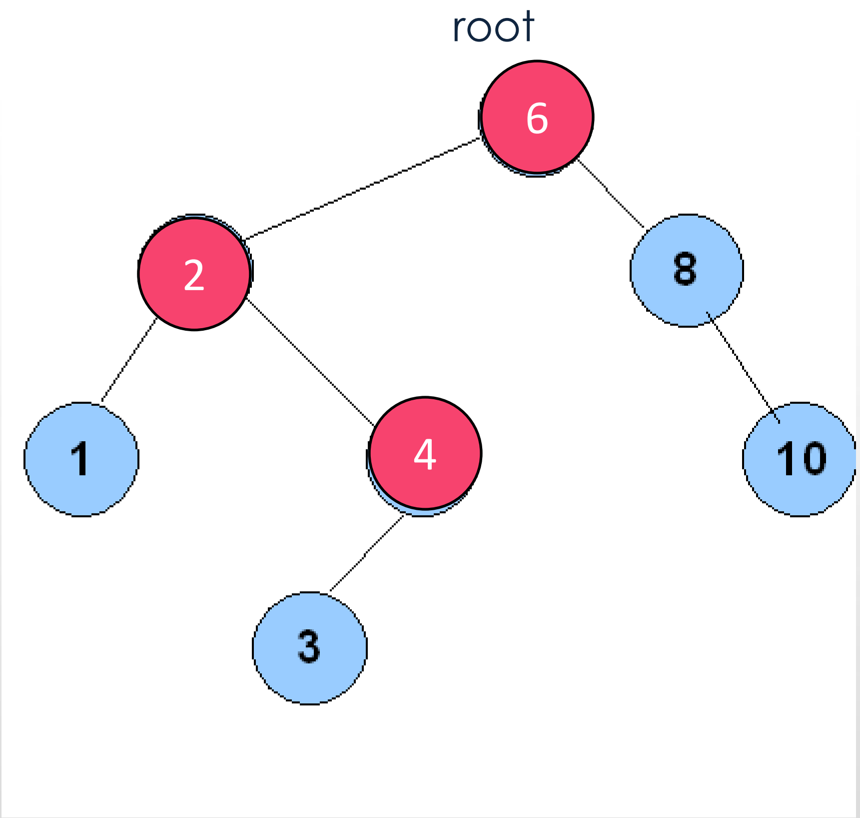
ptr = search(root, 3);

Search for a key



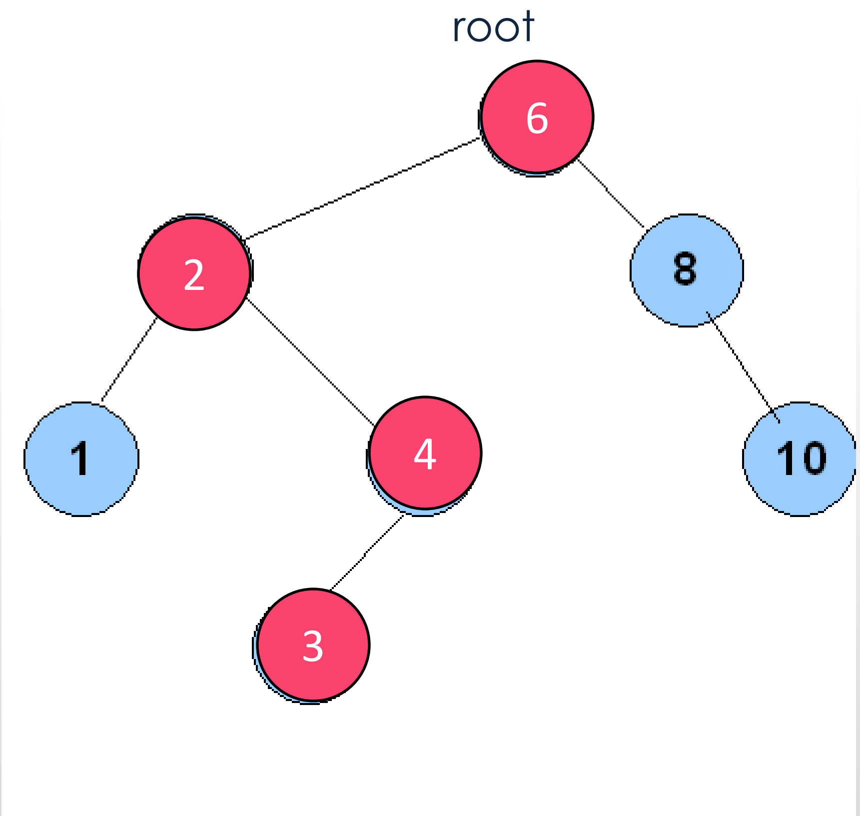
ptr = search(root, 3);

Search for a key



ptr = search(root, 3);

Search for a key



ptr = search(root, 3);

Insertion

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;

int main(){
  BST *root=NULL;

}
```

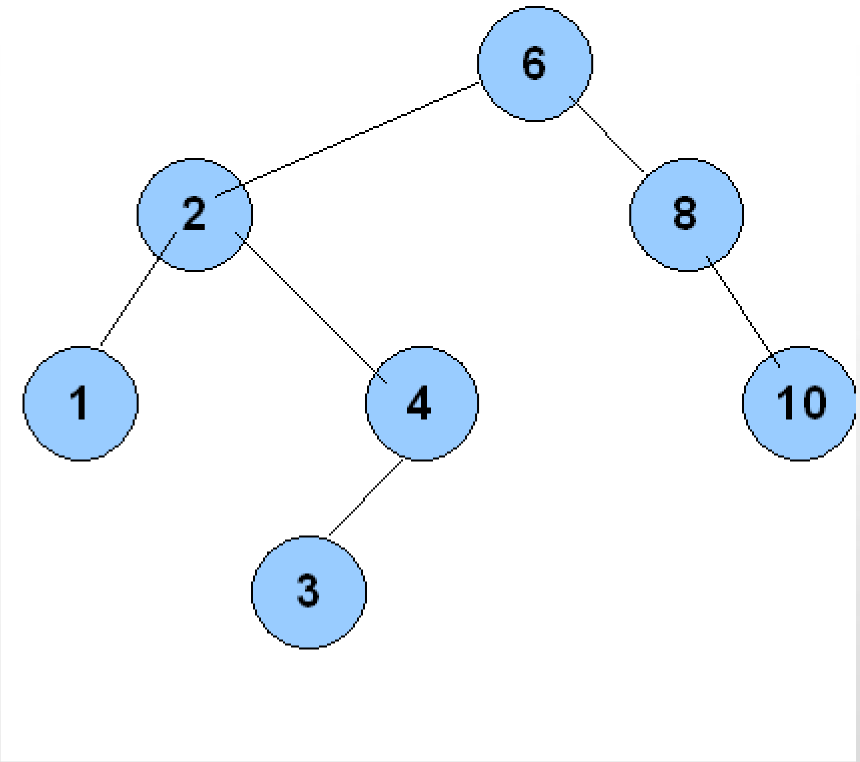
```
void insert(BST *root, int x){
  BST *temp;

  temp=(BST *)malloc(sizeof(BST));
  if(temp==NULL){
    printf("Insufficient Memory");
    exit(1);
  }
  temp->value=x;
  temp->left=NULL;
  temp->right=NULL;
  temp->parent=NULL;

  insert2(root, temp);
}
```

Insertion

```
typedef struct node{
  int value;
  struct node *left;
  struct node *right;
  struct node *parent;
}BST;

int main(){
  BST *root=NULL;

}
```

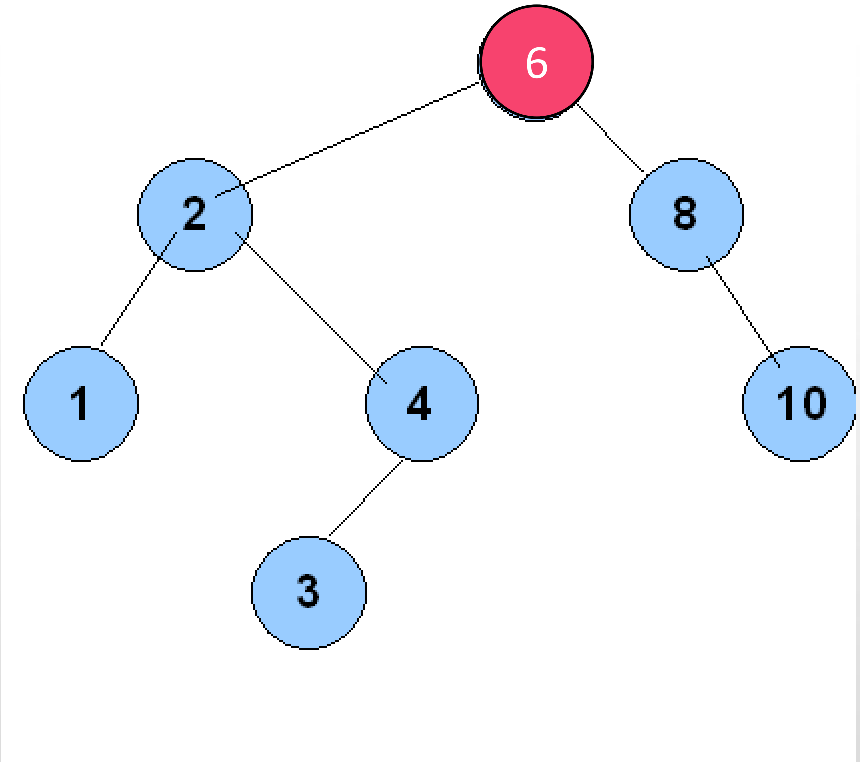
```
void insert2(BST *root, BST *temp){

 if(root==NULL)
   root=temp;
 else{
  temp->parent=root;
  if ((root)->value > temp->value)
   insert2(root->left,temp);
  else
   insert2(root->right,temp);
 }
}
```

Insertion



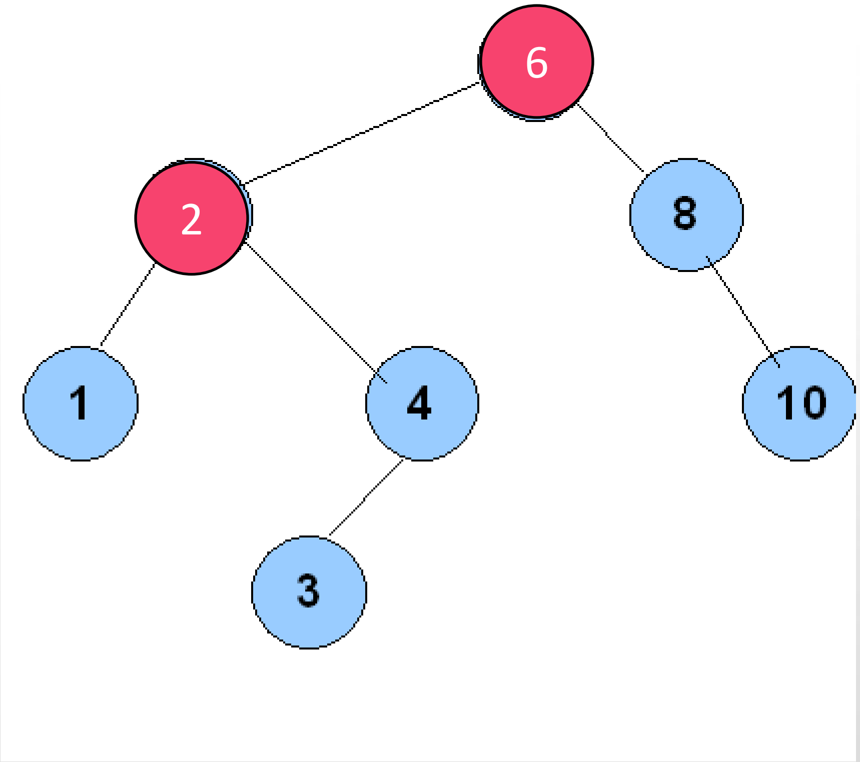
ptr = insert(root, 5);

Insertion



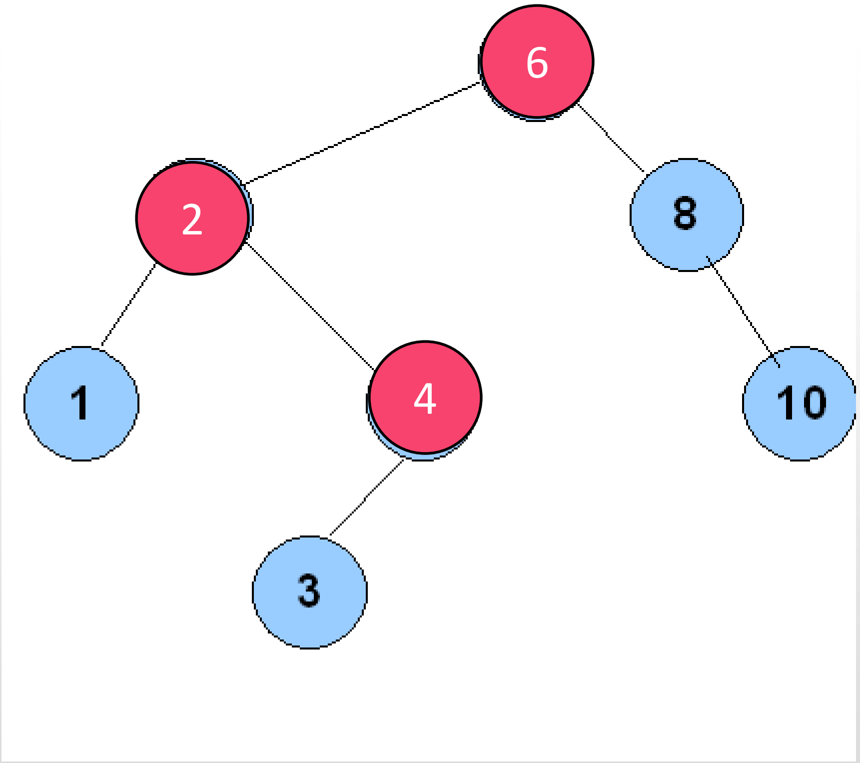
ptr = insert(root, 5);

Insertion



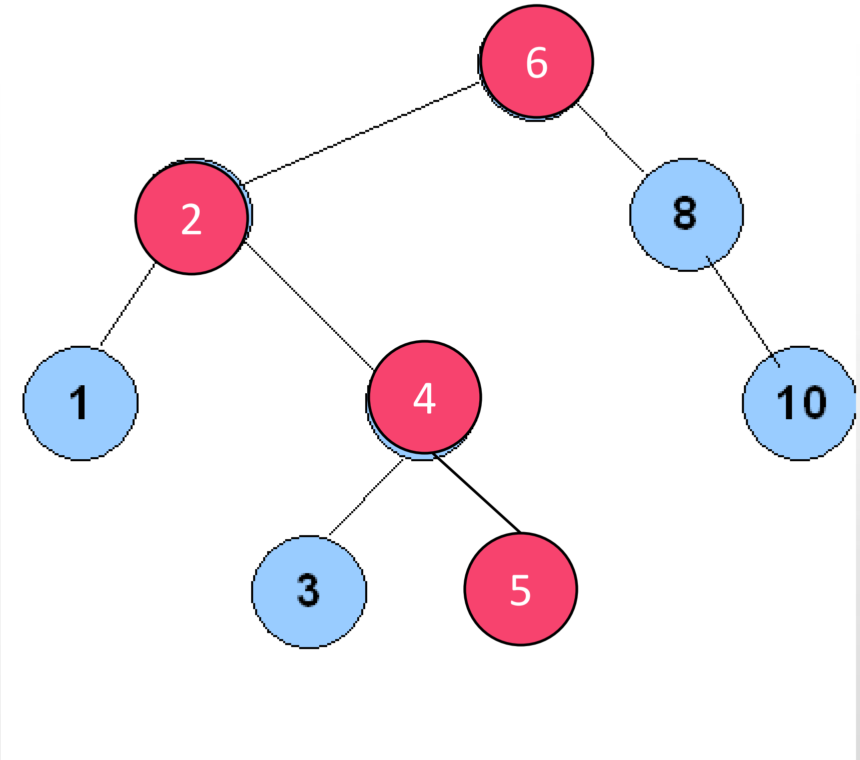
ptr = insert(root, 5);

Insertion



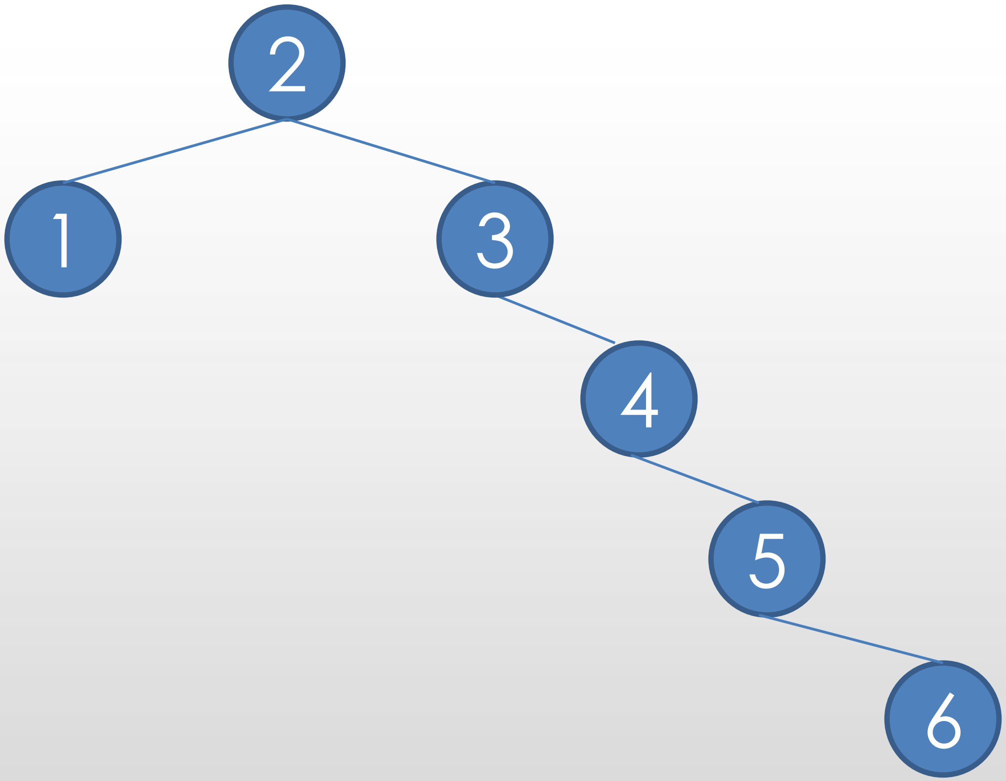
ptr = insert(root, 5);

Insertion



ptr = insert(root, 5);

BST - drawback



Tree-based Algorithms

- BST Operations

  Search/Insert:

  – Worst-case complexity: O(n)
- AVL Operations

  Search/Insert

  – Worst-case complexity: O(log n)

Design technique

- Exponentiation

  Obvious Algorithm:

  – compute x^n using n-1 multiples

  Recursive Algorithm:

  – less number of multiplications

Design technique

- Exponentiation

```
int pow(int x, int n){

    if(n==0) return 1;
    if(n==1) return x;
    if(even(n))
        return (pow(x*x,n/2));
    else
        return (pow(x*x,n/2)*x);
}
```

Design technique

```
int pow(int x, int n){

    if(n==0) return 1;
    if(n==1) return x;
    if(even(n))
        return (pow(x*x,n/2));
    else
        return (pow(x*x,n/2)*x);
}
```

- $2^9 = (4^4)*2$
- $4^4 = (16^2)$
- $16^2 = (256^1)$
- $256^1 = 256$