ECE36800 Data structures

Recursion (call stack and space complexity)

Chapter 5 (pp. 187-207)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

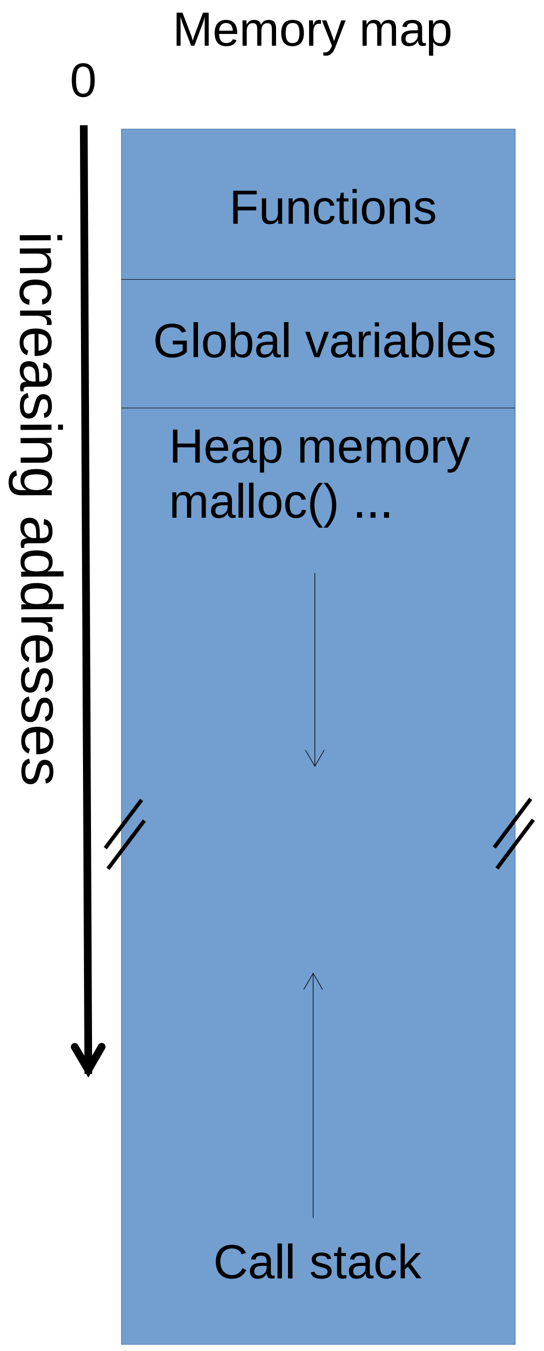
Overview

- Function calls and call stack
- Recursion
  - How does it work?
- Recursion vs. iteration
  - Tail recursion removal
  - Removing all recursive calls

- Reference pages: pp. 187-207

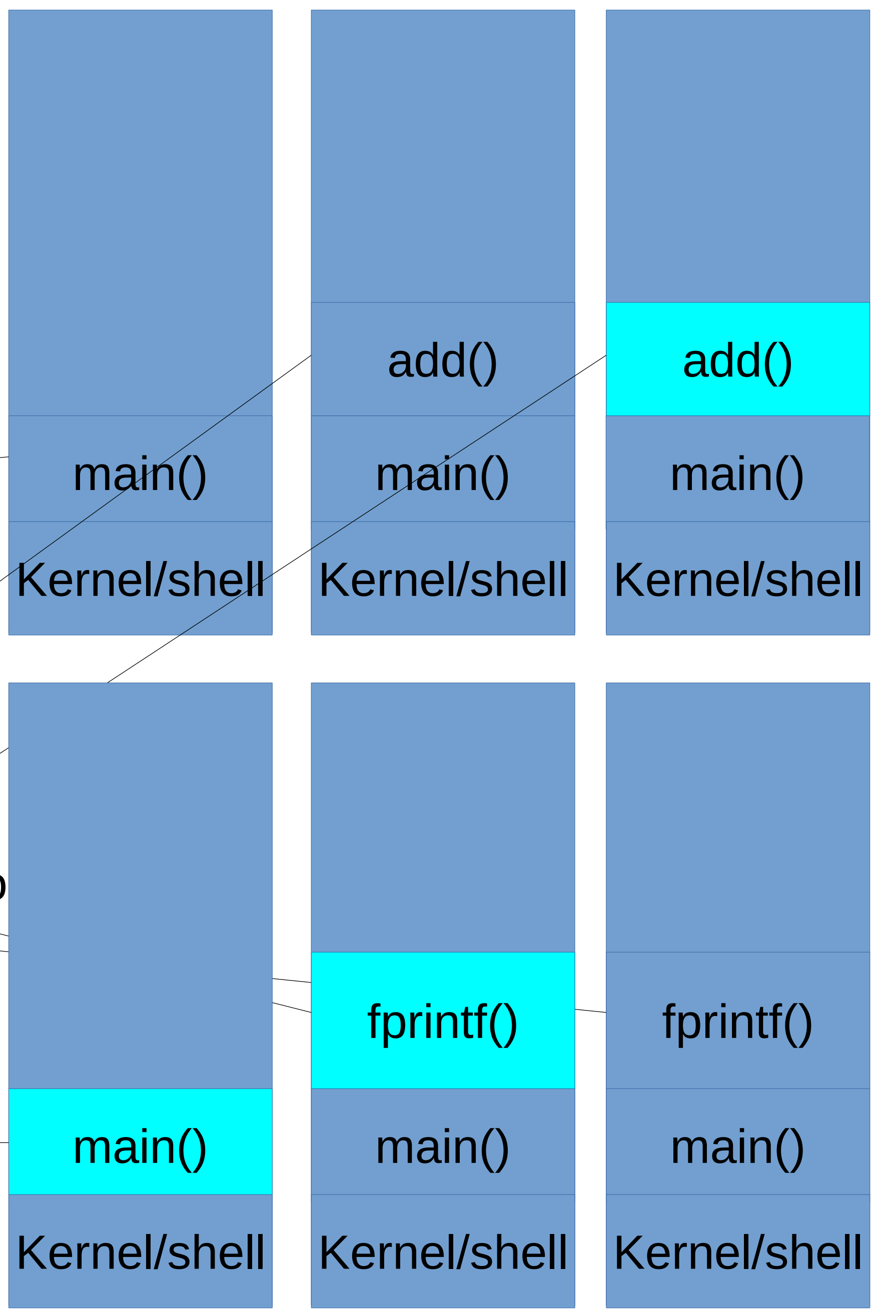
Call stack

- Every time you invoke a function, the invoked function takes some space, called a stack frame, on the call stack
- When you call a function, you push a stack frame for the function on the call stack
- When the function is completed, its corresponding stack frame Is popped from the call stack
- The stack frame includes
  - Function parameters
  - Local variables
  - Return address of the caller function about where to return to at the completion of the invoked function
  - Frame pointer of the caller function (so that you can pop the current frame when it is completed)
- Size of a stack frame depends on the number of function parameters and the number of local variables



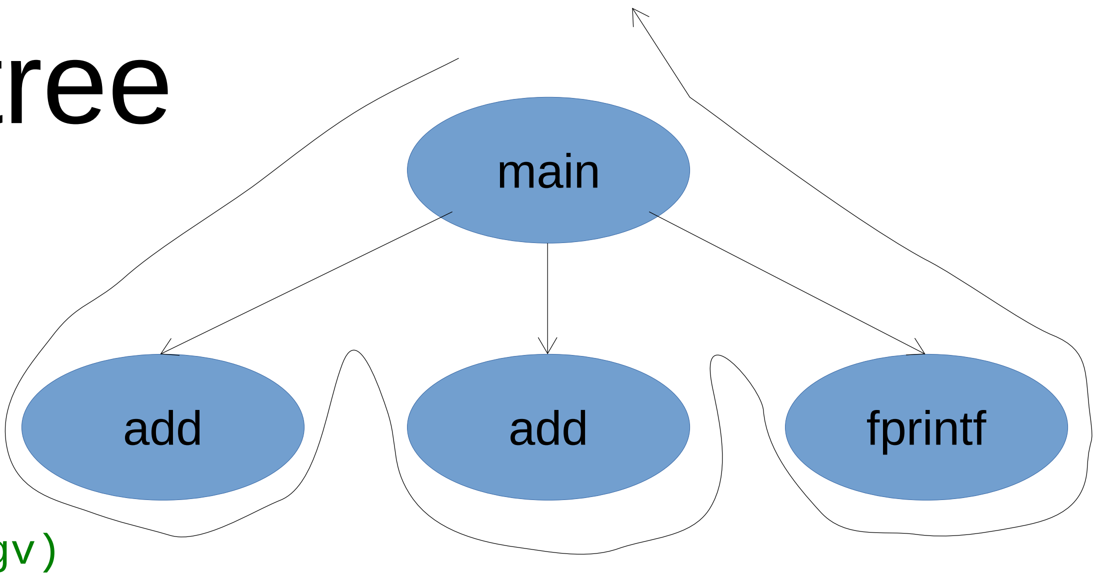
Example

```
int add(int i, int j)
{
   return i+j;
}

int main(int argc, char **argv)
{
   int i, j;

   …

   fprintf(stdout, “%d\n”, add(i, j));

   …

   return EXIT_SUCCESS;
}
```



Computation tree

```
int add(int i, int j)
{
   return i+j;
}

int main(int argc, char **argv)
{
   int i, j;

   …

   fprintf(stdout,
           “%d\n”,
           add(add(i,j),j));

   …

   return EXIT_SUCCESS;
}
```



- Traversal of the computation tree shows the order in which functions are executed
  - When main is executed, and it calls add
  - When inner-most add is completely executed, we return to main
  - main then calls outer add
  - When outer add is completely executed, we return to main
  - main then calls fprintf
  - When fprintf is completely executed, we return to main
- At any node, tracing the path to the root node (main) shows all valid stack frames on the call stack

Recursion

- A computation technique of determining a result by calling itself
- It is a divide-and-conquer technique:
  - It decomposes a big problem into smaller problems
  - Call itself on these smaller problems
  - Combine the solutions to these smaller problems to form the solution to the big problem
- Cannot keep calling itself on smaller problems
  - When a problem is small (simple) enough, solve the problem directly to obtain the solution
  - This is called the base case or the terminating or stopping condition

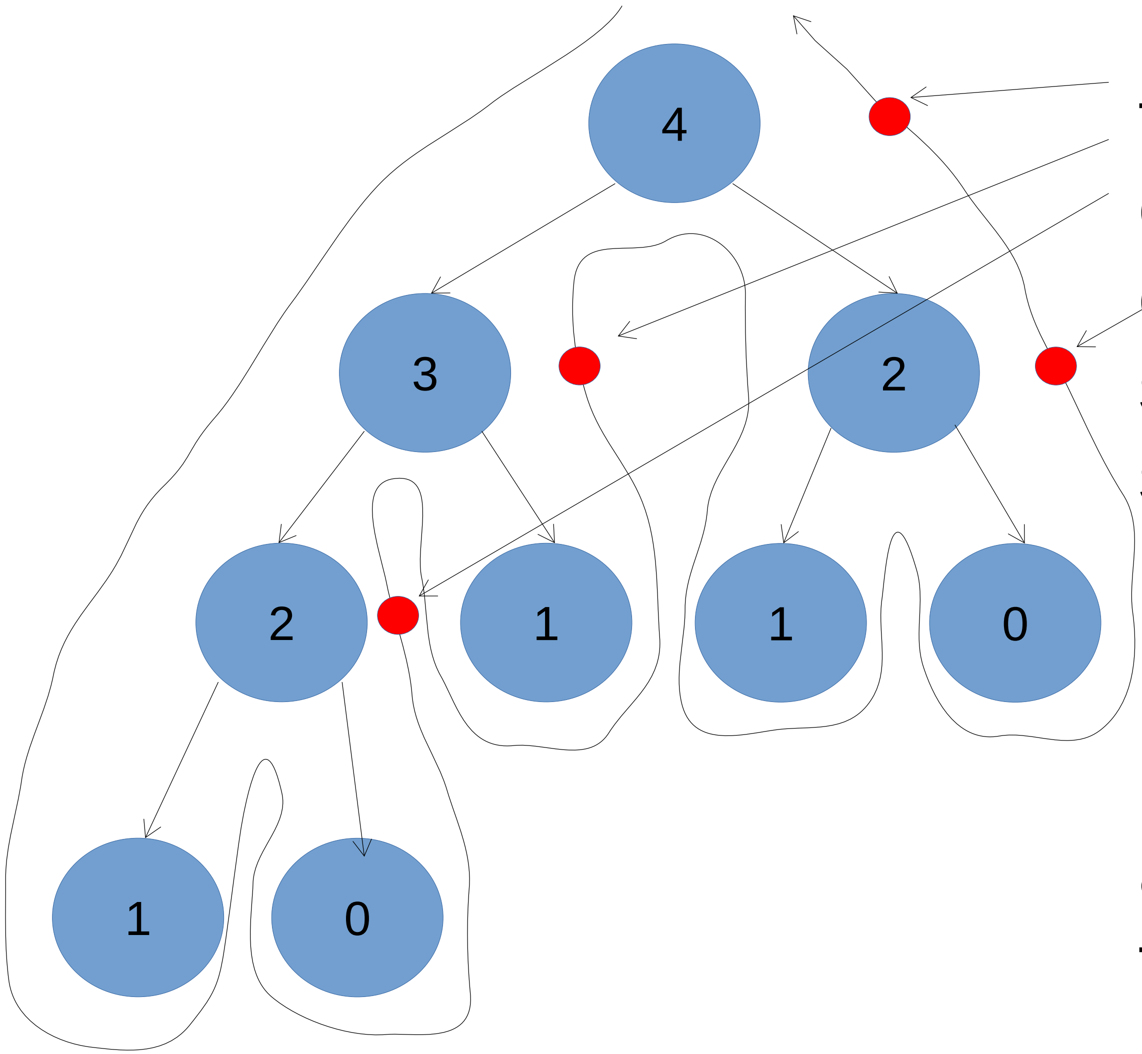
Example: Fibonacci sequence

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, …

```
unsigned int Fibonacci(unsigned int n)
{
  if ((n == 0) || (n == 1))
    return n;
  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

- Base case or stopping condition: when n == 0 or n == 1, we have the Fibonacci number directly
- Divide into two smaller problems of n – 1 and n – 2 and make recursive calls on them
- After you have the solutions of these smaller problems, combine them Fibonacci(n-1) + Fibonacci(n-2) to obtain the solution of the big (original) problem

Computation tree of calling Fibonacci(4)



- This is where we perform the most meaningful computation in this tree: combining the solutions to smaller problems to form a solution of a bigger problem
- Postorder traversal: Computation is performed after all recursive calls in a function

Time complexity of Fibnacci(n)

- Each function call executes a fixed number of statement, i.e. O(1) statements executed in each function call
- Time complexity is therefore the number of function calls multiplied by O(1) statements per function call
- The number of function calls can be obtained by counting the number of nodes in the computation tree
- Assume that the computation tree is a full binary tree of a height n, there are $O(2^n)$ function calls (nodes)
- Time complexity is therefore $O(2^n)$

Space complexity of Fibonacci(n)

- Each function call pushes a stack frame on the call stack
- At most, there will be n valid stack frames on the call stack simultaneously
  - From bottom to up in the call stack, the left branch of the computation tree corresponds to the stack frames of Fibonacci(n), Fibonacci(n-1), …, Fibonacci(1)
- The amount of additional space required to perform computation is call space complexity
- The space complexity of (recursive) Fibonacci(n) is O(n) since in the worst case, there are n stack frames

Example: Building a binary search tree

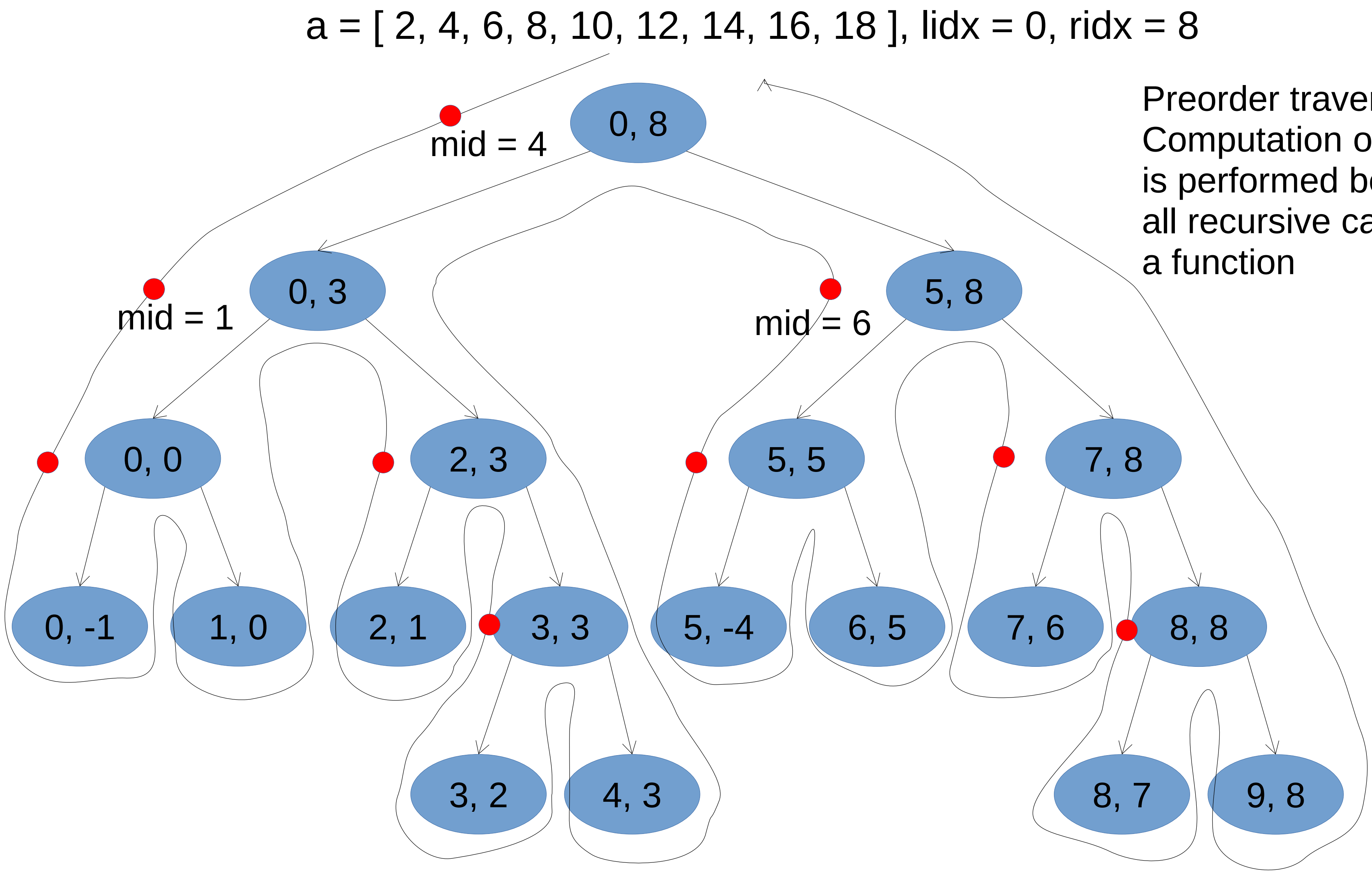
- Given an sorted array a[0..n-1] of n integers, construct a binary search tree where Tnode is a struct to store the integer in data field, left and right are addresses to point at Tnode as left and right child nodes
- lidx is the left most index of array and ridx is the right most index of array

```
Tnode *Build_BST(int *a, int lidx, int ridx)
{
  if (lidx > ridx)
    return NULL;
  int mid = (lidx + ridx)/2;
  Tnode *root = malloc(sizeof(*root));
  if (root != NULL) {
    root->data = a[mid];
    root->left = Build_BST(a, lidx, mid-1);
    root->right = Build_BST(a, mid+1, ridx);
  }
  return root;
}
```

Constructing binary search tree

- Base case: an empty array (lidx > ridx) == empty tree == NULL
- Decide where to divide the array: mid = (lidx + ridx)/2 (may cause overflow, safer to use mid = lidx + (ridx – lidx)/2)
- Divide the array into two halves: [lidx, mid-1] and [mid+1, ridx]
  - Apply recursion to the two halves
  - Ideal form of divide-and-conquer

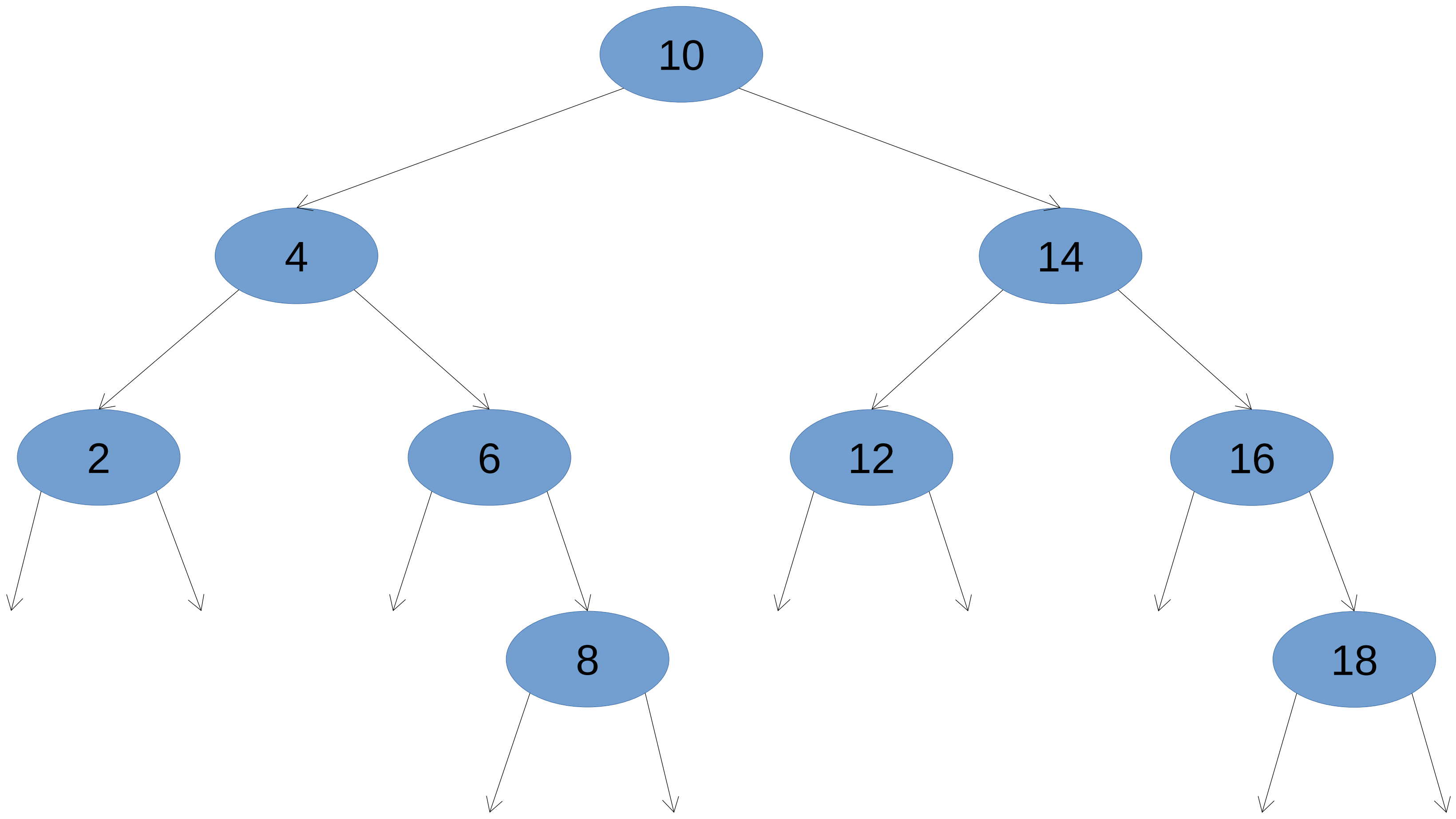
Computation tree



Preorder traversal: Computation of mid is performed before all recursive calls in a function

Binary search tree

a = [2, 4, 6, 8, 10, 12, 14, 16, 18], lidx = 0, ridx = 8



Time complexity

- Each function executes O(1) instructions
- If there are n integers in the array, there are 2n + 1 function calls
  - The computation tree is a strictly binary tree, each node either has 2 child nodes or no children
  - There are n non-leaf nodes in the computation tree, and n+1 leaf nodes
- Time complexity is therefore O(n)

Space complexity

- Do not have to account for the space for the binary search tree since we have to build one
- Must account for additional space required for the computation
- Recursive calls take up space on the call stack
- Each recursive call divides the problem evenly into two halves
  - As we traverse from the root node of the computation tree to the leaf node, the array size decreases from n $\rightarrow$ n/2 $\rightarrow$ n/4 $\rightarrow$ ... $\rightarrow$ 1
  - It takes log n calls to go from n to 1
  - There are at most log n stack frames on the call stake
- The space complexity is O(log n)

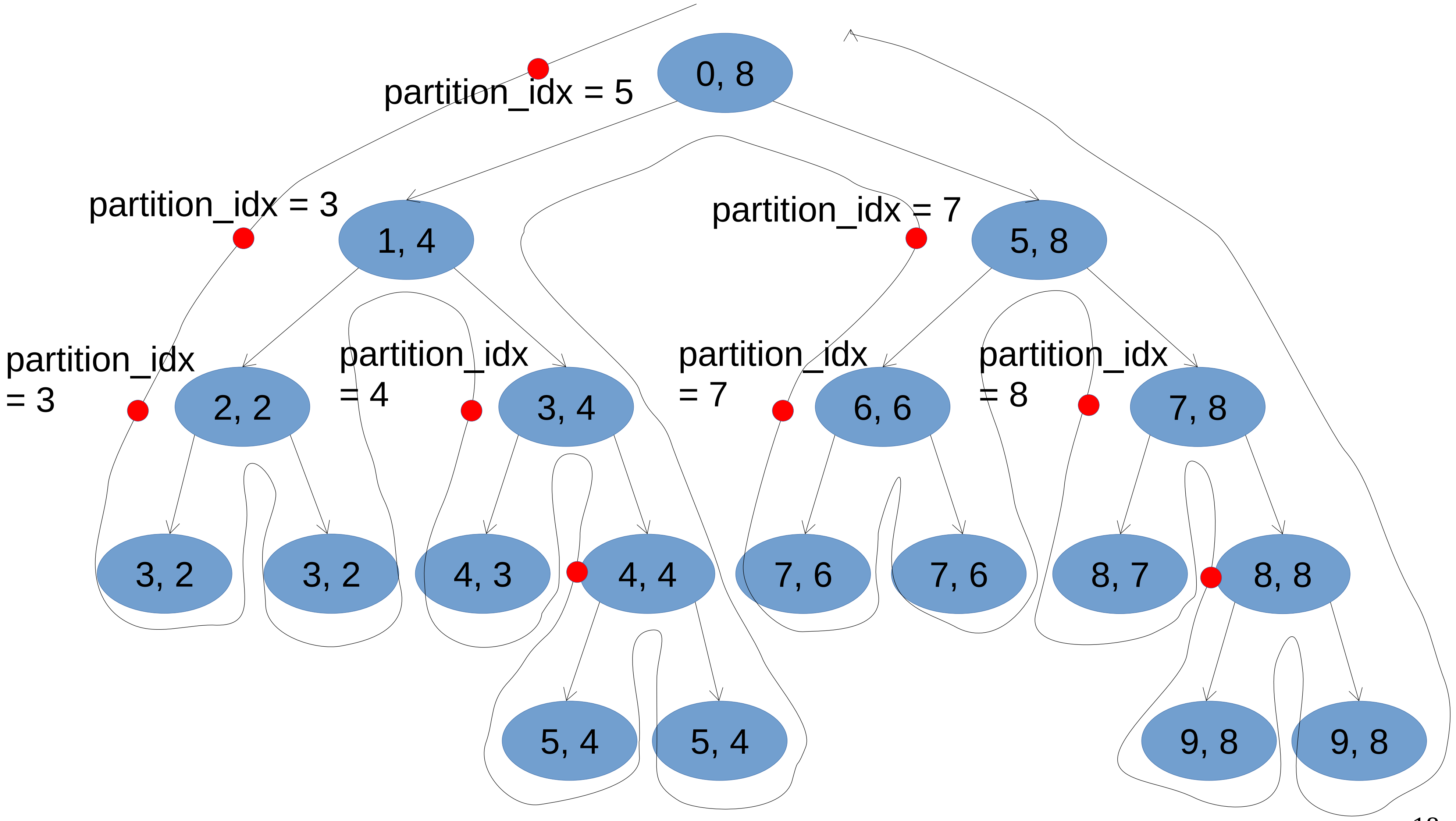
Re-building from pre-order

Given an array of integers obtained from the pre-order traversal of a binary search tree of distinct integers, rebuild the BST (not the most efficient, just to demonstrate time complexity and space complexity

```
Tnode *Preorder_rebuild_BST(int *a, int lidx, int ridx)
{
  if (lidx > ridx)
    return NULL;
  Tnode *root = malloc(sizeof(*root));
  if (root != NULL) {
    root->data = a[lidx];
    int partition_idx = lidx + 1;
    while (partition_idx <= ridx &&
           a[partition_idx] < a[lidx])
      partition_idx++;
    root->left = Preorder_rebuild_BST(a, lidx+1,
                                      partition_idx-1);
    root->right = Preorder_rebuild_BST(a, partition_idx,
                                       ridx);
  }
  return root;
}
```

Rebuild binary search tree

A = [10, 4, 2, 6, 8, 14, 12, 16, 18], lidx = 0, ridx = 8



Time complexity

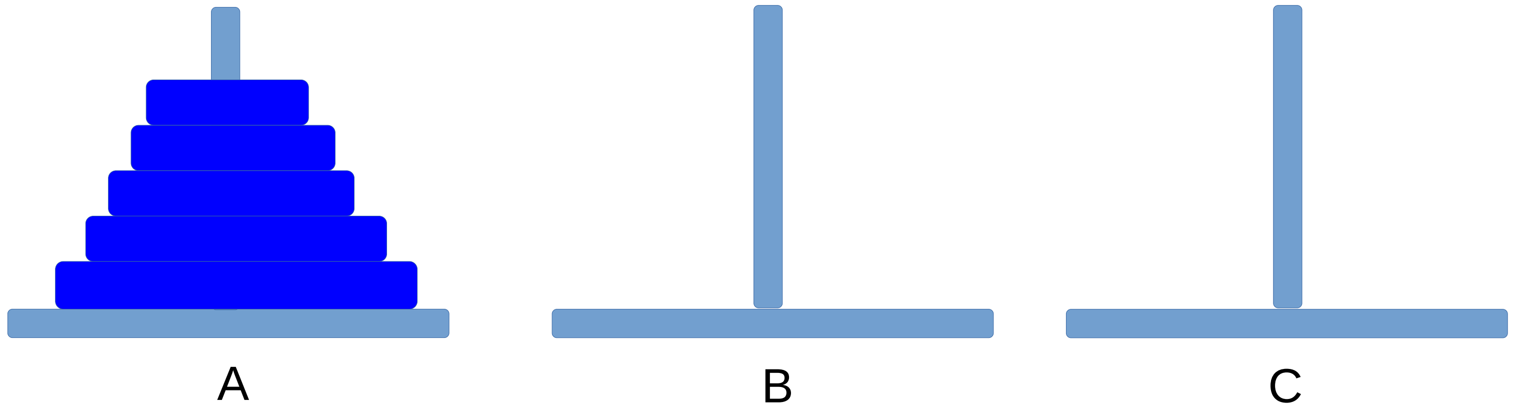
- Assume that there are n elements in the preorder array
- Best-case scenario, execute the while statement once and get out
  - O(1) statements executed in the function
  - Left recursive call has 0 elements, right recursive call has n – 1 elements
  - When the preorder array is in ascending order, the computation tree is skewed to the right, and overall time complexity is O(n) because there are n nodes in the computation tree and each has O(1) time complexity
  - The BST is in a sense a linked list formed by the right branches
- Worst-case scenario, execute the while statement O(n) times
  - O(n) statements executed in the function
  - Left recursive call has n – 1 elements, right recursive call has 0 elements
  - When the preorder array is in descending order, the computation tree is skewed to the left, and overall time complexity is dominated by the left recursive calls, with time complexity of $O(n + (n – 1) + … + 1) = O(n^2)$
  - The BST is in a sense a linked list formed by the left branches

Space complexity

- Worst-case scenario is when the computation tree or the BST has a height of O(n) (see the best and worst-case scenarios for time complexity, for example)
  - O(n) space complexity
- When the computation tree or BST is balanced, a height of O(log n)
  - O(log n) space complexity

Example: Tower of Hanoi

- 3 rods (A, B, C) and n disks of difference sizes that can slide onto any rod
- All disks are on rod A in ascending order of size, with the smaller disk at the top
- Goal is to move all disks to rod C, one disk at a time
- Disks may be placed on all rods, with the restriction that a larger disk cannot be placed on top of a smaller disk



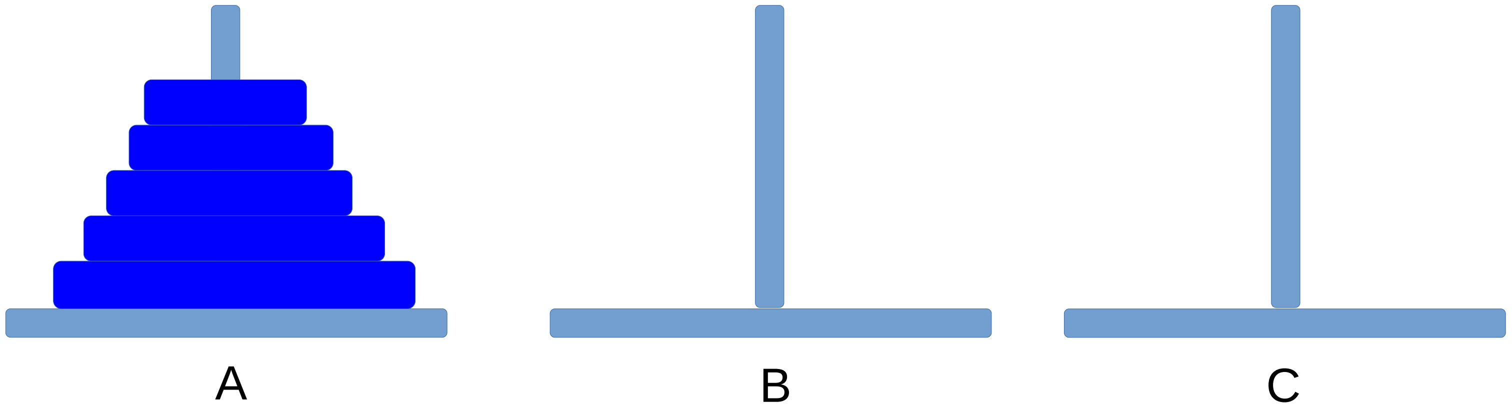
Example: Tower of Hanoi

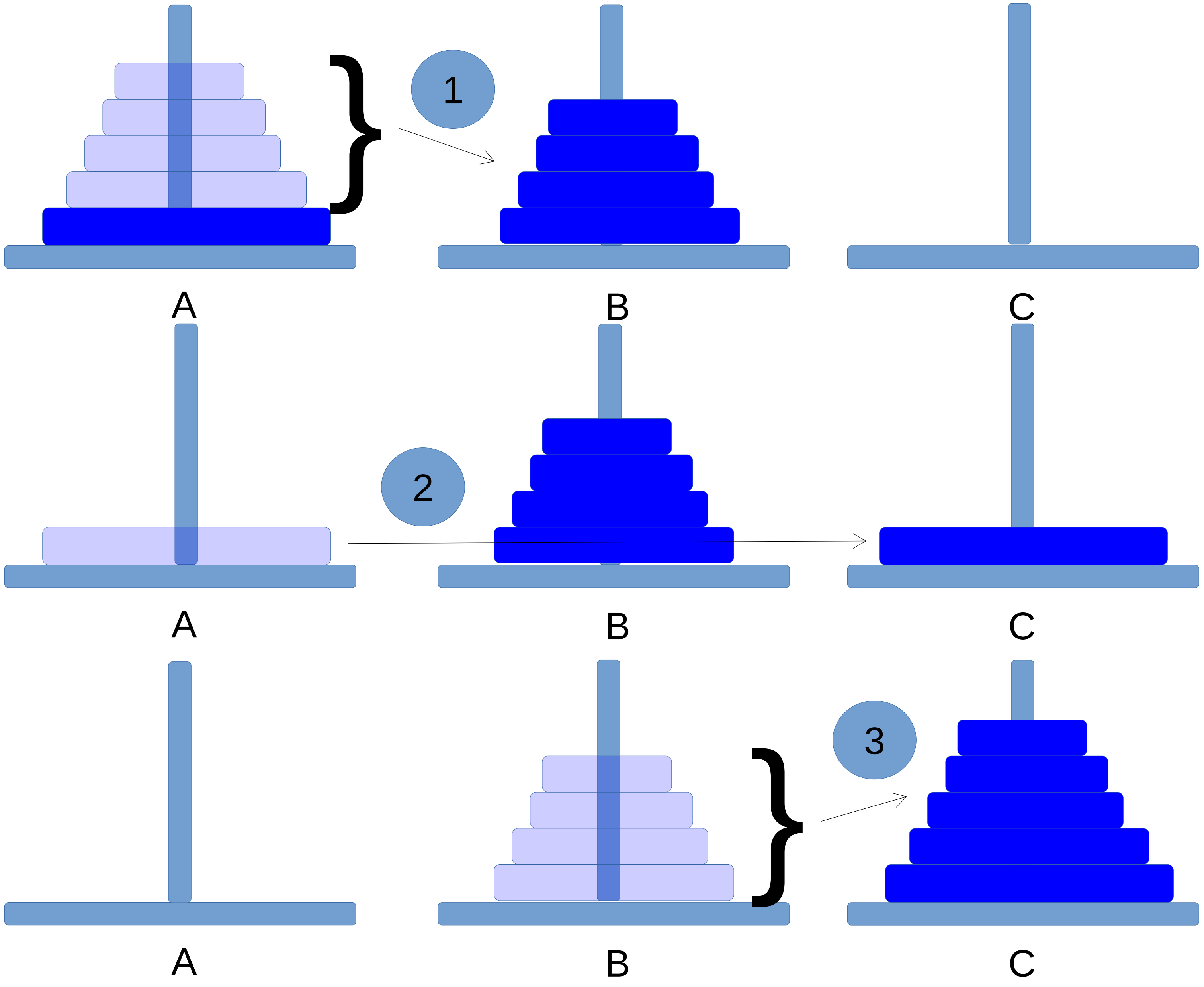
- Base case: if n is 1, move from A to C directly
- Divide-and-conquer: Assume that we know how to move n – 1 disks from a source rod to a destination rod using the remaining rod as an intermediate rod

1) Move the top n – 1 disks from A to B using C as an intermediate rod

2) Move the n-th disk from A to C

3) Move the n – 1 disks from B to C using A as an intermediate rod



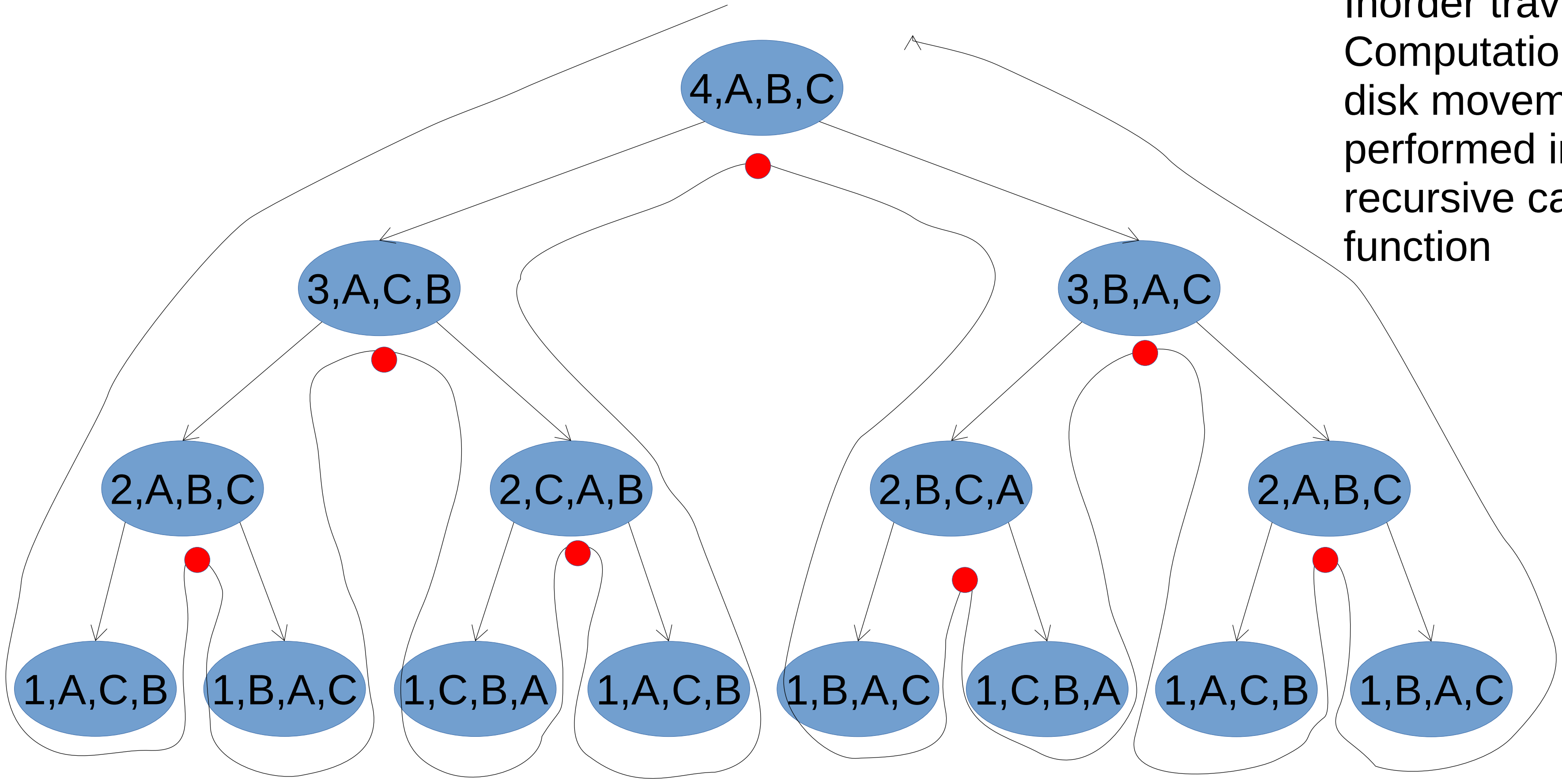

1
A
B
C
2
A
B
C
3
A
B
C

Example: Tower of Hanoi

- Move n disks from src rod to dest rod, using intm rod for intermediate storage, assume that n ≥ 1

```
void ToH(int n, char src, char intm, char dest)
{
  if (n == 1) {
    fprintf(stdout, “move disk %d from %c to %c\n”,
            n, src, dest);
    return;
  }
  ToH(n-1, src, dest, intm); // step 1
  fprintf(stdout, “move di sk %d from %c to %c\n”,
          n, src, dest);     // step 2
  ToH(n-1, intm, src, dest); // step 3
}
```

Computation tree: ToH(4, 'A', 'B', 'C')



Inorder traversal: Computation of the disk movement is performed in between recursive calls in a function

```
move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C
move disk 3 from A to B
move disk 1 from C to A
```

```
move disk 2 from C to B
move disk 1 from A to B
```
move disk 4 from A to C
```
move disk 1 from B to C
move disk 2 from B to A
```

```
move disk 1 from C to A
move disk 3 from B to C
move disk 1 from A to B
move disk 2 from A to C
move disk 1 from B to C
```

Time complexity and space complexity

- There are $2^n – 1$ function calls
  - Each function executes O(1) instructions
  - Time complexity is therefore $O(2^n)$
- The height of computation tree is O(n)
  - Space complexity is therefore O(n)

Example: Binary search

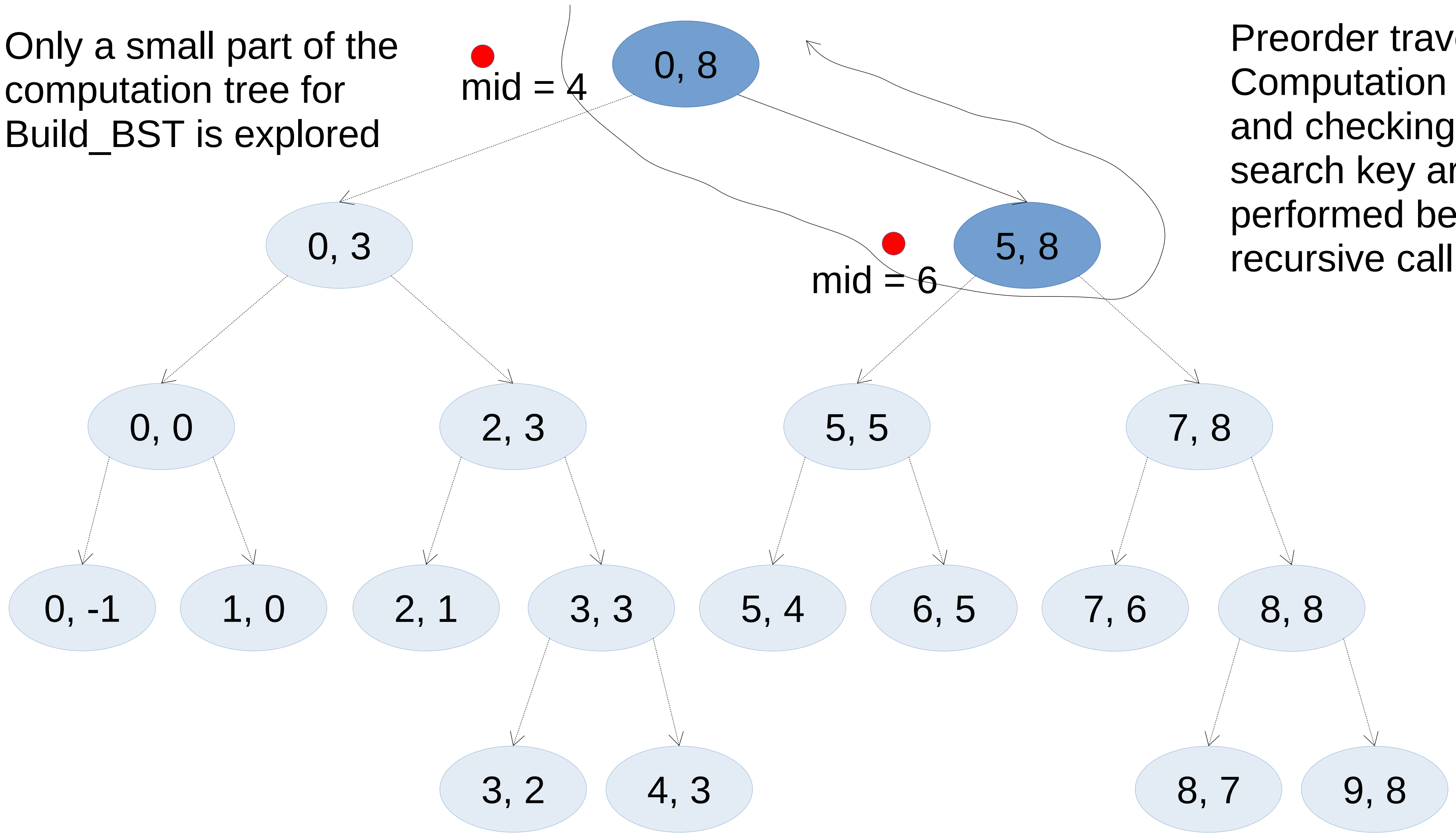
- Perform binary search on a sorted array of integers a[0, n-1]

```
int Binary_search(int *a, int lidx, int ridx, int key)
{
  if (lidx > ridx) // can't find key in an empty array
    return -1;
  int mid = (lidx + ridx)/2;
  if (a[mid] == key)
    return mid;
  if (a[mid] > key) // key may be in the left half
    return Binary_search(a, lidx, mid-1, key);
  else              // key may be in the right half
    return Binary_search(a, mid+1, ridx, key);
}
```

Computation tree

a = [2, 4, 6, 8, 10, 12, 14, 16, 18], lidx = 0, ridx = 8, key = 14

Only a small part of the computation tree for Build_BST is explored



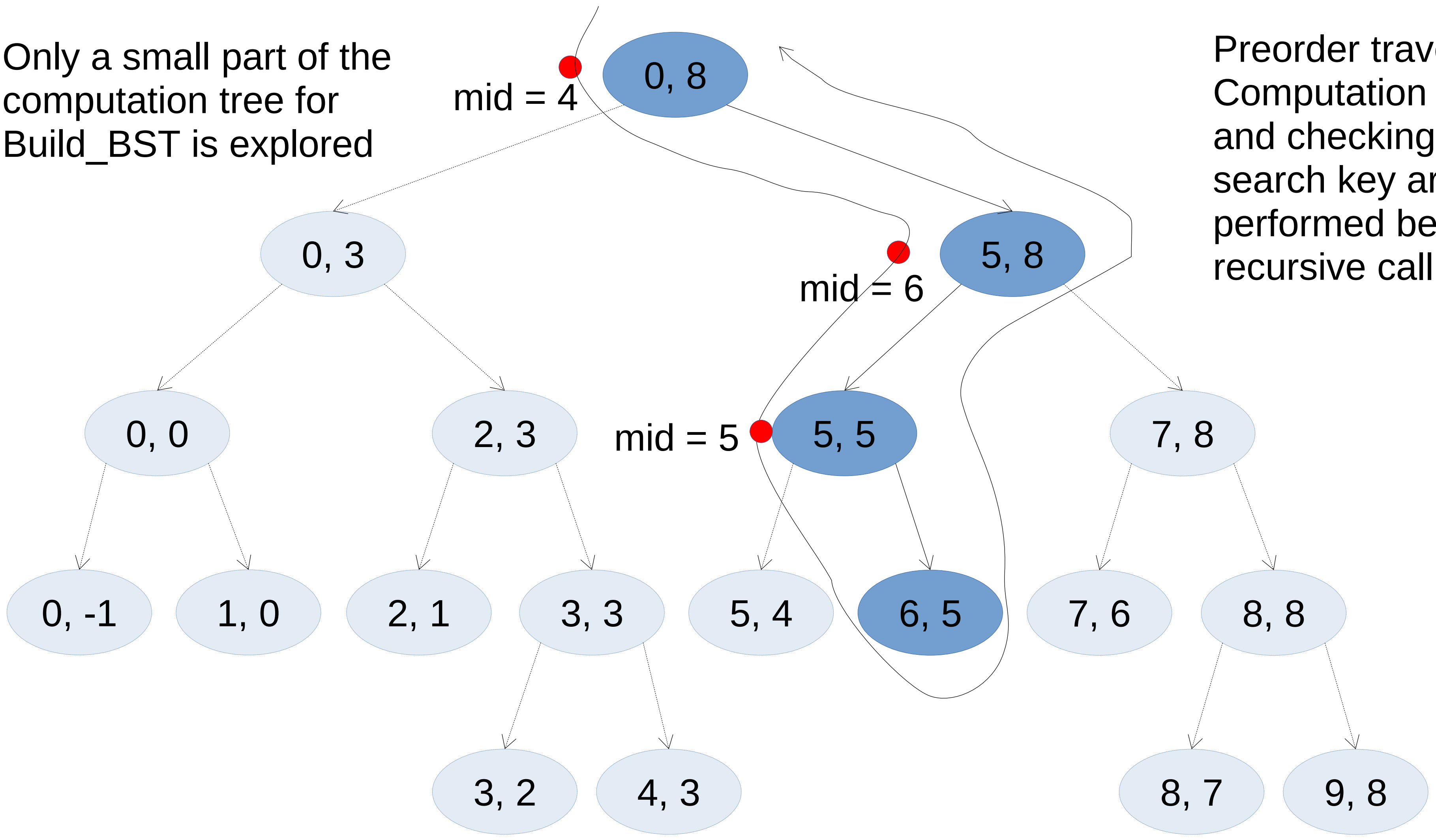
Preorder traversal: Computation of mid and checking for search key are performed before a recursive call

Computation tree

a = [2, 4, 6, 8, 10, 12, 14, 16, 18], lidx = 0, ridx = 8, key = 13

Only a small part of the computation tree for Build_BST is explored

Preorder traversal: Computation of mid and checking for search key are performed before a recursive call



Time complexity and space complexity

- Each function executes O(1) instructions
- Each recursive call halves the size of array
  - In the worst case, it takes ~ log n calls to reach the bottom of the tree
  - In the best case, we find the search key early
- Best case time complexity is O(1) and worst case time complexity is O(log n)
- Best case space complexity is O(1) and worst case space complexity is O(log n)

Order of traversals

- Think of preorder, inorder, or postorder traversal as the order in which information is passed around for computation
  - If the parent node has to pass information to all child nodes, you would use preorder traversal
- Try out whether the food is of the right texture before you feed your toddler
  - If all child nodes have to pass information to a parent node, you would use postorder traversal
- Clean out after your children have made a mess
- Knowing the correct traversal order would make it easier to write a recursive function
- You may need a combination of two or more traversal orders to perform a task

Recursion vs. iteration

- For a function that relies on divide-and-conquer technique, there is a way to write it *recursively* and a way to write it *iteratively*
  - The recursive way is often more convenient (more compact code) and more intuitive (easier to understand)
  - If there is no duplication of calculation, both iterative and recursive versions have typically the same time complexity and exhibit similar runtime performance
  - The iterative version has a chance of being more efficient in space because it does not create additional stack frames on the call stack
- Recursive calls take up space on call stack
  - Using as few parameters and local variables as possible reduces the size of a stack frame
  - Cleaner implementation allows you to have more recursive calls before you run out of stack space
  - Regardless, if you have a deep recursion, switch to iteration, e.g., traversing a linked list using a for-loop or while-loop instead of recursion

Recursion vs. iteration

- Dynamic programming typically relies on a recursive formulation, but uses an iterative approach to compute (and store) solutions to smaller problems (to avoid repeated computation)
  - Such an approach usually has better time complexity than a straightforward recursive implementation
  - Need additional space to store computed solutions
- Fibonacci number computation
  - Use an array to store a computed Fibonacci number
  - If it is available, use it; otherwise, call Fibonacci function recursively
  - Time complexity becomes O(n)
- Dynamic programming is an important technique (not formally covered in this class) that you should learn before you head out to interview for software engineering positions

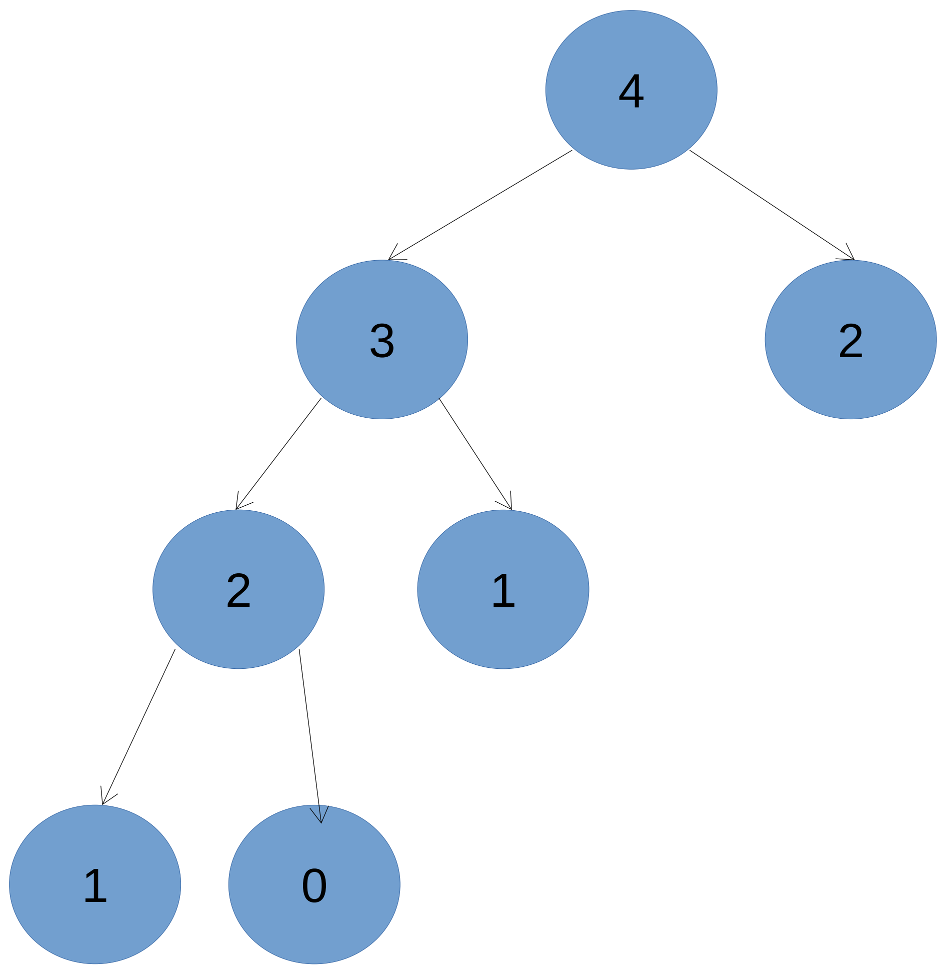
O(n) Fibonacci number

- Assume a global array Fib[·] that has been initialized to store -1 throughout, indicating that we have not computed the Fibonacci numbers

```
int Fibonacci(unsigned int n)
{
  if (Fib[n] != -1)
    return Fib[n];
  if ((n == 0) || (n == 1))
    return Fib[n] = n;
  return Fib[n] = Fibonacci(n-1) + Fibonacci(n-2);
}
```

- There is a better O(n) Fibonacci function
- In fact, there is an O(log n) Fibonacci function

Computation tree of Fibonacci(4)



- There are 2n – 1 function calls for Fibonacci(n)
- Time complexity is O(n)
- Space complexity is still O(n)
  - O(n) from call stack
  - O(n) for Fib[ⅹ]

Tail recursion removal

- There are certain recursive calls that can be turned into an iteration easily
- If the last statement of a function is a recursion, that recursion is called a tail recursion
- All recursive function that is doing a preorder or inorder traversal of the computation tree potentially has a tail recursion
  - In binary search, there are two statements that are recursive calls, but in each function call, at most one of the two recursive calls would be active, and it is a tail recursion
  - In binary search tree construction, switching the order in which we perform the recursive call produces the same binary search tree
  - In tower of Hanoi, the second recursive function call is the tail recursion
- Fibonacci function does not have a tail recursion; the last statement is actually the summation of Fibonacci(n-1) and Fibonacci(n-2), both of which are called before the summation
- Sometimes, it is possible to re-write a recursive function without a tail recursion into a version that has tail recursion

Tail recursion removal

- It is easiest to remove the tail recursion of a recursive function that does not return a value

```
function(a, b, c) {
  if (stopping_condition)
    base_case_body;
  other_divide_and_conquer_body;
  function(x, y, z);
}
```

```
function(a, b, c) {
  while (not stopping_condition) {
    other_divide_and_conquer_body
    a = x, b = y, c = z;
  }
  base_case_body;
}
```

Example: Tower of Hanoi

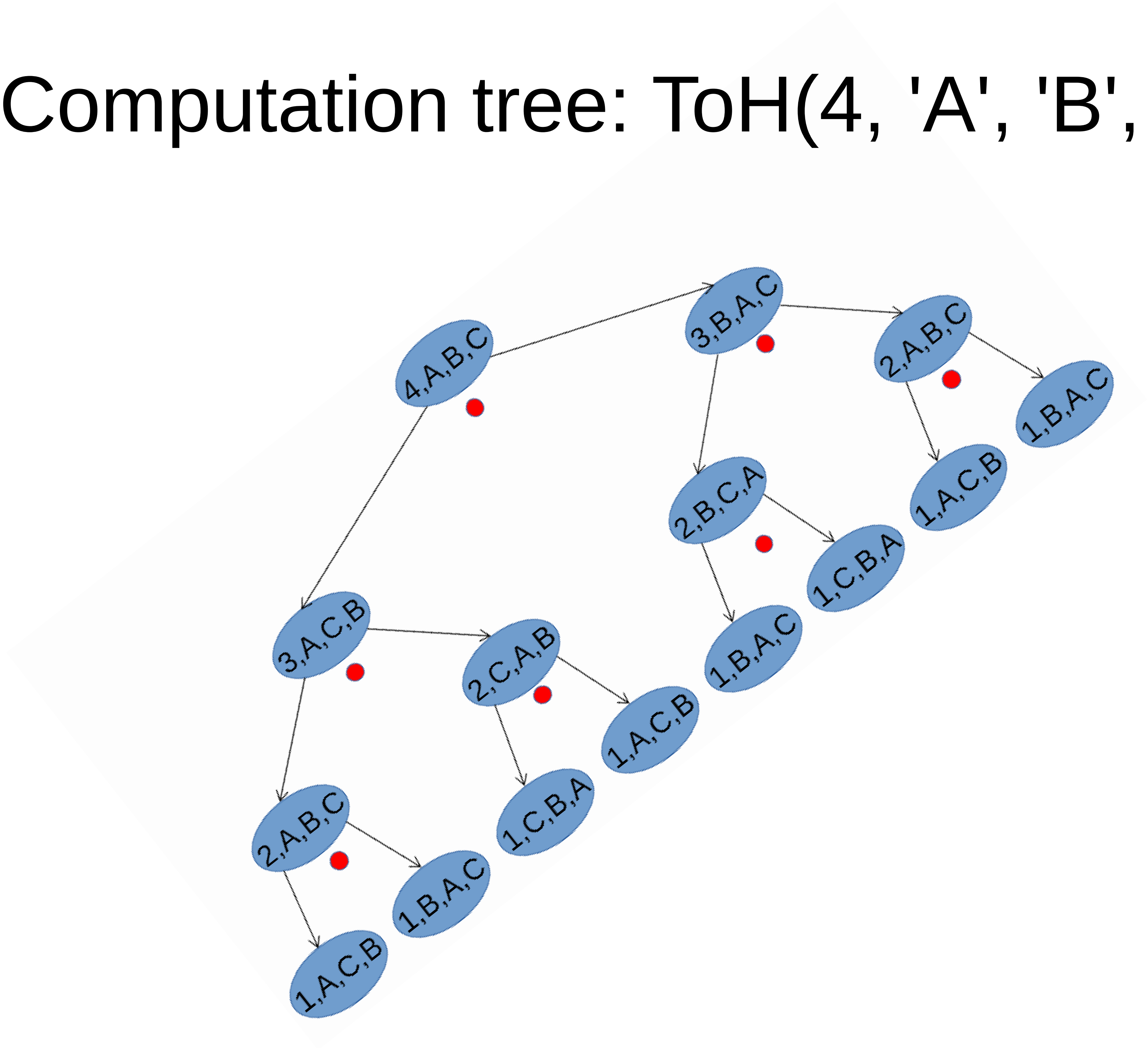
- Move n disks from src rod to dest rod, using intm rod for intermediate storage, assume that n ≥ 1

```
void ToH(int n, char src, char intm, char dest)
{
  if (n == 1) {
    fprintf(stdout, “move disk %d from %c to %c\n”,
            n, src, dest);
    return;
  }
  ToH(n-1, src, dest, intm); // step 1
  fprintf(stdout, “move di sk %d from %c to %c\n”,
          n, src, dest);     // step 2
  ToH(n-1, intm, src, dest); // step 3
}
```

Tower of Hanoi

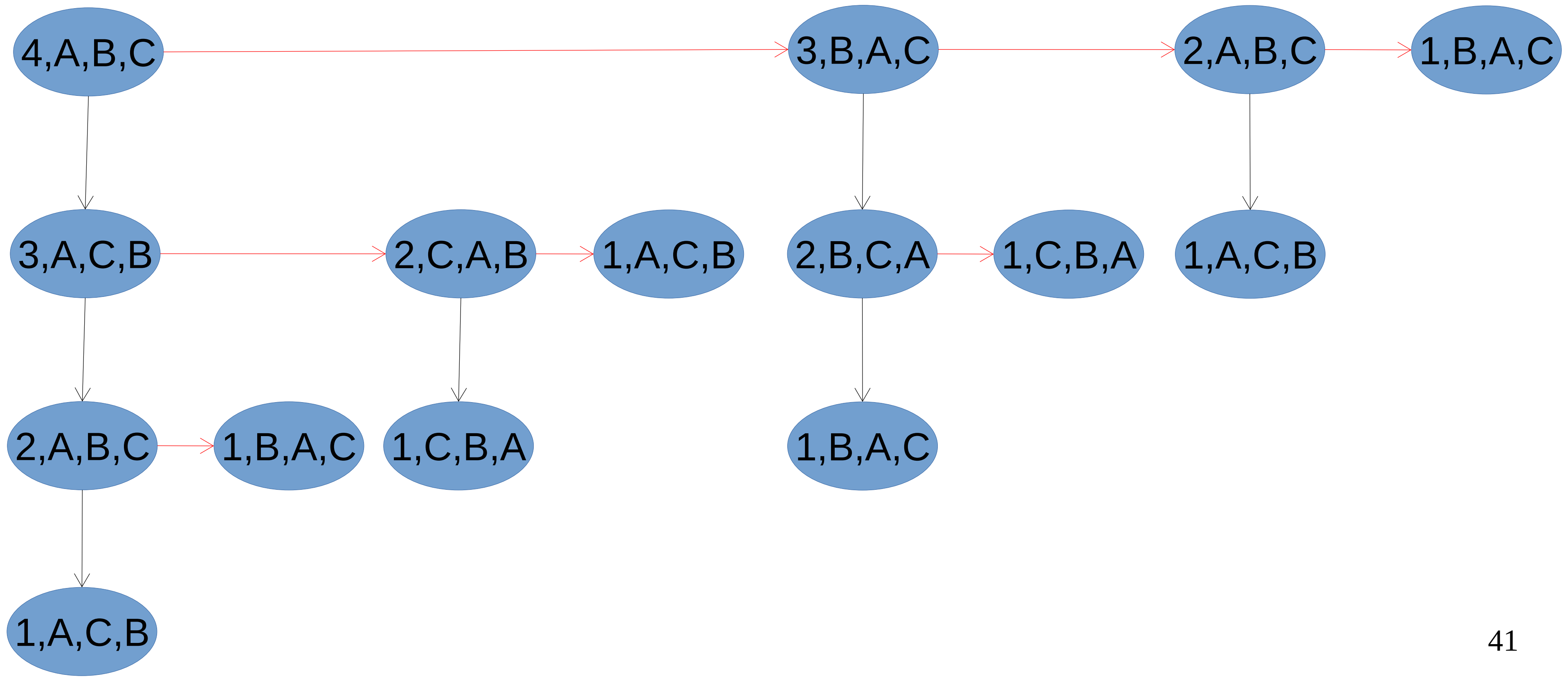
```
void ToH(int n, char src, char intm, char dest)
{
  While (!(n == 1)) {
    ToH(n-1, src, dest, intm); // step 1
    fprintf(stdout, “move disk %d from %c to %c\n”,
            n, src, dest);     // step 2
    // remove ToH(n-1, intm, src, dest); // step 3
    n = n-1;
    char tmp = src;
    src = intm;
    intm = tmp;
  }
  fprintf(stdout, “move disk %d from %c to %c\n”,
          n, src, dest);
}
```

Computation tree: ToH(4, 'A', 'B', 'C')



“Computation tree”

- A right branch is raised to be the same level as its “parent”, indicating that they (right branch and “parent”) occupy the same stack frame because the right branch and its parent in the iteration
- A downward arrow is a recursive call and a horizontal arrow is the tail recursion turned iteration
- All nodes at the same level occupies the same location in the call stack
- Time complexity is still O(2^n) and space complexity is still O(n)



Example: Building a binary search tree

- Given an sorted array a[0..n-1] of n integers, construct a binary search tree where Tnode is a struct to store the integer in data field, left and right are addresses to point at Tnode as left and right child nodes
- lidx is the left most index of array and ridx is the right most index of array

```
Tnode *Build_BST(int *a, int lidx, int ridx)
{
  if (lidx > ridx)
    return NULL;
  int mid = (lidx + ridx)/2;
  Tnode *root = malloc(sizeof(*root));
  if (root != NULL) {
    root->data = a[mid];
    root->left = Build_BST(a, lidx, mid-1);
    root->right = Build_BST(a, mid+1, ridx);
  }
  return root;
}
```

Building binary search tree

- Convert to a version that does not return Tnode *; return through one of the parameters Tnode **
- The caller must have a variable Tnode *root, and we pass &root into the function

```
void Build_BST(int *a, int lidx, int ridx, Tnode **root)
{
  if (lidx > ridx) {
    *root = NULL;
    return;
  }
  int mid = (lidx + ridx)/2;
  *root = malloc(sizeof(**root));
  if (*root == NULL)
    return;
  (*root)->data = a[mid];
  Build_BST(a, lidx, mid-1, &((*root)->left));
  Build_BST(a, mid+1, ridx, &((*root)->right));
}
```

Building binary search tree

```
void Build_BST(int *a, int lidx, int ridx, Tnode **root)
{
  While (!(lidx > ridx)) {
    int mid = (lidx + ridx)/2;
    *root = malloc(sizeof(**root));
    if (*root == NULL)
      return;
    (*root)->data = a[mid];
    Build_BST(a, lidx, mid-1, &((*root)->left));
    lidx = mid+1;
    root = &((*root)->right);
  }
  *root = NULL;
}
```

- Time complexity is still O(n) and space complexity is still O(log n)

Example: Binary search

- Perform binary search on a sorted array of integers a[0, n-1]

```
int Binary_search(int *a, int lidx, int ridx, int key)
{
  if (lidx > ridx) // can't find key in an empty array
    return -1;
  int mid = (lidx + ridx)/2;
  if (a[mid] == key)
    return mid;
  if (a[mid] > key) // key may be in the left half
    return Binary_search(a, lidx, mid-1, key);
  else                 // key may be in the right half
    return Binary_search(a, mid+1, ridx, key);
}
```

Binary search

```
int Binary_search(int *a, int lidx, int ridx, int key)
{
  While (!(lidx > ridx)) {
    int mid = (lidx + ridx)/2;
    if (a[mid] == key)
      return mid;
    if (a[mid] > key) // key may be in the left half
      ridx = mid-1;
    else              // key may be in the right half
      lidx = mid+1;
  }
  return -1;
}
```

- Tail recursion is the only recursion
- Removal of tail recursion converts the function into an iterative one
- Time complexity is still O(log n), but space complexity becomes O(1)

Fibonacci sequence

- Convert to a version that has tail recursion
- Assume that we would call the function with Fibonacci(n, 0)

```
unsigned int Fibonacci(unsigned int n, unsigned int fib)
{
  if ((n == 0) || (n == 1))
    return fib + n;
  fib = Fibonacci(n-1, fib);
  return Fibonacci(n-2, fib);
}
```

Fibonacci sequence

```
unsigned int Fibonacci(unsigned int n, unsigned int fib)
{
  While (!((n == 0) || (n == 1))) {
    fib = Fibonacci(n-1, fib);
    n = n-2;
  }
  return fib + n;
}
```

- Still uses n stack frames
- Switch the order of the recursion call and reduce the stack frames from n to n/2

```
unsigned int Fibonacci(unsigned int n, unsigned int fib)
{
  While (!((n == 0) || (n == 1))) {
    fib = Fibonacci(n-2, fib);
    n = n-1;
  }
  return fib + n;
}
```

Manipulating your own stack (for preorder)

- Use your own stack to store the parameters of recursive function call
- Immediately push the parameters onto your stack in the function
- While your stack not empty
  - Pop from stack the parameters of the topmost function
  - Perform the computation
  - For all recursive calls of this function in reverse order, push their parameters onto the stack
- For building of binary search tree, for example, push the right call followed by the left call so that the left call will be performed when it is popped from the stack
- For other traversal order, it is typically necessary to use multiple stacks to simulate recursions
- You can also combine tail recursion removal and maintaining your own stack

```
void Build_BST(int *a, int lidx, int ridx, Tnode **root)
{
   params stack;       // params is a linked list for stack
   stack.next = NULL;  // initialize a dummy header
   push(&stack, lidx, ridx, root);
   while (!empty(&stack)) {
      params *top = pop(&stack);
      lidx = top->lidx;
      ridx = top->ridx;
      root = top->root;
      free(top);
      if (lidx > ridx) {
         *root = NULL;
         continue;
      }
      int mid = (lidx + ridx)/2;
      *root = malloc(sizeof(**root));
      if (*root == NULL)
         continue;
      (*root)->data = a[mid];
      push(&stack, mid+1, ridx, &((*root)->right));
      push(&stack, lidx, mid-1, &((*root)->left));
   }
}
```