C Programming

Recursion

Spring 2017
Recitation 4

Outline

Recursion

- You know what recursion is.
- The idea is to get closer to the solution -usually by making the problem smaller- in each recursive call.
- Do not forget the base case!

Recursion Types

- Linear: single call to itself
  - Tail recursion: the recursive call is the last thing
- Exponential: multiple calls
  - Binary: only two calls
- Mutual: Multiple functions calling each other
- Nested: A recursive call as an argument of the function itself

Examples

See *recursion.c* file for several examples of simple recursive functions

- Linear: factorial
- Tail: factorial_2
- Binary: fibonacci
- Nested: ackermann
- Mutual: even-odd

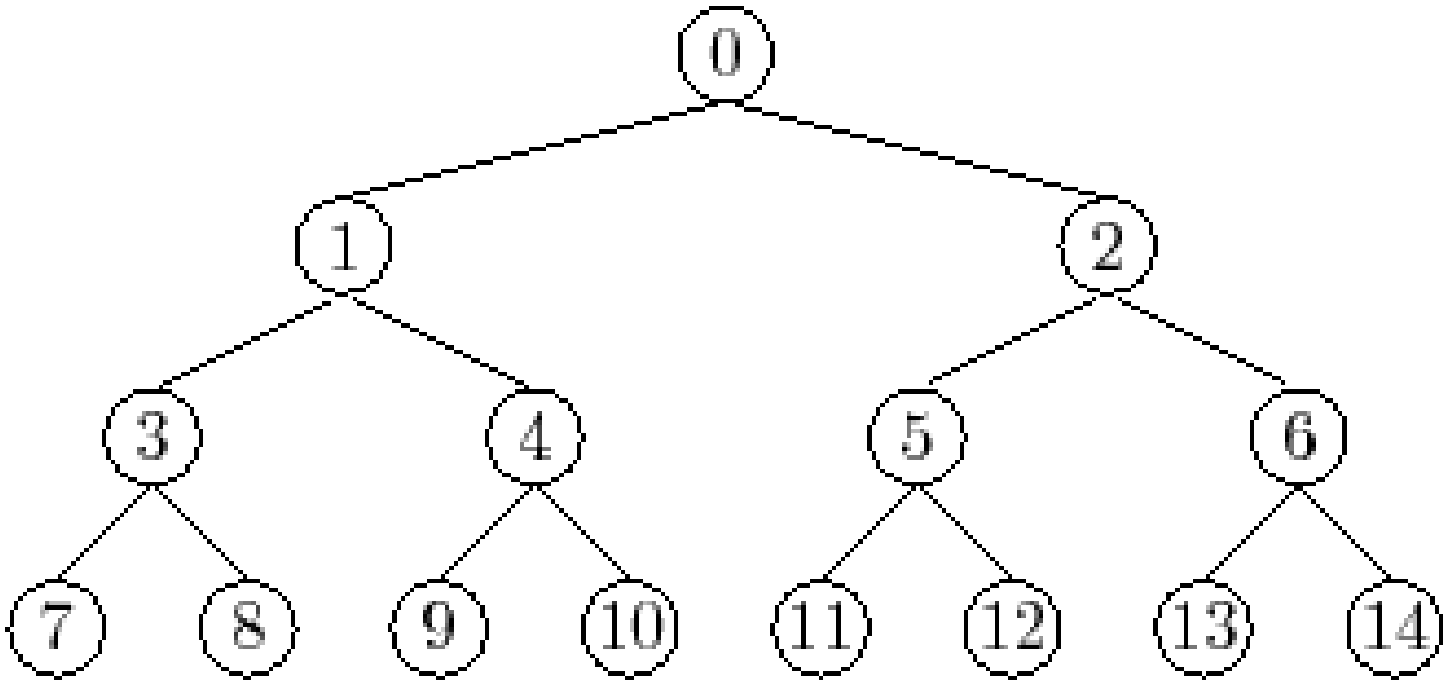
These are not the best ways to solve these problems, rather simple examples to demonstrate recursive functions.

This Recitation

- We will not spend time on simple examples like Factorial of Fibonacci in this recitation.
- We will focus on solving problems using helper functions with extra arguments.
- Recursive functions heavily rely on return values and parameters.
- Most of the time, you are expected to provide a function with a set of parameters that make it very hard or impossible to implement a recursive solution.
- When you encounter such a case, you need to define helper functions with additional parameters that make recursions work.

A Binary Tree

We will use a simple array representation for binary trees and implement recursive functions on these trees.

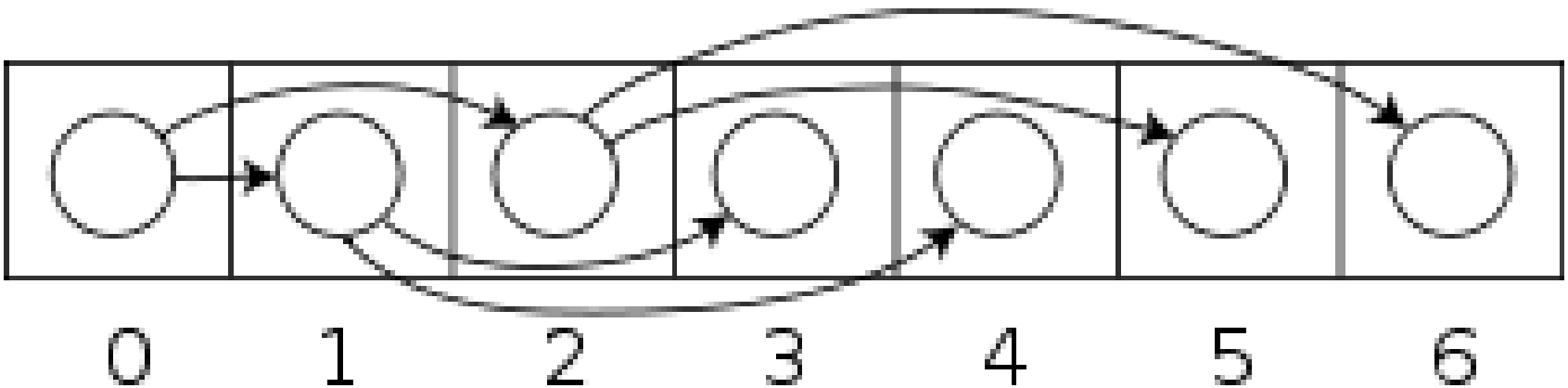


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

A Binary Tree

For a given node at index i

- the left child is at position $2 * i + 1$
- the right child is at position $2 * i + 2$
- the parent is at position $i - 1/2$



Binary Search

- For the first example, we will assume a special tree such that
  - every left child has a lesser value than its parent.
  - every right child has a greater value than its parent.
- This kind of a tree is called a binary search tree.
- We will try to find a given item in a given BST.
- The function prototype is given as below:

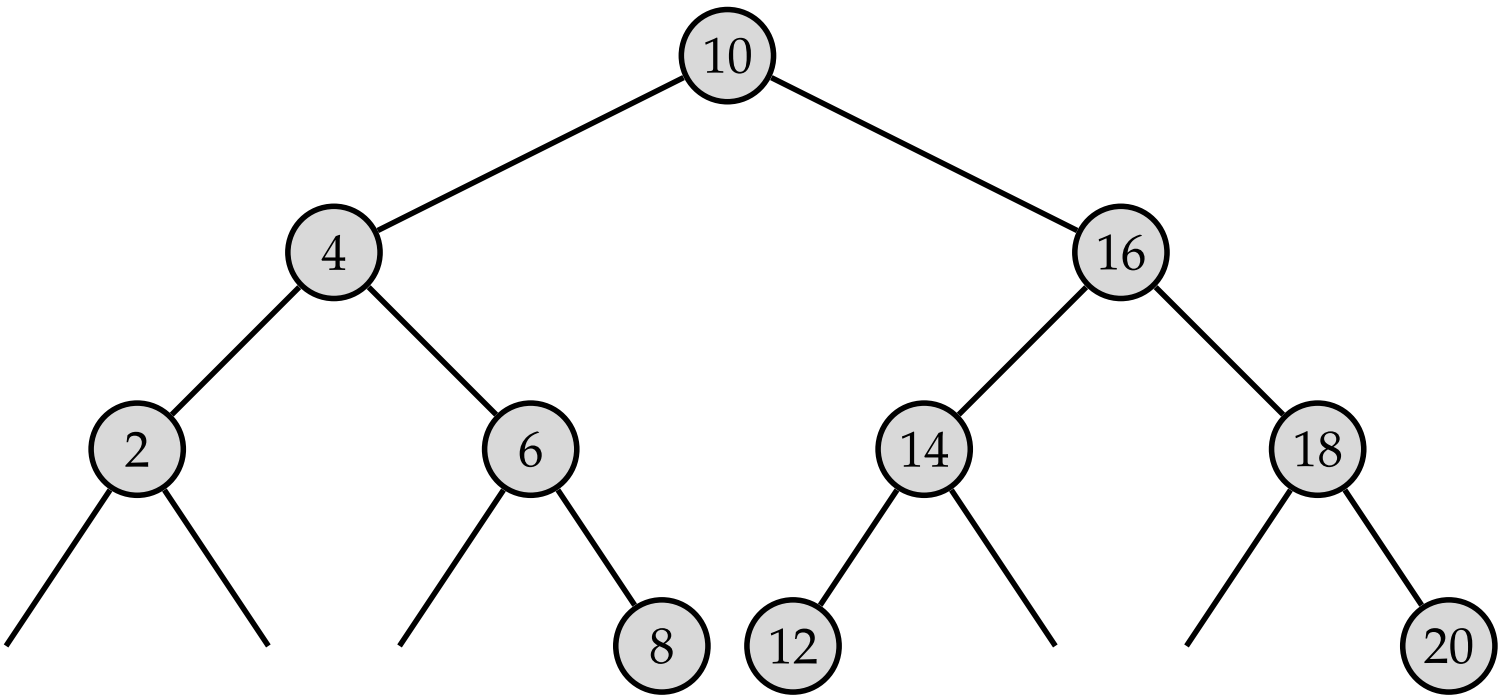
```
int search(int *tree, int size, int a);
```

- Its arguments are the array, the size of the array and the value we are looking for. For convenience, the array size is provided.
- It returns the index of a if the value is found, -1 otherwise.

A Binary Search Tree Example

This array represents the BST below (with 0's for empty nodes):

```
int bst[] = {10, 4, 16, 2, 6, 14, 18, 0, 0, 0, 8, 12, 0, 0, 20};
int bst_size = 15;
```



Binary Search

- In binary search over a BST, we look for the root value, depending on its value we continue searching on the one subtrees, either left or right.
- One way to do this is to create a new tree that contains all elements of the subtree and pass to the function at the recursive call.
- However, this is very time and memory consuming.
- Instead, we can keep the original array/tree, but search only on a subsection of it.

```
int search_help(int *tree, int size, int a, int root);
```

- In this helper function, we use an extra parameter to show the root of the subtree.

Binary Search

- We will delegate all the work to the helper function and only call it in the main search function with initial argument as 0 which is the index of the root of the whole tree.
- This is the search function implementation:

```
int search(int *tree, int size, int a){
        return search_help(tree, size, a, 0);
}
```

Binary Search

This is the helper function that does binary search recursively: [1]

```
int search_help(int *tree, int size, int a, int root){
    if (root >= size || tree[root] == 0)
        return -1;
    else if (tree[root] == a)
        return root;
    else if (tree[root] > a)
        return search_help(tree, size, a, left(root));
    else
        return search_help(tree, size, a, right(root));
}
```

[1] The functions in the *binary_tree.c* also print the arguments in each call to trace the recursive calls, they are omitted on the slides for brevity.

Binary Search with a Limit

- This time we will add a constraint to the search.
- We are only interested in nodes within a certain distance of the root.
- Therefore we will limit the depth of the search with an additional parameter:

```
int search_limit(int *tree, int size, int a, int limit);
```

Binary Search with a Limit

- The main function implementation is very similar except we also pass the *limit* to the helper function

```
int search_limit(int *tree, int size, int a, int limit){
    return search_limit_help(tree, size, a, 0, limit);
}
```

Binary Search with a Limit

We also have to modify the recursive calls in the helper function. Also, do not forget to modify the base case:

```
int search_limit_help(int *tree, int size, int a, int root, int limit){

    if (root >= size || tree[root] == 0 || limit == 0){
        return -1;
    else if (tree[root] == a)
        return root;
    else if (tree[root] > a)
        return search_limit_help(tree, size, a, left(root), limit-1);
    else
        return search_limit_help(tree, size, a, right(root), limit-1);
}
```

Find Path to a Given Node

- The search examples use linear recursion, only one of the recursive calls in the if-else blocks is used.
- We will see a more complex example that uses two recursive calls at each step.
- This function will find the path from the root of the tree to a given node:

```
int path(int *tree, int size, int a, int *p);
```

- For convenience, p is an array with enough space to keep the path. We do not need to worry about the allocation of the array or the size of the maximum path in this example.

Find Path to a Given Node

The *path* function calls the helper with root as 0: [2]

```
int path(int *tree, int size, int a, int *p){
    return path_help(tree, size, a, p, 0);
}
```

[2] The *path* functions in the *binary_tree.c* have an extra argument, *level*, for printing indentations in the function traces. The argument and the print statements are removed in these slides.

Find Path to a Given Node

This is the helper function that will find the path from the root of the tree to a given node:

```
int path_help(int *tree, int size, int a, int *p, int root){
    int len = 0;
    if (root >= size || tree[root] == 0)
        return 0;
    else if (tree[root] == a ||
        (len = path_help(tree, size, a, p, left(root))) ||
        (len = path_help(tree, size, a, p, right(root)))
        ){
        p[len++] = root;
        return len;
    }
    else
        return 0;
}
```

Find Path to a Given Node

- This time, we have a more capable helper function, it returns the length of the path if it can find the node in the tree. It returns 0 otherwise.
- The return value is very critical and we use it as the primary condition for a successful search.

```
len = path_help(tree, size, a, p, left(root))
```

- The condition for success will only hold in one of three conditions hold:
  - if the current root node is the one we are looking for
  - if we can find the node in the left subtree, which returns a non-zero value if that's the case
  - if we can find the node in the right subtree

Find Path to a Given Node

- The function is recursively called until the node is found, then it begins to append the nodes to the path and returns.
- At the end, p contains the path in reverse order.
- If none of the conditions returns true, then we return 0 to indicate no path is found.