

**CSE1303 Part A
Data Structures and Algorithms
Summer Semester 2003**

Lecture A11 – Recursion

Kymerly Fergusson

Overview

- Unary Recursion
- Binary Recursion
- Examples
- Features
- Stacks
- Disadvantages
- Advantages

2

What is Recursion - Recall

- A procedure defined in terms of itself
- Components:
 - Base case
 - Recursive definition
 - Convergence to base case

3

```
double power(double x, int n)
{
    int i;
    double tmp = 1;

    if (n > 0)
    {
        for (i = 0; i < n; i++)
        {
            tmp *= x;
        }
    }

    return tmp;
}
```

4

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

5

Unary Recursion

- Functions calls itself once (at most)
- Usual format:

```
function RecursiveFunction (<parameter(s)>)
{
    if (base case) then
        return base value
    else
        return RecursiveFunction (<expression>)
}
```

6

Unary Recursion

```

/* Computes the factorial */
function Factorial ( n )
{
    if ( n is less than or equal to 1 )
    then
        return 1
    else
        return n * Factorial ( n - 1 )
}

int factorial ( int n )
{
    if ( n <= 1 )
    {
        return 1;
    }
    else
    {
        return n*factorial(n-1);
    }
}

```

7

Binary Recursion

- Defined in terms of two or more calls to itself.
- For example – Fibonacci
 - A series of numbers which
 - begins with 0 and 1
 - every subsequent number is the sum of the previous two numbers
- 0, 1, 1, 2, 3, 5, 8, 13, 21,...

8

Binary Recursion

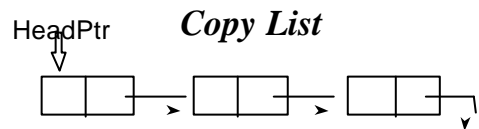
```

function Fibonacci ( n )
{
    if ( n is less than or equal to 1 ) then
        return n
    else
        return Fibonacci ( n - 2 ) + Fibonacci ( n - 1 )
}

/* Compute the n-th Fibonacci number */
long fib ( long n )
{
    if ( n <= 1 )
        return n ;
    else
        return fib( n - 2 ) + fib( n - 1 );
}

```

9



Copy List

```

struct LinkedListRec
{
    int    count;
    Node*  headPtr;
};

typedef struct LinkedListRec List;

```

10

```

#include "linkList.h"
Node* copyNodes(Node* oldNodePtr);
void copyList(List* newListPtr, List* oldListPtr)
{
    if (listEmpty(oldListPtr))
    {
        initializeList(newListPtr);
    }
    else
    {
        newListPtr->headPtr = copyNodes(oldListPtr->headPtr);
        newListPtr->count = oldListPtr->count;
    }
}

```

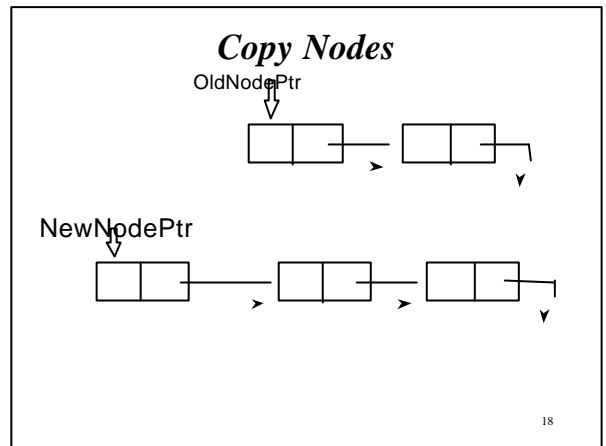
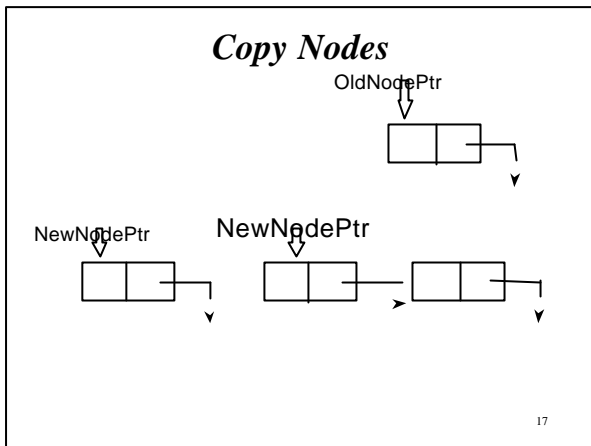
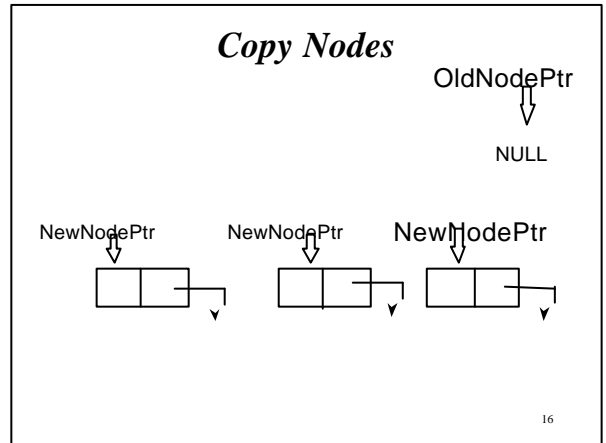
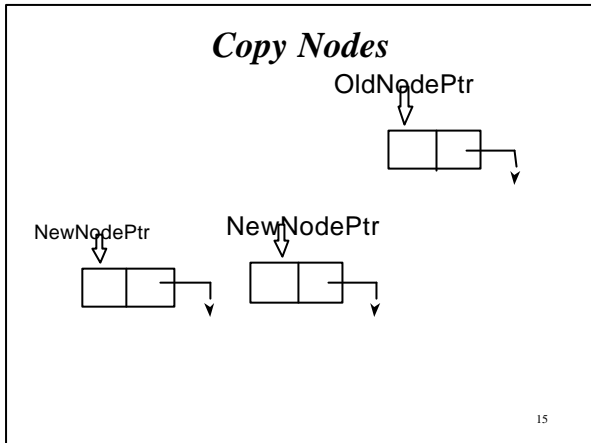
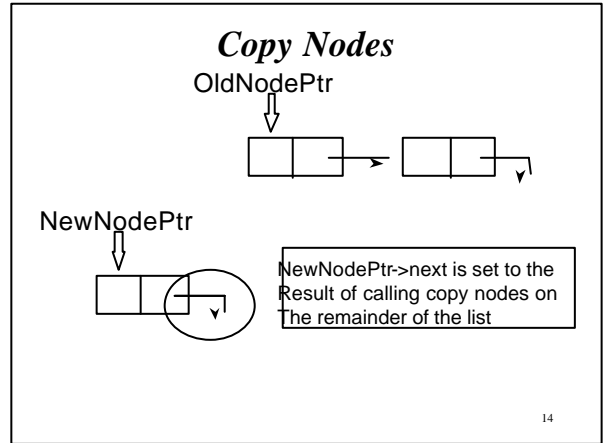
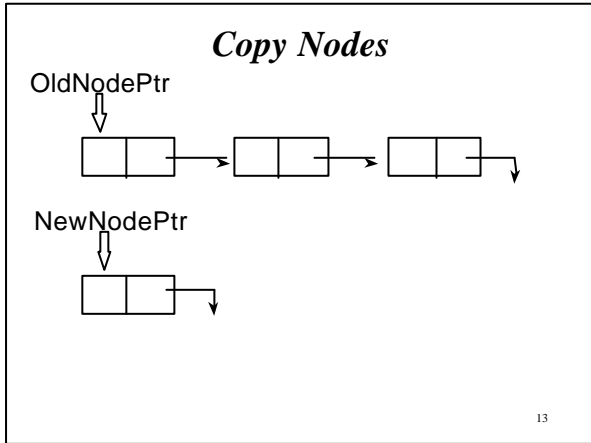
11

```

Node* copyNodes(Node* oldNodePtr)
{
    Node* newNodePtr = NULL;
    if (oldNodePtr != NULL)
    {
        newNodePtr = makeNode(oldNodePtr->value);
        newNodePtr->nextPtr = copyNodes(oldNodePtr->nextPtr);
    }
    return newNodePtr;
}

```

12



Recursion Process

Every recursive process consists of:

1. A base case.
2. A general method that reduces to the base case.

19

Types of Recursion

- Direct.
- Indirect.
- Linear.
- n-ary (Unary, Binary, Ternary, ...)

20

Stacks

- Every recursive function can be implemented using a stack and iteration.
- Every iterative function which uses a stack can be implemented using recursion.

21

```
double power(double x, int n)
{
    double tmp = 1;
    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

22

```
double power(double x, int n)
{
    double tmp = 1;
    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1
x = 2, n = 2
tmp = 1

23

```
double power(double x, int n)
{
    double tmp = 1;
    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1
x = 2, n = 2
tmp = 1
x = 2, n = 1
tmp = 1

24

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

x = 2, n = 2
tmp = 1

x = 2, n = 1
tmp = 1

x = 2, n = 0
tmp = 1

25

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

x = 2, n = 2
tmp = 1

x = 2, n = 1
tmp = 1*1*2
= 2

26

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 1

x = 2, n = 2
tmp = 2*2
= 4

27

```
double power(double x, int n)
{
    double tmp = 1;

    if (n > 0)
    {
        tmp = power(x, n/2);
        if (n % 2 == 0)
        {
            tmp = tmp*tmp;
        }
        else
        {
            tmp = tmp*tmp*x;
        }
    }
    return tmp;
}
```

x = 2, n = 5
tmp = 4*4*2
= 32

28

```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push n onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
        else {tmp = tmp*tmp*x}
    }
}
```

n = 5, x = 2

Stack

29

```
module power(x, n)
{
    create a Stack
    initialize a Stack
    loop{
        if (n == 0) then {exit loop}
        push n onto Stack
        n = n/2
    }
    tmp = 1
    loop {
        if (Stack is empty) then {return tmp}
        pop n off Stack
        if (n is even) {tmp = tmp*tmp}
        else {tmp = tmp*tmp*x}
    }
}
```

n = 5, x = 2

Stack

5

30

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 2, x = 2 Stack

2
5

31

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 1, x = 2 Stack

1
2
5

32

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 0, x = 2 Stack

1
2
5

33

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 0, x = 2 Stack

2
5

34

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 0, x = 2 Stack

5

35

```

module power(x, n)
{
  create a Stack
  initialize a Stack
  loop{
    if (n == 0) then {exit loop}
    push n onto Stack
    n = n/2
  }
  tmp = 1
  loop {
    if (Stack is empty) then {return tmp}
    pop n off Stack
    if (n is even) {tmp = tmp*tmp}
    else {tmp = tmp*tmp*x}
  }
}
    
```

n = 0, x = 2 Stack

36

```
double power(double x, int n)
{
    double tmp = 1;
    Stack theStack;

    initializeStack(&theStack);
    while (n != 0) {
        push(&theStack, n);
        n /= 2;
    }

    while (!stackEmpty(&theStack)) {
        n = pop(&theStack);
        if (n % 2 == 0)
            { tmp = tmp*tmp;}
        else
            { tmp = tmp*tmp*x;}
    }
    return tmp;
}
```

37

Disadvantages

- May run slower.
 - Compilers
 - Inefficient Code
- May use more space.

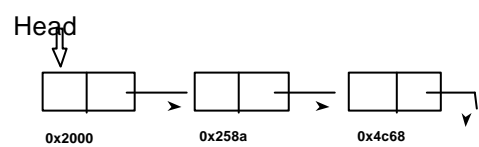
38

Advantages

- More natural.
- Easier to prove correct.
- Easier to analysis.
- More flexible.

39

Free List – Non Recursive

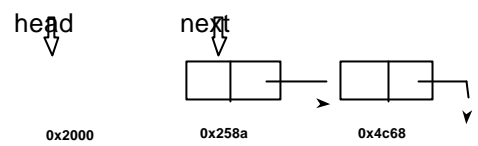


```
/* Delete the entire list */
void FreeList(Node* head){
    Node* next;

    while (head != NULL) {
        next=head->next;
        free(head);
        head=next;
    }
}
```

40

Free List – Non Recursive

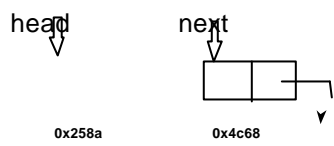


```
/* Delete the entire list */
void FreeList(Node* head){
    Node* next;

    while (head != NULL) {
        next=head->next;
        free(head);
        head=next;
    }
}
```

41

Free List – Non Recursive

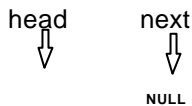


```
/* Delete the entire list */
void FreeList(Node* head){
    Node* next;

    while (head != NULL) {
        next=head->next;
        free(head);
        head=next;
    }
}
```

42

Free List – Non Recursive



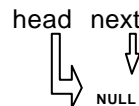
```

/* Delete the entire list */
void FreeList(Node* head){
    Node* next;

    while (head != NULL) {
        next=head->next;
        free(head);
        head=next;
    }
}
    
```

43

Free List – Non Recursive



```

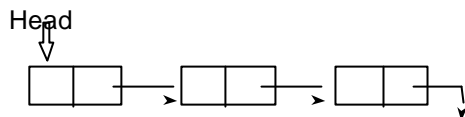
/* Delete the entire list */
void FreeList(Node* head){
    Node* next;

    while (head != NULL) {
        next=head->next;
        free(head);
        head=next;
    }
}
    
```

Has local variables on the stack. This is performing two assignments and one comparison per iteration.

44

Free List

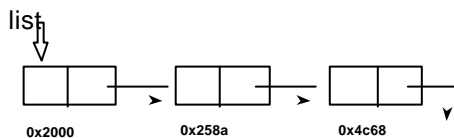


```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

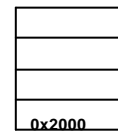
45

Free List



```

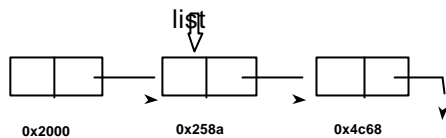
/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```



Stack in memory

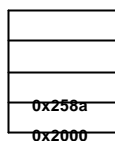
46

Free List



```

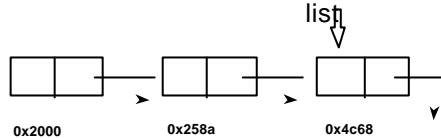
/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```



Stack in memory

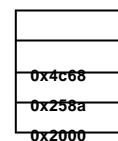
47

Free List



```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```



Stack in memory

48

Free List

```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

NULL
0x4c68
0x258a
0x2000

Stack in memory

49

Free List

```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

0x4c68
0x258a
0x2000

Stack in memory

50

Free List

```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

0x258a
0x2000

Stack in memory

51

Free List

```

/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

0x2000

Stack in memory

52

Free List

Head
↓

```

0x2000
/* Delete the entire list */
void FreeList(Node* list)
{
    if (list==NULL)
        return;
    FreeList(list->next);
    free(list);
}
    
```

Has no local variables on the stack. This is performing one assignment and one comparison per function call.

53

Revision

- Recursion

54

Revision: Reading

- Kruse 3.2 - 3.4
- Standish 3, 14
- Langsam 3

Preparation

Next lecture: Binary Trees

- Read Chapter 9 in Kruse et al.

55