CSE1303 Part A
Data Structures and Algorithms
Summer Semester 2003

Lecture A11 – Recursion

Kymberly 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

```
function Factorial ( n )
{
  if ( n is less than or equal to 1)
  then
    return 1
  else
    return n ´ Factorial ( n - 1 )
}
```

```
/* Computes the factorial */

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

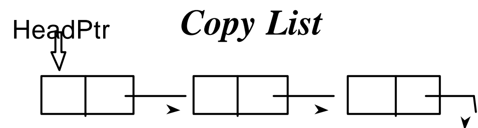
Binary Recursion

```
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;
```

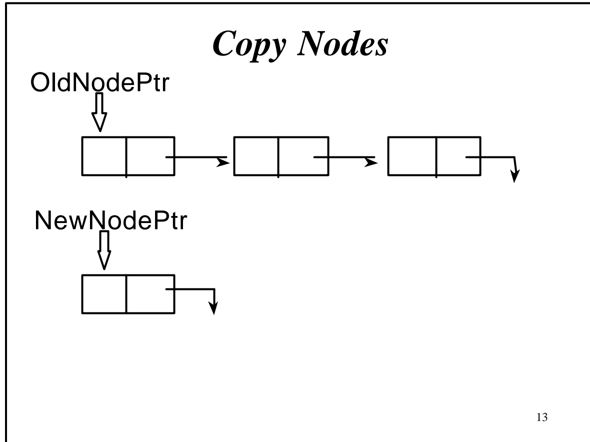
10

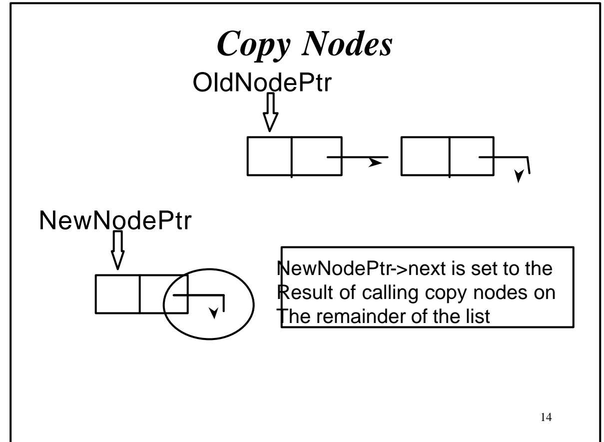
```
#include "linkList.h"

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;
   }
}
```

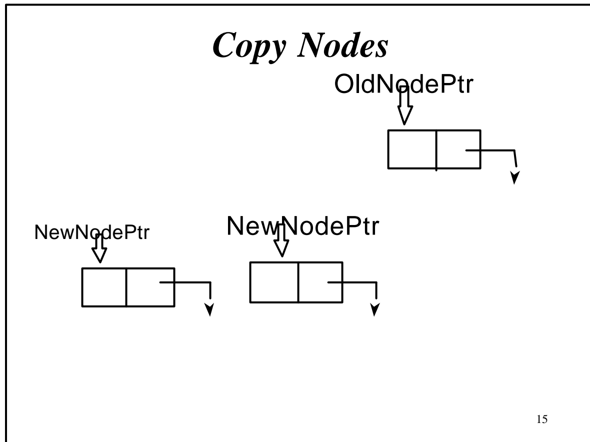
11

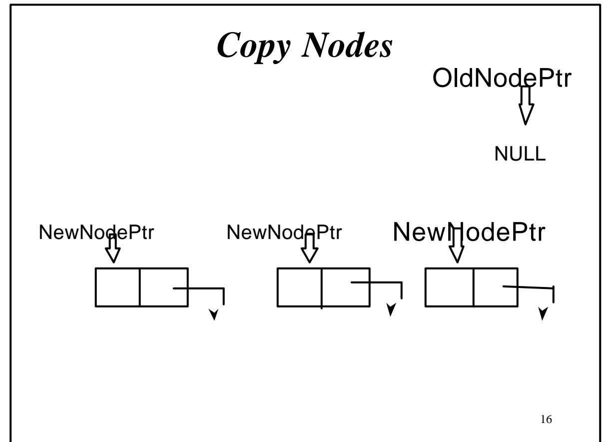
```
Node* copyNodes(Node* oldNodePtr)
{
   Node* newNodePtr = NULL:

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

   return newNodePtr;
}
```

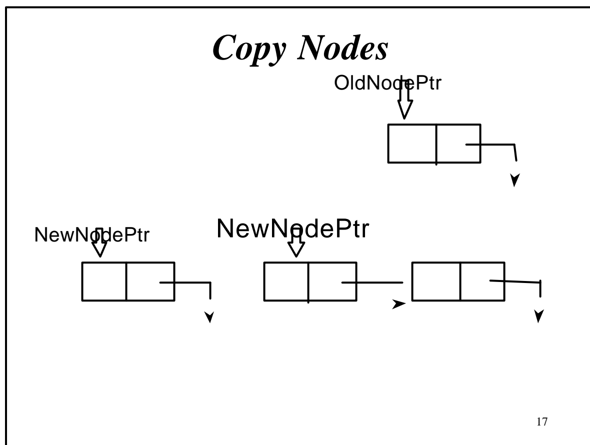
12

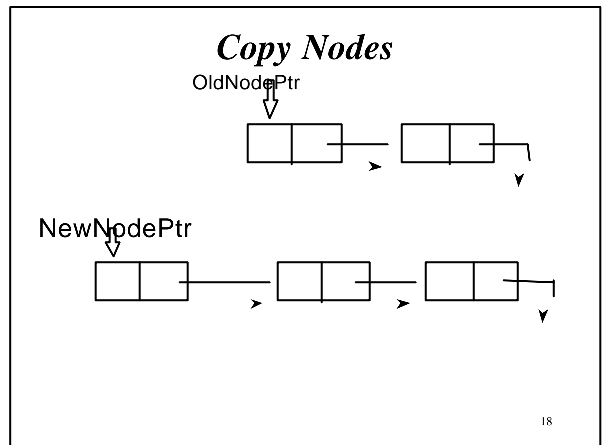

Copy Nodes
OldNodePtr
NewNodePtr
13


Copy Nodes
OldNodePtr
NewNodePtr
NewNodePtr->next is set to the
Result of calling copy nodes on
The remainder of the list
14


Copy Nodes
OldNodePtr
NewNodePtr
NewNodePtr
15


Copy Nodes
OldNodePtr
NULL
NewNodePtr
NewNodePtr
NewNodePtr
16


Copy Nodes
OldNodePtr
NewNodePtr
NewNodePtr
17


Copy Nodes
OldNodePtr
NewNodePtr
18

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

```
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;
}
```

Call	Variable	Value
Fourth call	tmp	1
	n	0
	x	5
Third call	tmp	1
	n	1
	x	5
Second call	tmp	1
	n	2
	x	5
First call	tmp	1
	n	2
	x	5

Runtime stack

29

```
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;
}
```

Call	Variable	Value
Return to Third call	tmp	2
	n	1
	x	5
Second call	tmp	1
	n	2
	x	5
First call	tmp	1
	n	2
	x	5

Runtime stack

30

```
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;
}
```

Return to Second call: tmp 4, n 2, x 5

First call: tmp 1, n 2, x 5

Runtime stack

31

```
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;
}
```

Return to First call: tmp 32, n 2, x 5

Runtime stack

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 = 5, x = 2

Stack: (empty)

Runtime stack: tmp, n 5, x 2

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 = 5, x = 2

Stack: 5

Runtime stack: tmp, n 5, x 2

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 = 2, x = 2

Stack: 2, 5

Runtime stack: tmp, n 2, x 2

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 = 1, x = 2

Stack: 1, 2, 5

Runtime stack: tmp, n 1, x 2

36

```
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

tmp = 1

Stack: 1, 2, 5

Runtime stack: tmp 1, n 0, x 2

37

```
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

tmp = 2

Stack: 2, 5

Runtime stack: tmp 2, n 0, x 2

38

```
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

tmp = 4

Stack: 5

Runtime stack: tmp 4, n 0, x 2

39

```
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

tmp = 32

Stack

Runtime stack: tmp 32, n 0, x 2

40

```
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;
}
```

41

Disadvantages

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

Advantages

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

42

Free List – Non Recursive

headPtr

0x2000 0x258a 0x4c68

```
/* Delete the entire list */
void FreeList(Node* headPtr){
  Node* nodePtr;

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

nodePtr
headPtr 0x2000

Runtime stack

43

Free List – Non Recursive

headPtr nodePtr

0x2000 0x258a 0x4c68

```
/* Delete the entire list */
void FreeList(Node* headPtr){
  Node* nodePtr;

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

nodePtr 0x258a
headPtr 0x2000

Runtime stack

44

Free List – Non Recursive

headPtr nodePtr

0x258a 0x4c68

```
/* Delete the entire list */
void FreeList(Node* headPtr){
  Node* nodePtr;

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

nodePtr 0x4c68
headPtr 0x258a

Runtime stack

45

Free List – Non Recursive

headPtr nodePtr

NULL

0x4c68

```
/* Delete the entire list */
void FreeList(Node* headPtr){
  Node* nodePtr;

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

nodePtr NULL
headPtr 0x4c68

Runtime stack

46

Free List – Non Recursive

headPtr nodePtr

NULL

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

```
/* Delete the entire list */
void FreeList(Node* headPtr){
  Node* nodePtr;

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

nodePtr NULL
headPtr NULL

Runtime stack

47

Free List

headPtr

0x2000 0x258a 0x4c68

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

0x2000

Runtime stack

48

Free List


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


Free List


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


Free List


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


Free List


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


Free List


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


Free List

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

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

Revision

- Recursion

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