RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 09 – Dynamic Data Structures (cont.)

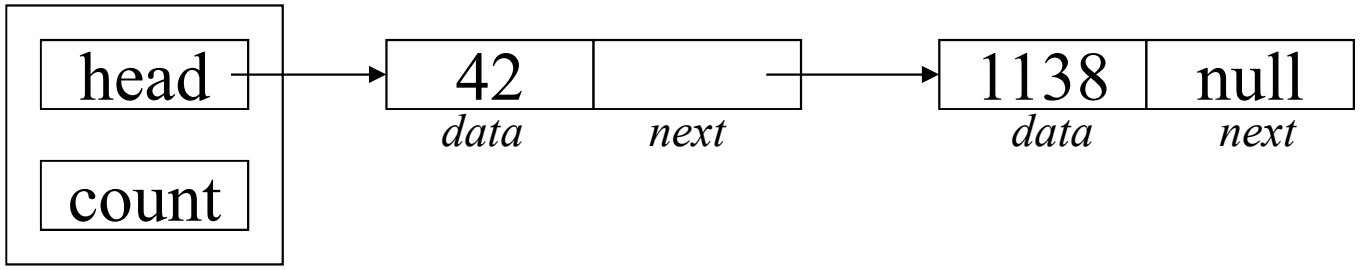
"There are two ways to write error-free programs;
only the third one works."
-- anon.

Topics

- Dynamically allocated linked lists (cont.)
- Recursive data structures
- Recursion
- Recursive traversal of linked lists
- ADT: Binary Search Trees

Linked Lists (cont'd)

- Singly Linked List



IntList - intlist.h using a linked list

```
/* IntList
 * -- simple unordered linked list implementation using dynamic memory
 */

… as per previous versions …

typedef struct intlistnode * IntListNodePtr;
typedef struct intlistnode
{   int num;
    IntListNodePtr next;
} IntListNode;

typedef struct
{   IntListNodePtr head;
    unsigned size;
} IntList;

… as per previous version …
```

IntList - intlist.c

```
/*
 * IntList
 * -- simple unordered linked list implementation using dynamic memory
 */

#include <stdio.h>
#include <stdlib.h>

#include "intlist-linked-list.h"

int MakeList(IntList* pil,int size)
{
   pil->head = NULL;
   pil->size = 0;

   return SUCCESS;
}
```

IntList - intlist.c (cont'd)

```
void FreeList(IntList* pil)
{
   IntListNodePtr current, next;

   current = pil->head;

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

   pil->head = NULL;
   pil->size = 0;
}
```

IntList - intlist.c (cont'd)

```
int AddList(IntList* pil,int num)
{
   IntListNodePtr newNode;

   if ((newNode = malloc(sizeof *newNode)) == NULL)
   {
      return FAILURE;
   }

   newNode->num = num;
   newNode->next = pil->head;
   pil->head = newNode;
   pil->size += 1;

   return SUCCESS;

}
```

IntList - intlist.c (cont'd)

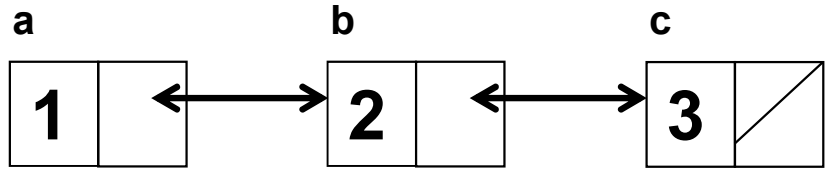
```
void DisplayList(IntList* pil)
{
   IntListNodePtr current;

   current = pil->head;
   while (current != NULL)
   {
      printf("%d\n", current->num);
      current = current->next;
   }
}

unsigned SizeList(IntList* pil)
{
   return pil->size;
}
```

Traversing a list – iterative solution

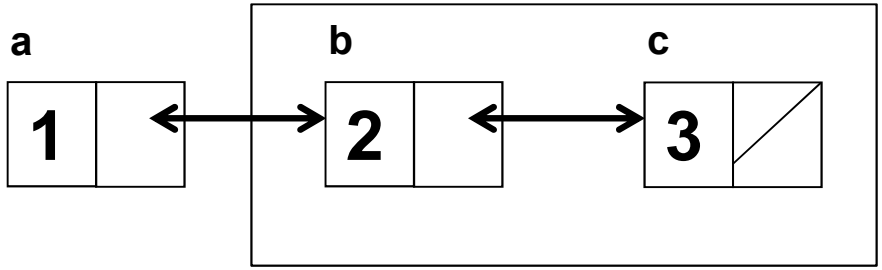
```
void DisplayList(IntList* pil)
{
   IntListNodePtr current;

   current = pil->head;
   while (current != NULL)
   {
      printf("%d\n", current->num);
      current = current->next;
   }
}
```

Lists are recursive structures

```
typedef struct intlistnode * IntListNodePtr;
typedef struct intlistnode
{   int num;
    IntListNodePtr next;
} IntListNode;
```



Considering the IntListNodes, list a,b,c = a + sub-list b,c



Can take advantage of lists recursive structure by using recursive functions

Recursion

- The ability for a function to call itself (either directly or indirectly)
- Some problems/solutions are inherently recursive – e.g. the sum of the first N natural numbers

NaturalSum(N) = N + NaturalSum(N-1) for N > 1

NaturalSum(N) = 1 for N = 1

e.g. NaturalSum(4) = 4 + NaturalSum(3)
= 4 + 3 + NaturalSum(2)
= 4 + 3 + 2 + NaturalSum(1)
= 4 + 3 + 2 + 1
= 10

Recursion (cont'd)

- In languages that allow recursion, such as C, this can lead to simple, elegant solutions – e.g.

```
int naturalSum(int n)
{
    if ( n == 1 ) /* terminating condition */
       return 1;
    else
        return (n + naturalSum(n-1));
}
```

Recursion (cont'd)

- Most simple recursive functions can easily be rewritten as iterative functions

```
/* Iterative factorial */
int factorial (int n)
{
  int product = 1;

  for (; n > 1; --n)
    product *= n;

  return product;
}
```

Recursion (cont'd)

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

Recursion (cont'd)

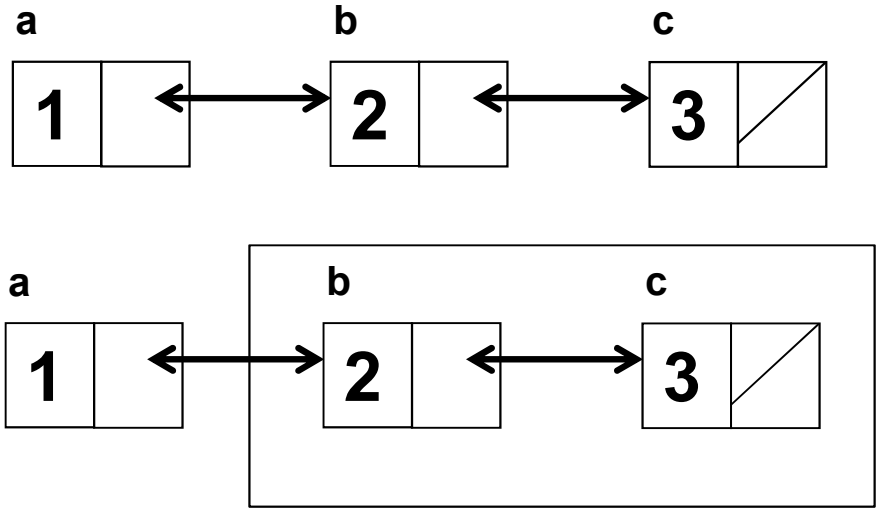
- An elegant recursive function to calculate Fibonacci numbers

  Fibonacci numbers are $fib_0 = 0$, $fib_1 = 1$ and $fib_{i+1} = fib_i + fib_{i-1}$ for i = 1, 2, …

```
/* Fibonacci numbers */
int fib(int n)
{
  if(n <= 1)
    return n;
  else
    return(fib(n-1) + fib(n-2));
}
```

Lists are recursive structures

```
typedef struct intlistnode * IntListNodePtr;
typedef struct intlistnode
{   int num;
    IntListNodePtr next;
} IntListNode;
```



Considering the IntListNodes, list a,b,c = a + sub-list b,c

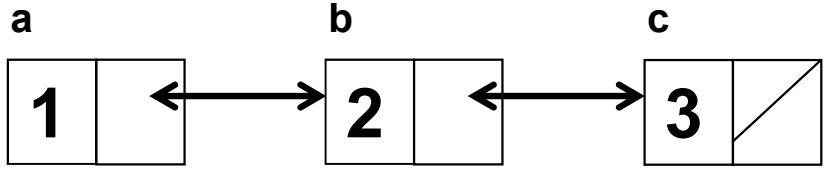
Traversing a list – recursive solution

```
void DisplayNodeList(IntListNode* current)
{
   if (current != NULL)
   {
      printf("%d\n", current->num);
      DisplayNodeList(current->next);
   }
}
```

```
void DisplayList(IntList* pil)
{
   DisplayNodeList(pil->head);
}
```

Traversing a list – recursive solution

```
void DisplayNodeList(IntListNode* current)
{
   if (current != NULL)
   {
      printf("%d\n", current->num);
      DisplayNodeList(current->next);
   }
}
```



1 2 3

DisplayNodeList(a)
printf... a->num
DisplayNodeList(a->next)

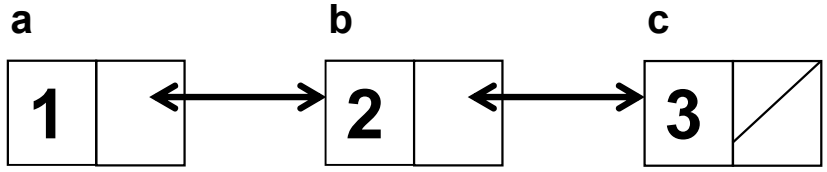
DisplayNodeList(b)
printf... b->num
DisplayNodeList(b->next)

DisplayNodeList(c)
printf... c->num
DisplayNodeList(c->next)

DisplayNodeList(NULL)
'if' test fails

Traversing a list – recursive solution

```
void DisplayNodeList(IntListNode* current)
{
   if (current != NULL)
   {
      DisplayNodeList(current->next);
      printf("%d\n", current->num);
   }
}
```



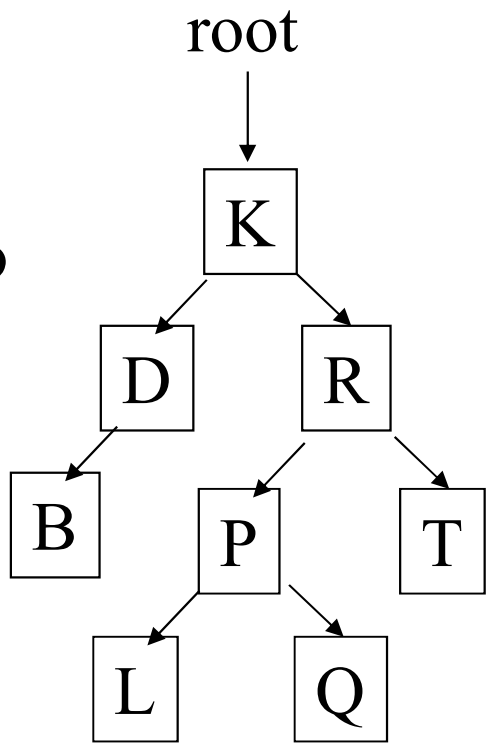
3 2 1

Abstract Data Types: BST

- Example: A **binary search tree (BST)**
  - Also called *sorted binary trees* or *ordered binary trees*
  - Actions (uses)
    - Insert, Remove, Search for an item in the tree
    - Ask if a tree is empty
  - Implementation of the data structure
    - Array, dynamic data structure etc

BST Abstract Data Type

- Logically, the nodes of the BST are structured as a tree, with a maximum of two (i.e. binary) branches per node
  - A BST is *ordered*
  - Usually, smaller values go to the left, larger to the right

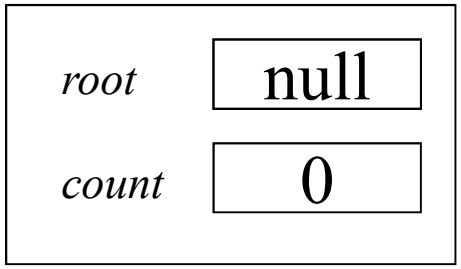


Binary Search Trees

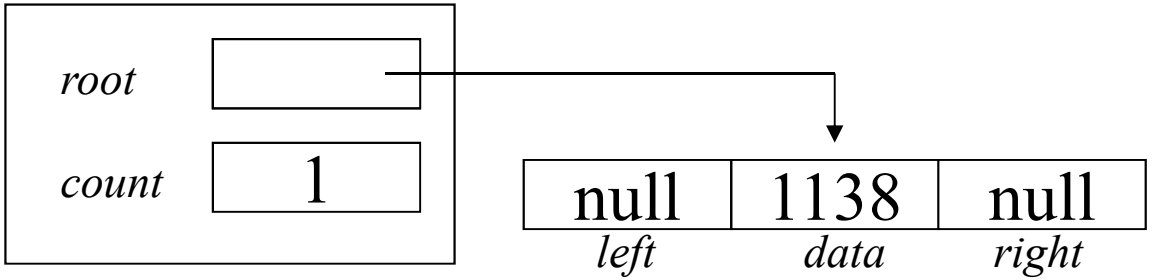
- Each node in a binary tree can hold some data, plus ‘pointers’ to up to two other binary sub-trees.
- By convention we describe these pointers (or links) as pointing to the left or right sub-tree.
- Binary search trees impose an order on the data held within the binary tree, and by doing so combine some of the best features of ordered arrays with linked lists.
- Binary search trees (BSTs) are dynamic, and therefore can grow/shrink as application needs change, and are ordered so that a binary tree search can be applied.
  - On average, each node traversal cuts the remaining search space in half

Binary Search Trees (cont'd)

- Empty BST



- BST with just one data item

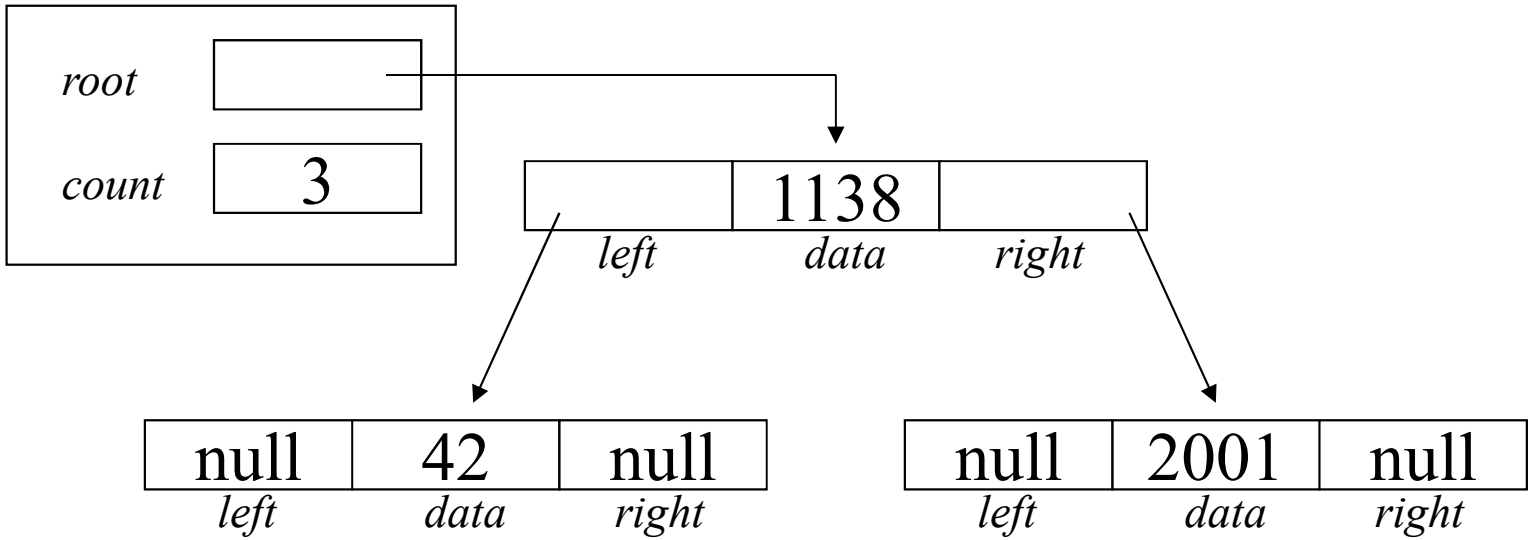


Binary Search Trees (cont'd)

- BSTs are ordered by (usually) placing the smaller values to the 'left' of existing values, and larger values (and duplicates) to the 'right'.
- Insertion into a BST always begins with a search from the top (root) level of the tree. Comparisons of the target data and the data at the 'current' level control whether the search continues down the left hand (<) branch or the right hand (>=) branch of the tree.
- This search is attempting to find the end of the branch onto which to attach the new leaf node to complete the insert. New nodes are always added as leaf nodes.

Binary Search Trees (cont'd)

- BST with three nodes



- We can use the *same* IntList interface to manipulate a BST
  - (initial example implements a BST using an *array*)

IntList - intlist.h for *array*-based BST

```
/* IntList
 * -- simple BST implementation using an array
 */

  … as per previous versions …

#define INTLISTSIZE 100
#define EMPTY -1

typedef struct
{  int numArray[INTLISTSIZE];
   unsigned size;
} IntList;

 … as per previous version …

void inorderDisplay(IntList*, int);
int getLeftIndex(int);
Int getRightIndex(int);
```

Initialising an array-based BST

```
int MakeList(IntList* pil, int size)
{
   int i;
   phil->size = 0;
   for(i = 0; i < INTLISTSIZE; i++)
      phil->numArray[i] = EMPTY;

   return SUCCESS;
}
```

Inserting into an array-based BST

```
        /* for current node at array position i, work out index
         * of 'left child node' of this node */
int getLeftIndex(int i)
{
        return (2 * i) + 1;
}

        /* for current node at array position i, work out index
         * of 'right child node' of this node */

int getRightIndex(int i)
{
        return (2 * i) + 2;
}
```

Inserting into an array-based BST

```
int AddList(IntList* pBST,int data)
{
   int current = 0;

   if (pBST->size == 0)
   {
         /* This insertion is the first */
      pBST->numArray[0] = data;
      pBST->size += 1;
      return SUCCESS;
   }
```

Inserting into an array-based BST

```
      /* search for branch onto which to attach the new leaf node */
while (current < INTLISTSIZE && pBST->numArray[current] != EMPTY)
{
   if (data < pBST->numArray[current])
      current = getLeftIndex(current);
   else /* ie. data >= current->data */
      current = getRightIndex(current);
}

      /* are we off the end of the array? */
if (current >= INTLISTSIZE)
   return FAILURE;
```

Inserting into an array-based BST

```
/* create the new leaf node */
   pBST->numArray[current] = data;
   pBST->size += 1;

   return SUCCESS;
}
```

Traversal of a BST

- Traversal of a BST can be done recursively inorder/preorder/postorder

```
inorder(currentNode)
        inorder(currentNode->leftLink)
        process current node
        inorder(currentNode->rightLink)

preorder(currentNode)
        process current node
        preorder(currentNode->leftLink)
        preorder(currentNode->rightLink)

postorder(currentNode)
        postorder(currentNode->leftLink)
        postorder(currentNode->rightLink)
        process current node
```

Traversal of an array-based BST

- Exercise: Consider the following function definition and desk-check its operation on a small BST.

```
void inorderDisplay(IntList* pil, int current)
{
   if (current < INTLISTSIZE && pil->numArray[current] != EMPTY)
   {
      inorderDisplay(pil,getLeftIndex(current));
      printf("%d\n", pil->numArray[current]);
      inorderDisplay(pil,getRightIndex(current));
   }
}

void DisplayList(IntList* pil)
{
   inorderDisplay(pil,0);
}
```

IntList - intlist.h using a linked BST

```
/* IntList
 * -- simple BST implementation using dynamically allocated links
 */

  … as per previous versions …

typedef struct intlistnode * IntBSTNodePtr;
typedef struct intlistnode
{  int data;
   IntBSTNodePtr left;
   IntBSTNodePtr right;
} IntBSTNode;

typedef struct
{  IntBSTNodePtr root;
   unsigned size;
} IntBST;

 … as per previous version …
```

Inserting into a BST

```
int InsertBST(IntBST *pBST, int data)
{
   IntBSTNodePtr current, previous, newNode;

   previous = NULL;
   current = pBST->root;

   /* search for branch onto which to
    * attach the new leaf node.
    */
   while (current != NULL)
   {
      previous = current;
      if (data < current->data)
         current = current->left;
      else /* ie. data >= current->data */
         current = current->right;
   }
```

Inserting into a BST (cont'd)

```
/* Create the new leaf node */
newNode = malloc(sizeof IntBSTNode);
if (newNode==NULL)
   return FAILURE; /* Memory allocation failed */
newNode->data = data;
newNode->left = NULL;
newNode->right = NULL;
pBST->size += 1;

/* Our search (above) either revealed this insertion
 * is the first; hence a root node is being created;
 * or we found the branch on which to attach our new
 * leaf node
 */
if (previous == NULL)
{
   pBST->root = newNode;
   return SUCCESS;
}
```

Inserting into a BST (cont'd)

```
    /* if attached to a branch, we need to know if it
     * is attached as a left or right sub-tree.
     */
    if (data < previous->data)
    {
       previous->left = newNode;
    }
    else /* ie. data >= previous->data */
    {
       previous->right = newNode;
    }

    return SUCCESS;

} /* End of InsertBST */
```

BST Declarations and Traversal

```
void inorderDisplay(IntBSTNodePtr pNode)
{
   if (pNode)
   {
      inorderDisplay(pNode->left);
      printf("%d\n", pNode->data);
      inorderDisplay(pNode->right);
   }
}
```

free()ing a BST

• Consider the following function for free()ing a BST. It is recursive, and performed what is called a post-order traversal, that is, one where all of the leaf nodes are processed before recursively backtracking up the tree to process the higher up nodes.

```
void postorderFree(IntBSTNodePtr pNode)
{
   if (pNode)
   {
      postorderFree(pNode->left);
      postorderFree(pNode->right);
      free(pNode);
   }
}
```

Data Structure Comparison

- There are many ways we can compare data structures, and our purpose for doing so is to help with selection of the most appropriate data structure for our intended application.
- Comparisons are often based on the amount of memory (space) used to store a given set of data can be used, and on the amount of execution time (time) used to perform basic operations on the data structure. Eg. time to insert a new item, time to search for an existing item, and so on.
- The next slide shows a simple comparison of a range of data structures we have seen so far in this course. Follow-on courses from this one will examine many many other well known data structures.

Data Structure Comparison (cont'd)

- The table below presents a summary of the time costs for inserting and searching various data structures. The table entries represent the order of the function relating the size of the data structure (i.e. number of elements) to the execution time for the operation.

		average insert	***worst insert***	***average search***	***worst search***
unordered array		constant	constant	linear	linear
ordered array		linear	linear	logarithmic	logarithmic
unordered linked list		constant	constant	linear	linear
ordered linked list		linear	linear	linear	linear
binary search tree		logarithmic	linear	logarithmic	linear