Foundations of Programming for High Performance Computing: Assignment 2

Marc Moreno Maza
University of Western Ontario
CS211 b – Winter 2011

Posted: Thursday 16, 2012
Due: Friday, March 16, 2012

Problems 1 and 2 involve C programming. You need to submit both the electronic and printed version of the .c file for each of them.

1 Problem 1

1.1 Preliminaries

A *binary tree* is a tree data structure in which each node has at most two children, called the *left child* and the *right child*. Binary trees are a very popular *data-structure* in computer science. We shall see in this exercise how we can encode it using C arrays. The formal recursive definition of a binary tree is as follows. A *binary tree* is

- either empty,
- or a node with two children, each of which is a binary tree.

The following terminology is convenient:

- A node with no children is called a *leaf* and a node with children is called an *internal node*.

- If a node `A` is a child of a node `X` then we say that `X` is the *parent* of `A`.
- In a non-empty binary tree, there is one and only one node with no parent; this node is called the *root node* of the tree.

You will find examples and comments at the wiki page:

`http://en.wikipedia.org/wiki/Binary_tree`

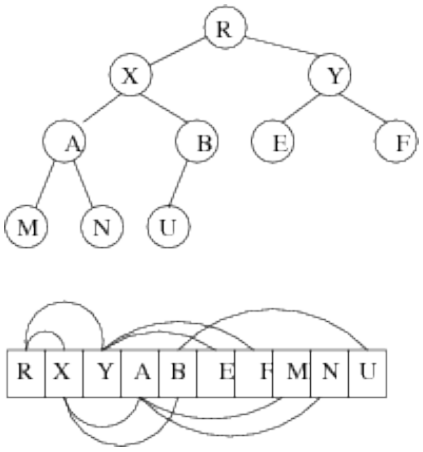


Figure 1: A binary tree and its array encoding.

A non-empty binary tree T with n nodes can be encoded in an array A. Indeed, one can always label the nodes with the integers $1, 2, \ldots, n$ such that:

- the root has label 1,
- if an internal node has label i then its left child (if any) has label $2i$ and its right child (if any) has label $2i + 1$.

Using this observation, we can store the nodes of T in A as follows:

- the root of the tree is in $A[1]$, and its left and right children are stored in $A[2]$ and $A[3]$ respectively,
- given the index i of an internal node, different from the root, the indices of its parent **PARENT(i)**, left child **LEFT(i)**, and right child **RIGHT(i)** can be computed simply by:

- **PARENT(i)** $= \lfloor i/2 \rfloor$
- **LEFT(i)** $= 2i$
- **RIGHT(i)** $= 2i + 1$

Consider the binary tree of the above figure:

- The root node `R` is stored in slot 1,
- its left child `X` is in slot 2, its right child `Y` is at slot 3,
- for `X`, its left child `A` is in slot 4 and its right child `B` is in slot 5,
- ...

1.2 Questions

The purpose of this problem is to realize a simple C implementation of binary trees encoded as arrays. This program will actually make use of the example in the above figure. To guide you toward this goal, we provide a template program hereafter. We ask you to use this template and fill in the missing code. In this template, we use the array `binaryTree[]` to encode the binary tree in Figure **??**. The first slot of `binaryTree[]` is used to record the number of nodes in the tree. The rest of the slots keep the values contained in the nodes. You are asked to implement 4 functions:

1. **printParentNode(int *bt, int node)**. For a given node index, this function will print the corresponding parent node. Your code should handle exceptional cases. If the given node is the root node, that is, if **node** is 1, then there is no parent. If the given node is not in the given tree, that is, if the given node index is too large or non-positive, then an error message should be printed.

2. **printLeftChildNode(int *bt, int node)**. For a given node index, this function will print its left child node. As above, your code should handle exceptional cases. Namely, when the given node has no left child, or the given node is not in the given tree, then an error message should be printed.

3. **printRightChildNode(int *bt, int node)**. Similar to the above function, except that it deals with the right child node instead of the left child node

4. **printNearestCommonAncestor(int *bt, int node1, int node2)**. For two given nodes, this function returns their nearest common ancestor in the tree. See the sample output below.

Remark. In the template code below, we also provide the prototype of an extra function **getIndex**, which you could use for implementing the above four functions.

The template code.

```
#include <stdio.h>


// INPUT: 'bt', a binary tree.
//        'node', a node in the tree.
// OUTPUT: the index of given value in the tree.
int getIndex(int *bt, int node){
 // ...
}

// INPUT: 'bt', a binary tree.
//        'node', a node in the tree.
// OUPUT: print the parent node of 'node'.
void printParentNode (int *bt, int node){
 //...
}

// INPUT: 'bt', a binary tree
//        'node', a node in the tree.
// OUPUT: print the left child of 'node'
void printLeftChildNode (int *bt, int node){
 //...
}

// INPUT: 'bt', a binary tree
//        'node', a node in the tree.
// OUPUT: print the right child of 'node'
void printRightChildNode (int *bt, int node){
```

```
 //...
}

// INPUT: 'bt', a binary tree
//        'node1', a node in the tree.
//        'node2', a node in the tree.
// OUPUT: print the nearest ancestor of 'node1' and 'node2'.
void
printNearestCommonAncestor(int *bt, int node1, int node2){
 //...
}




int main(){

  /* binaryTree is an array to encode a binary tree.
     The first element of this array is used to encode
     the number of nodes.
  */

  int binaryTree[11] = {10, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1};

  printParentNode(binaryTree, 16);
  printParentNode(binaryTree, 3);

  printLeftChildNode(binaryTree, 7);
  printLeftChildNode(binaryTree, 9);

  printRightChildNode(binaryTree, 7);
  printRightChildNode(binaryTree, 10);

  printNearestCommonAncestor(binaryTree, 8, 1);
  printNearestCommonAncestor(binaryTree, 8, 9);

  return 0;
}
```

The sample output from above template.

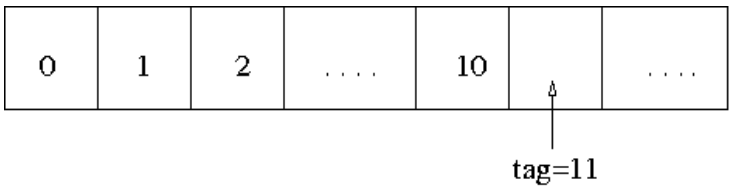
```
Node 16 is the root and has no parent.

The parent node of 3 is 10.

The left child node of 7 is 1.

Node 9 has no children!

Node 7 has no right child!

The right child node of Node 10 is 3.

The nearest common ancesor of Node 8 and Node 1 is Node 14.

The nearest common ancesor of Node 8 and Node 9 is Node 16.
```

2 Problem 2

2.1 Data structure

In this assignment, you are asked to implement the following data structures.

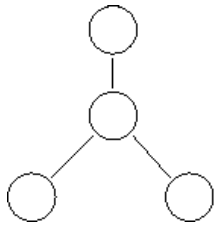
Bucket. A bucket consists of a fixed-size array and a tag, which is the index of the first free slot in the array. More precisely, all buckets have the same size, typically 256 bytes. The following gives an example of how it looks like.



Its type definition in **C** is as follows:

```
/* definition of a bucket */
typedef struct {
  int *elements;
  int tag;
} Bucket;
```

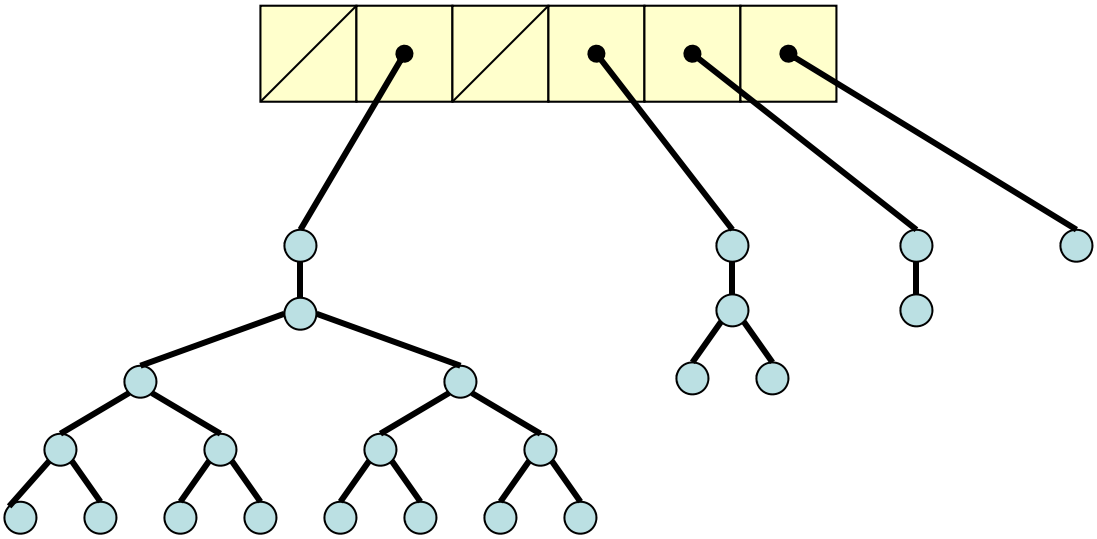
Pennant. A pennant is a special binary tree. It consists of a root and a left child. This child itself is a complete binary tree. Therefore a pennant always has 2^k nodes for some integer k, where k is called the *height* of the pennant. For example, a pennant consists with 4 nodes is depicted below:



In the implementation, each node of a pennant stores a bucket. For convenience, we use the data structure of a general binary tree to represent a pennant. Therefore we have the following type definition.

```
/* definition of a pennant */
typedef struct pennant {
  int height;
  Bucket *bucket;
  struct pennant *left;
  struct pennant *right;
} Pennant;
```

Bag. A bag is a collection of pennants, each of a different size. More precisely, a bag is a fix-sized array, where the k-th entry, for $k \geq 0$, in the array is either a null pointer or a pointer pointing to a pennant of size 2^k. In addition, all bags have the same size, typically 16. The following depicts an example of a bag.



In the implementation, besides the array, a bag has an extra pennant which has a single node and which works like a cache. (We will explain thereafter in more detail the use of this extra pennant.) Therefore, the data structure of a bag is defined as follows.

```
/* definition of a bag */
typedef struct {
  Pennant* *elements;
  Pennant* pennant;
  int tag;
} Bag;
```

Observe that `elements` is an array of pointers, each of them pointing to a pennant. In the above definition, `tag` is the index of the first `NULL` pointer in the array `elements`. If there is no null pointer in `elements`, the value of the `tag` is the size of the bag, which implies that the bag is full.

2.2 Operations on each data structure

Based on the data structures in the last section, you are asked to implement the following operations.

- **NewBucket**. The prototype is

```
Bucket* NewBucket()
```

It is used to create a new bucket. Recall that all buckets have the same size, called BUCKET_SIZE and which must be defined in a header file. You will need to use `malloc` in order to allocate memory space for this bucket.

- **FreeBucket**. The prototype is

```
void FreeBucket(Bucket* pBucket)
```

It is used to free the memory space you allocated for the bucket passed as argument.

- **PrintBucket**. The prototype is

```
void PrintBucket(Bucket* pBucket);
```

It is used to print the elements of the bucket passed as argument.

- **NewPennant** The prototype is

```
Pennant* NewPennant()
```

It is used to create a new pennant with one node **only**. To do so, you will need to create a new bucket for that node.

- **FreePennant** The prototype is

```
void FreePennant(Pennant* pPennant)
```

It is used to free the memory space you allocated for the pennant.

- **PrintPennant**. The prototype is

```
void PrintPennant(Pennant* pPennant);
```

It is used to print the elements of the pennant passed as argument. You are required to print the elements of the pennant as follows:

1. first print the elements in the bucket of the root of the pennant;
2. secondly print the elements of each bucket of the left child following an *infix traversal*

For a complete binary tree `T`, an infix traversal prints the root of `T`, followed by an infix traversal of its left child (if any), followed by an infix traversal of its right child (if any).

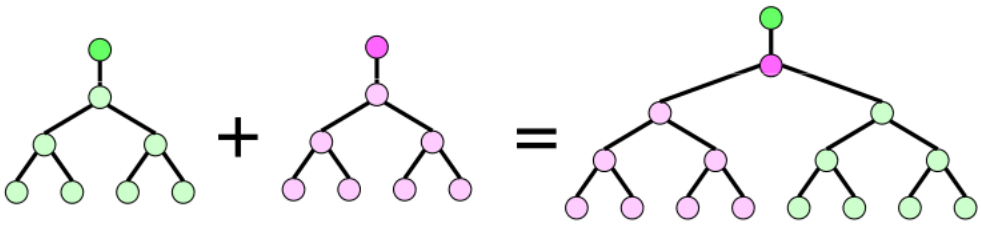
- **UnionPennant**. The prototype is

```
void UnionPennant(Pennant* p1, Pennant* p2);
```

For two pennants, both with 2^k nodes for some integer $k \geq 0$, this function creates a new pennant with 2^{k+1} nodes, consisting of the nodes of the input pennants. This function uses the pointer `p1` to point to the resulting pennant. This *union* operation runs in the following way:

1. It modifies `p2` such that it is now a complete binary tree where the right child of the root of `p2` is the left child of the root of `p1`.
2. It modifies `p1` so that the left child of the root of `p1` is now `p2`.

Observe that both the pennants pointed by `p1` and `p2` are modified by this operation. Recall that the new pennant is pointed by `p1`. See the following picture for illustration.



- **NewBag**. The prototype is

```
Bag* NewBag()
```

It is used to create a new bag. Recall that all bags store their pennants in an array of the same size, called BAG_SIZE. This size must be defined in a header file. You will need to use `malloc` in order to allocate memory space:

1. for this array and,
2. also for the extra pennant associated with the bag and used for caching; more details on this thereafter.

- **FreeBag**. The prototype is

  ```
  void FreeBag(Bag* pBag)
  ```

  It is used to free the memory space allocated for all pennants inside the bag.

- **PrintBag**. The prototype is

  ```
  void PrintBag(Bag* pBag)
  ```

  It is used to print the elements in the bag. You need to print the bag in the following way:

  1. first print the elements in the extra pennant.
  2. then print the pennants in the array, one after the other, by increasing index of their slot.

- **InsertBag**. The prototype is

  ```
  void InsertBag(int element, Bag* pBag)
  ```

  It is used to insert an integer into a bag. This operation needs to be implemented in the following way.

  1. First check whether the *cache* (i.e. the bucket of the extra pennant) of the bag is full. if the cache is not full, it inserts `element` into the cache and the operation returns. Otherwise go to the next step

2. Check whether the bag is full. If this is the case, then one prints "The bag is full." and the operation returns. Otherwise go to the next step
3. At this point, the cache, say x is full but not the bag. Then, a new (and empty) cache is created and `element` is inserted there. Meanwhile x is inserted into the array of the bag in the following way.
   (a) If $pBag[0]$ is null, then $pBag[0] = x$ and the operation returns.
   (b) If $pBag[0]$ is not null, then do
   `UnionPennant(pBag[0], x)`
   (c) now $pBag[0]$ has two elements; if $pBag[1]$ is null, the $pBag[1] = pBag[0]$; $pBag[0] = NULL$, and the operation returns.
   (d) otherwise, do
   `UnionPennant(pBag[1], pBag[0])`
   and check whether $pBag[2]$ is null or not.
   (e) And so on so forth, until you find a k such that $pBag[k] = NULL$; then set $pBag[k] = pBag[k-1]$; $pBag[k-1] = NULL$; $pBag[k-2] = NULL$; ...; $pBag[0] = NULL$; and the operation returns.

2.3 Organizing the code into multiple files

For this assignment, you are asked to organize the code in the following way:

- In the file **bucket.h**, define the type **Bucket** and declare the prototype of the operations on **Bucket**.
- In the file **bucket.c**, implement the functions on **Bucket**.
- In the file **pennant.h**, define the type **Pennant** and declare the prototype of the operations on **Pennant**.
- In the file **pennant.c**, implement the functions on **Pennant**.
- In the file **bag.h**, define the type **Bag** and declare the prototype of the operation on **Bag**.
- In the file **bag.c**, implement the functions on **Bag**.

- In the file **macros.h**, define the macro BUCKET_SIZE = 4 (the size of any bucket array) define the macro BAG_SIZE = 16 (the size of any bag array).
- In the file **mybag.h**, include all the header files you created.
- In the file **main.c**, you are asked to prompt the user to type a positive integer n. Then your program will
  1. create a new bag
  2. insert all integers $1, \ldots, n$ into the bag
  3. print the elements in the bag
  4. free all allocated memory spaces and terminate.

2.4 Creating a Makefile to compile the source code

You are asked to create a Makefile to compile your source code, similar to the one for linked lists described in class. When "make" is typed, an executable program called "mybag" is generated. Typing "make clean" cleans all the files generated by "gcc".

2.5 Testing your program

Your program should have no segmentation fault, no memory leak. Your program should print all the elements correctly. For example, running `mybag` and entering `17` should produce the following:

```
How many elements you want to insert:
17
The elements are:
17 1 2 3 4 9 10 11 12 13 14 15 16 5 6 7 8 .
```

3 Problem 3

Write two C programs `testList` and `testBag` with the following specifications:

- Both programs read an integer `k` from the user

- `testList` and `testBag` creates an initial empty linked list and bag data structure, respectively. These linked list and bag data structure will store integers of type `int`.
- for `i` from 1 to $2 * 10^{\texttt{k}}$ add a random integer into the linked list (resp. bag data structure).
- Each program frees the space it has allocated and terminates

For each `k` in the range 3 to 6, measure the running time of `testList` and `testBag`. Interpret the results.