UNIVERSITY OF CALIFORNIA BERKELEY Engineering 7
Department of Civil and Environmental Engineering
Spring 2013 Professor: S. Govindjee

Data Structures: Linked Lists and Trees

1 Introduction

So far we have seen a number of different types of data structures. The most elementary were the primitive data types like `double` and `char`. By means of the `[ ]` and { } symbols we were able to build these up into the higher order data types of arrays and cell-arrays. These latter two data types are examples of indexed data and are very useful concepts. Notwithstanding, there are a few other common indexed data types that are useful to know about: linked lists and trees. Both are types of indexed data but they allow for some types of flexibility that are not present with arrays.

2 Linked Lists

Linked lists are like 1-dimensional arrays, except that they also support a delete operation. This is something that is not supported with arrays. With an array you can not simply remove an element from the middle without also having to perform (potentially a lot of) data copying. For example if you have an array `A` and you want to remove the `30`$^{\texttt{th}}$ element, then one needs to execute

```
A = [A(1:29) A(31:end)]
```

In this process one ends up creating a whole new array by copying the first 29 entries and then copying entries 31 up to the end of the array to a new memory location and then binding `A` to this new data. In the best case the first 29 entries are not moved but one still has to move entries 31 to the end of the array. A linked list allows us to perform such an operation with much less data movement and hence in many cases much more efficiently.

Linked lists are based on a node structure with two fields: a value field, and a next field. A node structure represents one element of the list and an array of nodes represents the whole list. The values that one wishes to store in the list are placed in the value field of the nodes. The next field of each node tells you which node in the array comes next in the list. If the next field equals zero then one has arrived at the end of the list. The start of the list is termed the root. There are many schemes for storing lists but we will adhere to the following convention:

- The list itself will be a structure `list`.
- The root node index will be stored in `list.root`.

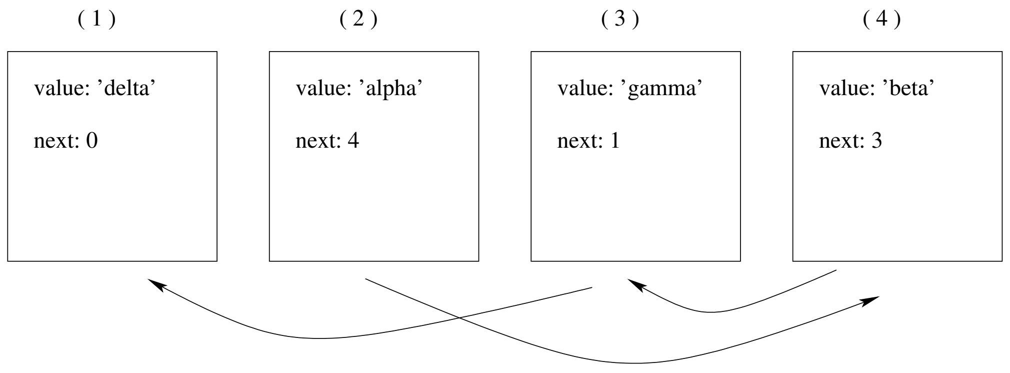

Figure 1: A linked list with 4 elements, where the root node is located at index 2.

- The values will be stored in `list.node( ).value`.
- The next pointers will be stored in `list.node( ).next`

Figure 1 shows a simple way of diagramming a list with 4 nodes. The root of the list is node 2 (it need not start with node 1) and the end of the list is node 1. To define the list in MATLAB one could type:

```
list.root = 2;
list.node(1).value = 'delta';
list.node(1).next = 0;
list.node(2).value = 'alpha';
list.node(2).next = 4;
list.node(3).value = 'gamma';
list.node(3).next = 1;
list.node(4).value = 'beta';
list.node(4).next = 3;
```

The order of the values in the list is `'alpha'`, `'beta'`, `'gamma'`, `'delta'`; i.e. node 2 comes first (it is the root), followed by node 4, then by node 3, and finally by node 1. On the surface this seems rather complicated since one could simply define a cell array to accomplish the same thing `CellArray = {'alpha', 'beta', 'gamma', 'delta'}`. However, lists provide flexibility that cell arrays do not as we shall see. As we look at developing functions to help us manage lists, it is important to keep in mind that the order of the data is defined by the next pointers and that the length of `[list.node]` may not be equal to the number of actual entries in the list.

2.1 Length of a list

Let us first consider how to compute the length of a list (which is not necessarily equal to the length of `[list.node]`). Perhaps the simplest way to accomplish such a task is to start at the root node and to traverse the list, counting along the way, until the end is reached. A simple function to accomplish this is:

```
function len = listLength(list)
% Usage: len = listLength(list)
% Purpose: Compute the length of a list
% Inputs: list -- a list structure
% Outputs: len -- the length of the list

% Check if the root is valid
if list.root == 0
    len = 0;
    return;
end

% Initialize the counter
len = 1;
idx = list.root;

% Iterate on the list, accumulate the number of items until the end is reached
while ( list.node(idx).next ~= 0 )
    len = len + 1;
    idx = list.node(idx).next;
end

end
```

First we check if the root is valid. If it is invalid the list has zero length. Next we simply add one to the counter `len` to accumulate the length of the list.

2.2 Inserting a value into a list

Let us consider inserting the value `'epsilon'` into our list after the node located at index 2. In order to do this, we need to add another node to the list, which is most easily done by appending it to the end of the node array. Then we need to set the next pointer of the new node to point to where the next pointer of node 2 points to; finally the next pointer of the node at index 2 should be set to point to our new node. Pictorially, we should have the situation in Fig. 2. This pattern is general and thus we can easily write a function to perform this operation on any list with respect to any index:

```
function list = listAdd(list,idx,val)
% Usage: list = listAdd(list,idx,val)
% Purpose: Add val to the list AFTER item at idx
% Inputs: list -- a list structure
%         idx -- index after which to add item
%         val -- value of the item to add
% Outputs: list -- modified list

% Compute an index one beyond the current end
newpos = length(list.node) + 1;
```

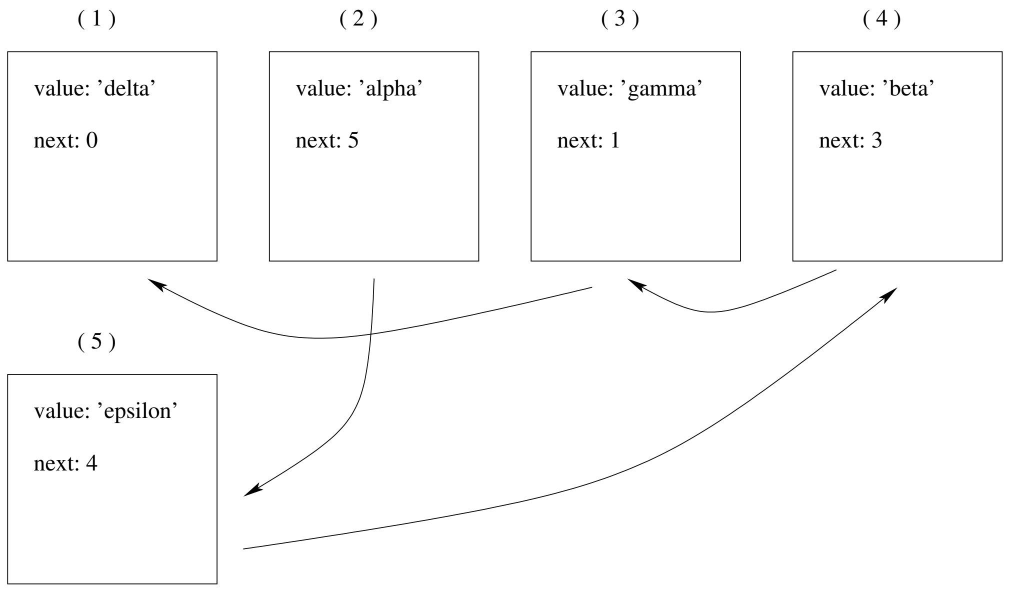

Figure 2: A linked list with 5 elements, where the root node is located at index 2.

```
% Insert the value
list.node(newpos).value = val;

% Set the next pointer of the new entry
list.node(newpos).next = list.node(idx).next;

% Reset the next pointer of the entry at idx
list.node(idx).next = newpos;

end
```

2.3 Removing a value from the list

To remove a value from a list works almost as simply as adding an item. Knowing the index of the item to be removed, all one needs to do is reset the next pointer of the node that points to it. Consider our five element list and removing the item at index 3. In this situation we take the value of the next pointer of node 4 and change it so that it points to where node 3 was pointing. The resulting structure appears as in Fig. 3. It is important to note that length of `[list.node]` remains at 5 and that no data needed to be shifted around. A general purpose function to accomplish this task is given as:

```
function list = listRemove(list,idx)
% Usage: list = listRemove(list,idx)
% Purpose: Remove item at idx from list
% Inputs: list -- list structure
%         idx -- index of item to be removed
```

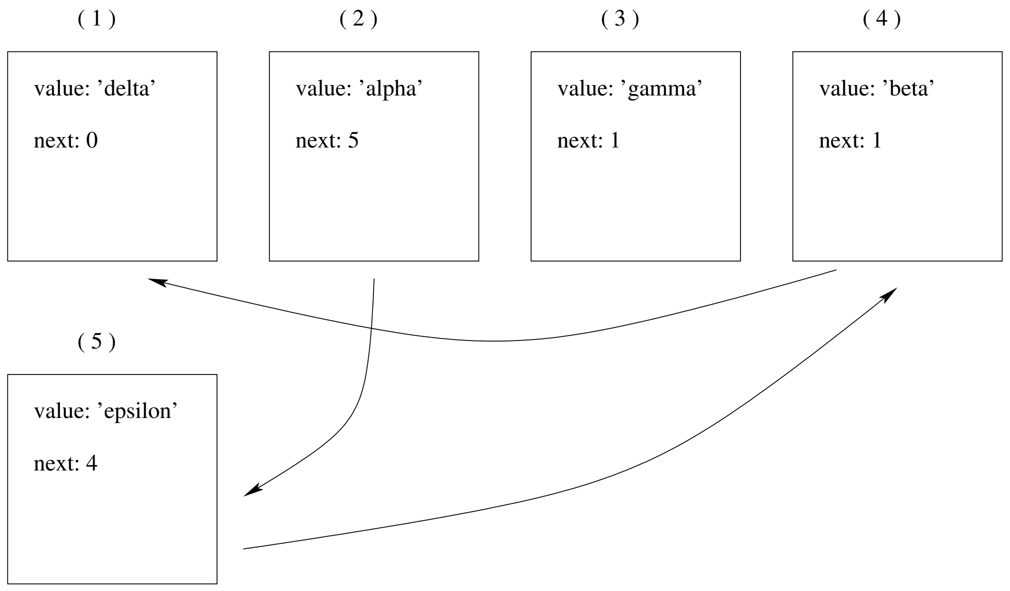

Figure 3: A linked list with 4 elements, where the root node is located at index 2.

```
% Outputs: list -- modified list with item at idx removed

if idx == list.root
    % Case of deleting the root node
    list.root = list.node(idx).next;
else
    % Find the index of the node prior to idx in the list
    pidx = listFindPrior(list,idx);
    % Reset the prior nodes next pointer
    list.node(pidx).next = list.node(idx).next;
end

end
```

This function uses the helper function `listFindPrior` which as the following listing:

```
function p = listFindPrior(list,idx)
% Usage: p = listFindPrior(list,idx)
% Purpose: Find the index of the node before the node at idx
% Inputs: list -- list structure
%         idx -- index of the node whose prior is desired
% Outputs: p -- index of prior node

% Nothing comes before the root node
if idx == list.root
    p = [];
else
    % Traverse the list until the next field points to idx
    p = list.root;
    while list.node(p).next ~= idx
        p = list.node(p).next;
    end
end

end
```

2.4 Other important list functions

There are many things that one can do with lists and the functions given above provide some important functionalities. Beyond these, other common functions include:

- Functions to print the list either forwards or backwards. The forward print is easily programmed using iteration or recursion. The reverse printing is most easily handled recursively.
- A function to find the indices (sometimes called the keys) associated with a given value.
- A function to compress a list's node array to remove entries that are no longer reachable while traversing the list.

3 Trees

A linked list is a data structure that is linear – one element follows another. For certain applications this is not a natural way to organize data and we need other organizational strategies. Consider a data structure to store the geopolitical landscape of the world. The world is organized into countries and each country is organized into states, and the states are organized in to cities, etc. If we want to preserve this hierarchy in a data structure that stores this information, then we will need something different from an array or a linked list. Our data can be visualized as a *tree* as shown in Fig. 4. The *root* of the tree is the world and the world points to all the countries in the world; in the language of trees, these are known as the *children* of the world node. The *parent* of a country node is the world. Each country in turn points to its states and each state can point to its cities. If we stop at the city as our smallest geopolitical entity then the city (nodes) will be called *leaves*. Leaves have no children. If a node in a tree can have at most two children, then we call the tree a *binary tree*. Figure 5 shows an example of a binary tree. In this tree the value associated with the root is 100. The root has a left child with value 75 and a right child with value 200. The node with value 75 has only a left child with value 25 – this is a leaf. The node with value 200 has two children and its right child has only a left child. The parent of the node with value 250 has a value of 300. The purpose or need for such a data structure is perhaps not apparent but later in the course we will see that this can be a very handy way to organize certain types of data.

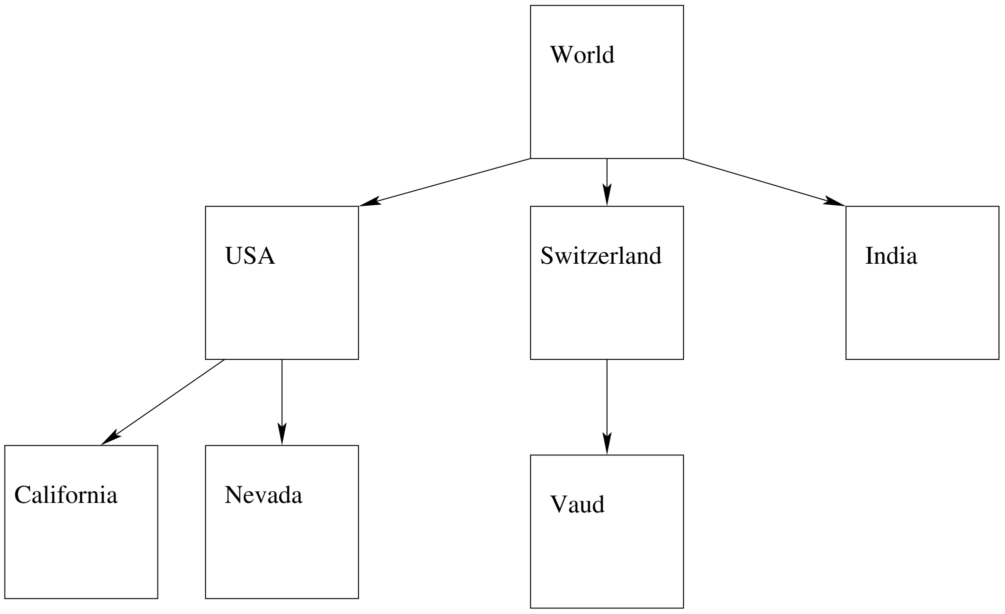

Figure 4: A tree structure to organize the geopolitical landscape of the world.

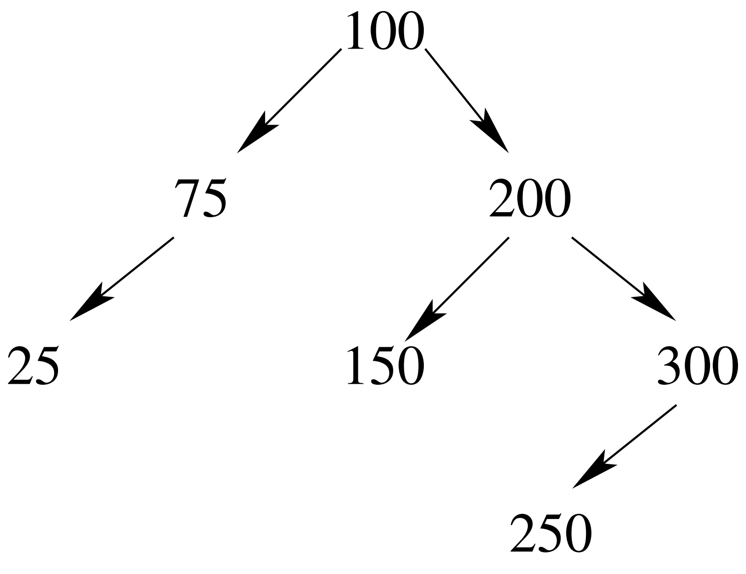

Figure 5: Example of a binary tree.

To keep things simple let us concentrate on binary trees. To store a binary tree we will use a structure similar to what we did with linked lists.

- The tree will be a structure `tree` with several fields.
- The index of the root node will be stored in `tree.root`.
- The values of the tree will be stored in an array of structures: `tree.node( ).value`.
- Each node structure will also have fields for storing pointers to the left and right children: `tree.node( ).left` and `tree.node( ).right`. If there is no child then the pointer will take on value 0.

A direct way to create the tree for the data in Fig. 5 would be to type

```
tree.root = 1;
tree.node(1).value = 100;
tree.node(1).left = 2;
tree.node(1).right = 4;

tree.node(2).value = 75;
tree.node(2).left = 3;
tree.node(2).right = 0;

tree.node(3).value = 25;
tree.node(3).left = 0;
tree.node(3).right = 0;

tree.node(4).value = 200;
tree.node(4).left = 5;
tree.node(4).right = 6;

tree.node(5).value = 150;
tree.node(5).left = 0;
tree.node(5).right = 0;

tree.node(6).value = 300;
tree.node(6).left = 7;
tree.node(6).right = 0;

tree.node(7).value = 250;
tree.node(7).left = 0;
tree.node(7).right = 0;
```

Like with linked lists there are many common operations that are performed on trees. The basic feature of these operations is one where one traverses the tree and does some kind of operation with the value stored in the node of the tree. In a linked list there are two ways of doing this: start at the root and perform work down to the end of the list or traverse to the end of the list and perform work on the way up the list. With trees one speaks of preorder traversal, postorder traversal, and inorder traversal. These are defined as:

Preorder: Do something with the current node; visit the children.

Postorder: Visit all the children; do something with the current node.

Inorder: Visit the left child; do something with the current node; visit the right child.

Let us look how one performs a few common functions on trees.

3.1 Print a tree using preorder traversal

To print the tree in Fig. 5 using preorder traversal should result in the ordered output of 100, 75, 25, 200, 150, 300, 250. The easiest way to program this is to use a recursive algorithm:

```
function treePrintPreorder(t,idx)
% Usage: treePrintPreorder(t,idx)
% Purpose: Print a tree with preorder traversal starting at idx
% Inputs: t -- a tree structure
%         idx -- index to start printing from
% Outputs: Print to the command window

disp(t.node(idx).value)

left = t.node(idx).left;
right = t.node(idx).right;
if left ~= 0
    treePrintPreorder(t,left);
end
if right ~= 0
    treePrintPreorder(t,right);
end

end
```

To convert this to a postorder printing one simply has to move the `disp` statement after the two recursive calls. In this case, the output would be 25, 75, 150, 250, 300, 200, 100. An inorder traversal would have the `disp` statement between the two recursive calls. In this case, the output would be 25, 75, 100, 150, 200, 250, 300.

3.2 Count the leaves of a tree

To count the leaves of a tree we need to count how many nodes have no children. Again we can traverse the tree in three basic ways: preorder, inorder, or postorder. A recursive postorder traversal to approach this problem would look like:

```
function [cnt] = treeLeafcount(t,idx)
% Usage: treeLeafcount(t,idx)
% Purpose: Count the leaves of a tree starting at idx
% Inputs: t -- a tree structure
%         idx -- index to start counting from
% Outputs: cnt -- number of leaves

left = t.node(idx).left;
right = t.node(idx).right;

if left==0 & right==0
    cnt = 1;
else
    cntr = 0;
    cntl = 0;
    if left ~=0
        cntl = treeLeafcount(t,left);
    end
    if right ~= 0
        cntr = treeLeafcount(t,right);
    end
    cnt = cntl + cntr;
end

end
```

We descend left then right, then we compute the number of leaves: `cnt = cntl + cntr`

3.3 Prune a tree

The pruning of a tree is a very simple operation. If we know the index of the node that is to be pruned, then we simply set its parent's pointer to zero. Note that pruning eliminates all the descendents of a given node, but similar to linked lists, no data is moved.

```
function t = treePrune(t,idx)
% Usage: treePrune(t,idx)
% Purpose: Prune tree starting at idx
% Inputs: t -- a tree structure
%         idx -- index to start pruning from
% Outputs: Modified tree

% First find the node's parent index and side (left,right)
[pidx,side] = treeFindParent(t,t.root,idx);

% Set the appropriate child pointer to zero
switch side
    case 'left'
        t.node(pidx).left = 0;
    case 'right'
        t.node(pidx).right = 0;
end
```

```
end
```

This function makes use of the helper function that finds the parent:

```
function [p,side] = treeFindParent(t,stix,idx)
% Usage: [p,side] = treeFindParent(t,stix,idx)
% Purpose: Find the parent of the node at idx
% Inputs: t -- tree structure
%          stix -- node to start the search from
%          idx -- index of the node whose parent is desired
% Outputs: p -- index of parent node
%           side -- side (left,right) of idx

p = [];
side = [];

left  = t.node(stix).left;
right = t.node(stix).right;

if idx == t.root
    % Nothing comes before the root node
    return;
elseif left == idx
    p = stix;
    side = 'left';
elseif right == idx
    p = stix;
    side = 'right';
else
    % Traverse the tree until a child field points to idx
    if left ~= 0
        [p,side] = treeFindParent(t,left,idx);
    end
    % If empty on the left look at the right
    if isempty(p) & right ~= 0
        [p,side]=treeFindParent(t,right,idx);
    end
end
```

There are many uses for tree data structures. At this point it is only important to be aware of them and be comfortable with how one operates with them. When appropriate, we will use tree structures in the latter part of the course.