

Binary trees

- a binary tree is a finite set of nodes which is either empty or consists of a data item (called the `root`) and two disjoint trees called `left` and `right` subtrees
- recall with the list implementation, there was no easy and fast method for in order insertion
 - conversely binary trees allow for very fast in order insertion
- peculiarly the data structure declaration for a binary tree resembles that of a double linked list
 - the access mechanisms might also be very similar

Properties of binary trees of integers

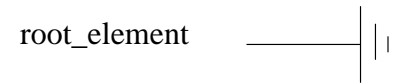
- are traditionally organised such that:
 - left branch contains values $<$ root value
 - right branch contains values \geq root value

- the root value here is the value at a possible node (or subtree)
 - not the value at the top of the complete tree

Example 1

- let us add: 2, 3, 1 and 4 to a binary tree

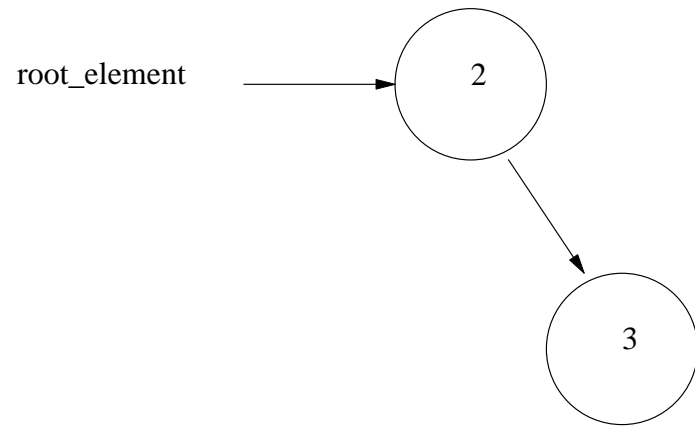
Empty tree



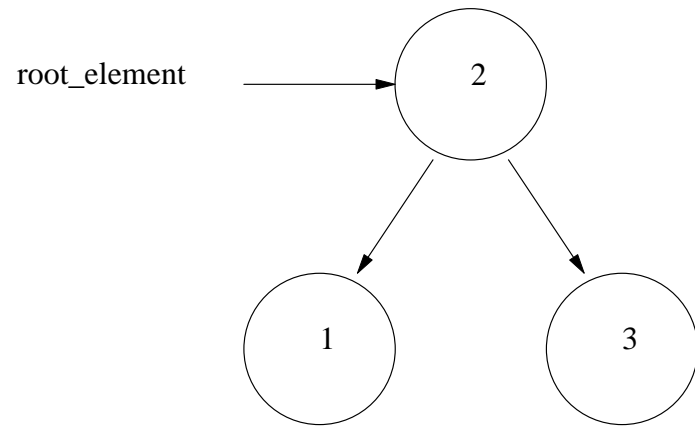
Adding 2 to the tree



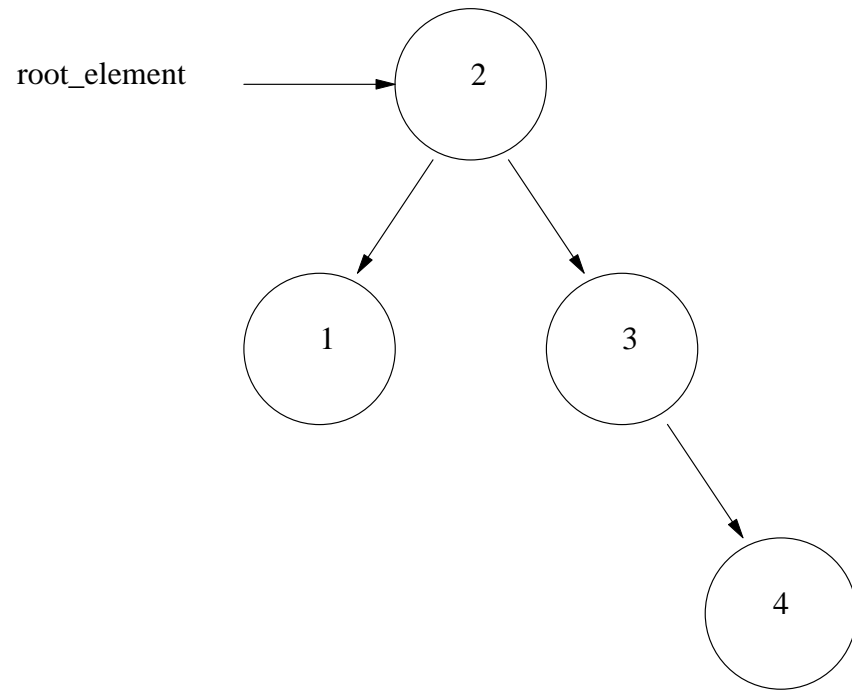
Adding 3 to the tree



Adding 1 to the tree



Adding 4 to the tree



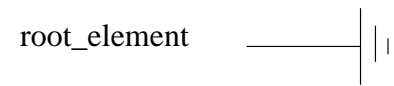
tree organisation

- we note that if we were to add numbers in a different order the tree would be populated differently
- yet the tree still maintains the property expressed earlier

Example 2

- let us add: 1, 2, 3 and 4 to a binary tree

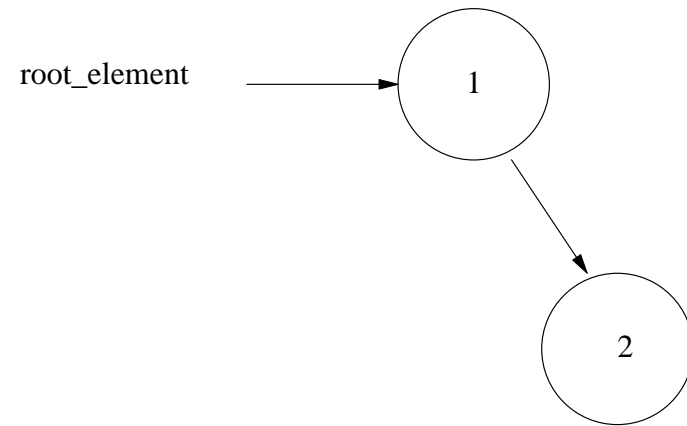
Empty tree



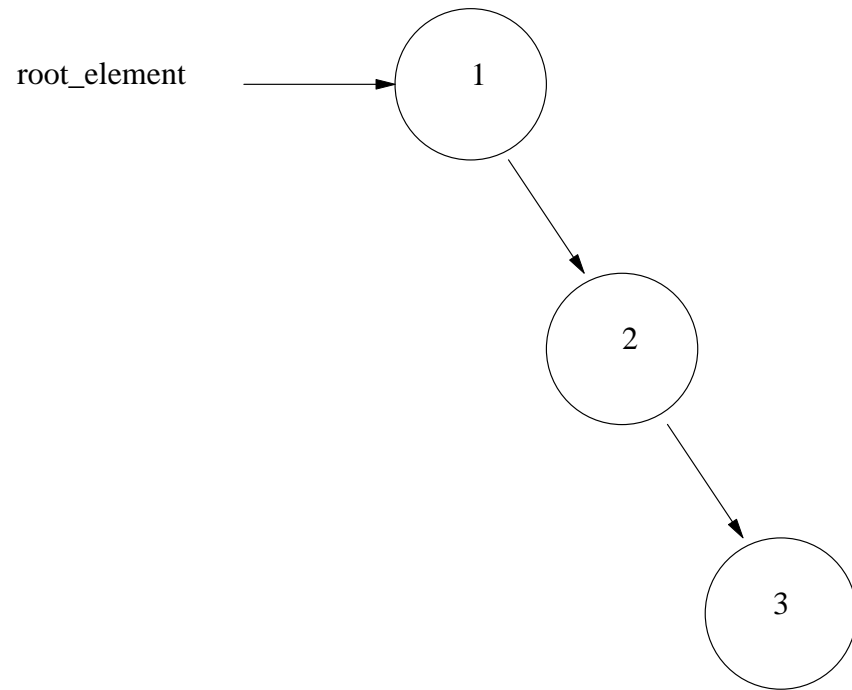
Adding 1 to the tree



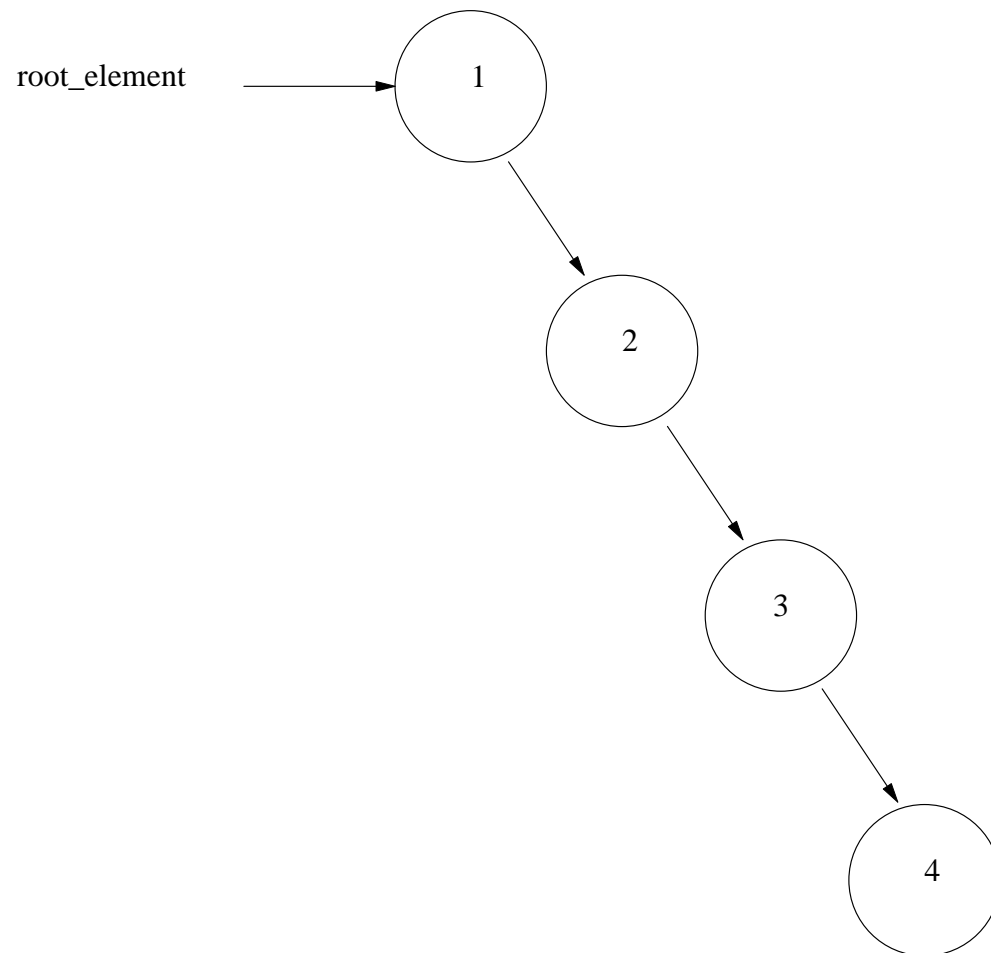
Adding 2 to the tree



Adding 3 to the tree



Adding 4 to the tree



Printing trees

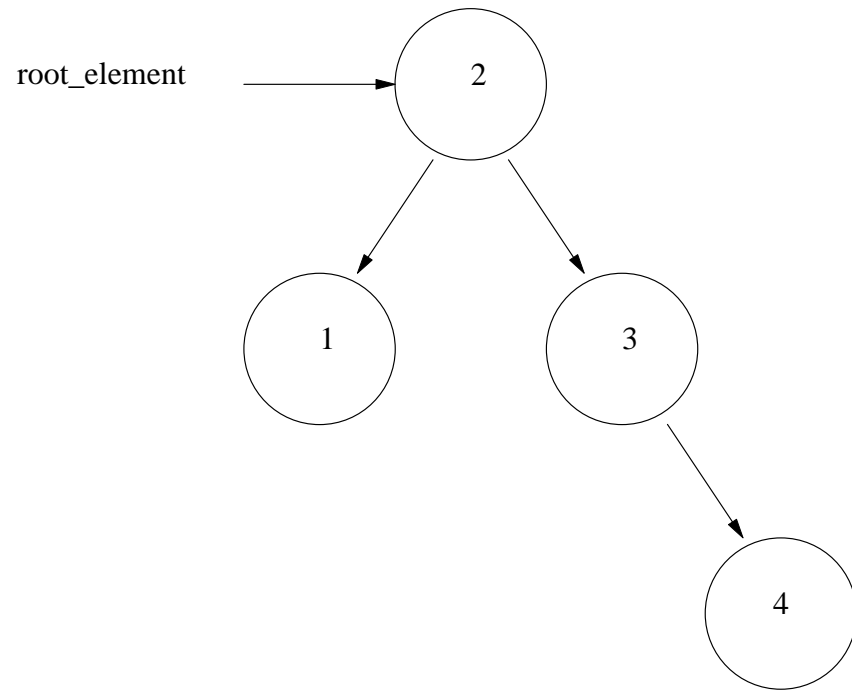
- there are three common methods of tree traversal:
 - (when do we print the root value?)

- preorder
 - **root**, left, right

- inorder
 - left, **root**, right

- postorder
 - left, right, **root**

Example of order



- preorder: 2, 1, 3, 4
- inorder: 1, 2, 3, 4
- postorder: 1, 4, 3, 2

class element definition



`examples/c++/trees/int/tree.h`

```
class element
{
public:
    element *left;
    element *right;
    int     data;
};
```

class tree definition

examples/c++/trees/int/tree.h

```
class tree
{
private:
    element *root_element;
    friend std::ostream& operator<< (std::ostream& os, const tree& l);
    // helper methods
    int no_of_items (element *e);
    int height (element *e);
    void delete_element (element *e);
    element *duplicate_elements (element *e);
    void insert (element **e, int i);
    tree cons (int i, element *l, element *r);
```

class tree definition

examples/c++/trees/int/tree.h

```
tree (void);  
~tree (void);  
tree (const tree &from);  
tree& operator= (const tree &from);  
  
void inorder (std::ostream& os, element *e);  
void preorder (std::ostream& os, element *e);  
void postorder (std::ostream& os, element *e);  
  
tree empty (void);  
bool is_empty (void);  
tree cons (int i, tree l, tree r);  
int root (void);  
tree cons (tree l);  
int height (void);  
int no_of_items (void);  
tree insert (int i);
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * tree - constructor, builds an empty list.  
 *     pre-condition:  none.  
 *     post-condition: tree is created and is empty.  
 */  
  
tree::tree (void)  
{  
    root_element = 0;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * ~tree - destructor, releases the memory attached to the list.  
 *       pre-condition:  none.  
 *       post-condition: tree is empty.  
 */  
  
tree::~~tree (void)  
{  
    delete_element (root_element);  
    root_element = 0;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * delete_element - pre-condition : none.  
 *                  post-condition: left and right branches are deleted.  
 */  
  
void tree::delete_element (element *e)  
{  
    if (e != 0)  
    {  
        delete_element (e->left);  
        delete_element (e->right);  
        delete e;  
    }  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * copy operator - redefine the copy operator.  
 * pre-condition : a tree.  
 * post-condition: a copy of the tree and its elements.  
 */  
  
tree::tree (const tree &from)  
{  
    root_element = duplicate_elements (from.root_element);  
}
```


class tree implementation

examples/c++/trees/int/tree.cc

```
/*
 * operator= - redefine the assignment operator.
 *           pre-condition : a list.
 *           post-condition: a copy of the list and its elements.
 *                       We delete 'this' lists elements.
 */

tree& tree::operator= (const tree &from)
{
    if (this->root_element == from.root_element)
        return *this;

    delete_element (root_element);
    root_element = duplicate_elements (from.root_element);
}
```

class tree implementation



[examples/c++/trees/int/tree.cc](#)

```
/*  
 * is_empty - returns true if tree is empty.  
 */  
  
bool tree::is_empty (void)  
{  
    return root_element == 0;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * empty - returns a new empty tree.  
 *       pre-condition:  none.  
 *       post-condition: a new empty tree is returned.  
 */  
  
tree tree::empty (void)  
{  
    tree *t = new tree;  
    return *t;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * cons - concatenate i to tree.  
 *       pre-condition:  none.  
 *       post-condition: returns a tree which has the  
 *                       contents of, l, and, r, as its branches.  
 */  
  
tree tree::cons (int i, tree l, tree r)  
{  
    return cons (i,  
                 duplicate_elements (l.root_element),  
                 duplicate_elements (r.root_element));  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*
 * cons - concatenate i to tree.
 *       pre-condition:  none.
 *       post-condition: returns a tree which has the
 *                       contents of, l, and, r, as its branches.
 */

tree tree::cons (int i, element *l, element *r)
{
    tree *t = new tree;
    t->root_element = new element;

    t->root_element->data = i;
    t->root_element->left = l;
    t->root_element->right = r;
    return *t;
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * root - returns the data at the root of the tree.  
 *       pre-condition : tree is not empty.  
 *       post-condition: data at the front of the list is returned.  
 *       tree is unchanged.  
 */  
  
int tree::root (void)  
{  
    assert (! is_empty());  
    return root_element->data;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * no_of_items - return the number of items in the tree.  
 * pre-condition : none.  
 * post-condition: returns an integer indicating  
 * the number of items in the tree.  
 */  
  
int tree::no_of_items (void)  
{  
    return no_of_items (root_element);  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * height - pre-condition : none:  
 *          post-condition: returns the max height between  
 *          the left and right branches.  
 */  
  
int tree::height (element *e)  
{  
    if (e == 0)  
        return 0;  
    else  
        return 1 + max (height (e->left), height (e->right));  
}
```


class tree implementation

examples/c++/trees/int/tree.cc

```
/*  
 * insert - places, i, into the tree in the correct place.  
 *         pre-condition : none.  
 *         post-condition: if i < data then i is stored to the left  
 *                         else i is stored to the right.  
 */  
  
tree tree::insert (int i)  
{  
    if (is_empty ())  
        return cons (i, 0, 0);  
    else  
        insert (&root_element, i);  
    return *this;  
}
```

class tree implementation

examples/c++/trees/int/tree.cc

```
void tree::insert (element **e, int i)
{
    if ((*e) == 0)
    {
        (*e) = new element;
        (*e)->left = 0;
        (*e)->right = 0;
        (*e)->data = i;
    }
    else
    {
        if (i < (*e)->data)
            insert (&(*e)->left, i);
        else
            insert (&(*e)->right, i);
    }
}
```