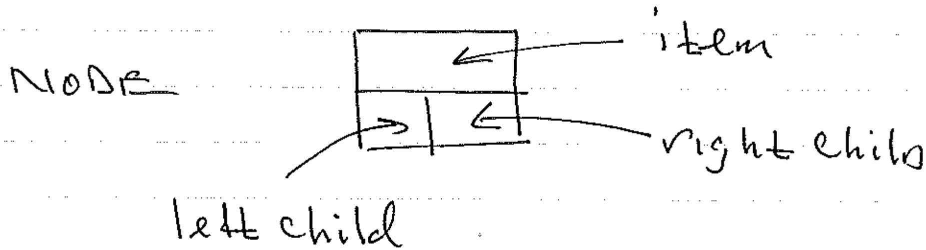
CHAP 11 – BINARY SEARCH TREES –

A BINARY SEARCH TREE IS A LINKED DATA STRUCTURE, IN SOME WAYS ANALOGOUS TO A LINKED LIST.



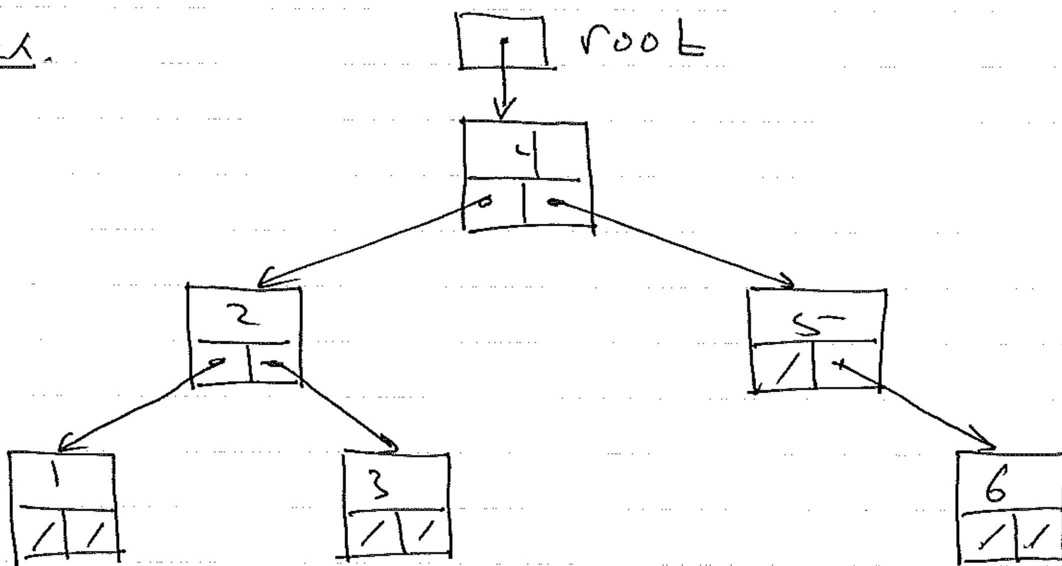
WE CREATE SUCH A NODE TYPE IN C BY

```
typedef struct Node {
    int item;
    struct Node* left;
    struct Node* right;
} Node;
```

AND A REFERENCE TO THIS Node BY

```
typedef Node* NodeRef;
```

<u>EX.</u>



BINARY SEARCH TREE PROPERTIES:

LET X AND Y BE OF TYPE NodeRef:

- IF Y IS IN THE LEFT SUBTREE OF X THEN y->item <= x->item
- IF Y IS IN THE RIGHT SUBTREE OF X THEN x->item <= y->item

THE BST PROPERTIES MAKES IT POSSIBLE TO TRAVERSE THE TREE IN SEVERAL ORDERS:

```
void printInOrder(NodeRef x){
    if (x != NULL){
        PrintInOrder(x->left);
        Printf("%d ", x->item);
        PrintInOrder(x->right);
    }
}
```

CALL: PrintInOrder(root)

OUTPUT FROM LAST EX:

1 2 3 4 5 6

```
void PrintPreOrder(NodeRef x) {
    if (x != NULL) {
        Printf("%d ", x->item);
        PrintPreOrder(x->left);
        PrintPreOrder(x->right);
    }
}
```

CAll: PrintPreOrder(root);

OUTPUT: 4 2 1 3 5 6

```
void PrintPostOrder(NodeRef x) {
    if (x != NULL) {
        PrintPostOrder(x->left);
        PrintPostOrder(x->right);
        Printf("%d ", x->item);
    }
}
```

CAll: PrintPostOrder(root)

OUTPUT: 1 3 2 6 5 4

THE BST PROPERTIES ALLOW ONE TO SEARCH FOR A GIVEN TARGET EFFICIENTLY. THE FOLLOWING FUNCTION RETURNS A REFERENCE TO THE NODE IN THE SUBTREE ROOTED AT R WITH item K, OR NULL IF NO SUCH NODE EXISTS.

```
NodeRef findItem(NodeRef R, int K){
    if( R==NULL || K==R->item)
        return R;
    if( K < R->item )
        return findItem(R->left, K);
    else
        return findItem(R->right, K);
}
```

TO SEARCH THE ENTIRE TREE WE CALL

```
findItem(root, K) ;
```

WHERE root is A NodeRef POINTING TO THE ROOT NODE.

THE RUNTIME OF findItem is $\Theta(h)$ WHERE h IS THE HEIGHT OF THE BST, IN WORST CASE.

OTHER OPERATIONS WHICH CAN BE PERFORMED EFFICIENTLY ARE findMin AND findMax WHICH RETURN THE NODES WITH MINIMUM AND MAXIMUM KEYS, RESPECTIVELY

```
NodeRef findMin(NodeRef R) {
   NodeRef L = R;
   if(L != NULL) for( ; L->left != NULL; L=L->left);
   return L;
}
```

THIS SIMPLY RETURNS THE LEFTMOST NODE IN THE SUBTREE ROOTED AT R, OR NULL IF R IS NULL.

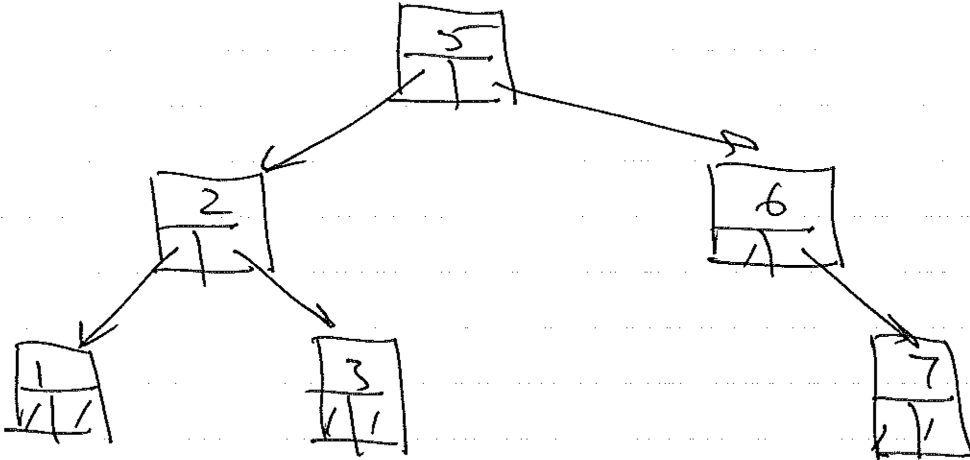
A BST CAN BE USED TO IMPLEMENT THE DICTIONARY ADT FROM PA2 AND LAB5. THE ESSENTIAL DICTIONARY OPERATIONS ARE

- Insert
- Delete
- lookup

lookup CAN BE BASED ON findItem ABOVE.

TO DO INSERTION WE DO SOMETHING VERY SIMILAR TO lookup. LET NodeRef A INITIALLY POINT TO THE ROOT, AND NodeRef B BE NULL. ALLOW A TO CLIMB DOWN THE TREE TO THE NULL CHILD POSITION WHERE THE NEW KEY BELONGS, WITH B MAINTAINED AS A'S PARENT. INSERT THE NEW NODE AS B'S CHILD.

Ex Insert(4):



A Deletion is considerably more complicated.
We have 3 cases

1: N has no children
2: N has 1 child
   2.1 left
   2.2 right
3: N has 2 children.

See Dictionary.c

1126

Dictionary.h

```
/*
*  Dictionary.h
*  Header file for the Dictionary ADT
*/

#if !defined(_DICTIONARY_H_INCLUDE_)
#define _DICTIONARY_H_INCLUDE_

#define UNDEF -1

/*
*  DictionaryRef
*  Exported reference type which points to a Dictionary struct
*/
typedef struct Dictionary* DictionaryRef;

/*
*  newDictionary
*  Allocates and initializes a struct representing an empty Dictionary.  Returns a
*  pointer to new memory if successful, terminates if not successful.
*/
DictionaryRef newDictionary(void);

/*
*  freeDictionary
*  Frees all heap memory associated with *pD and sets the reference *pD to NULL
*/
void freeDictionary(DictionaryRef* pD);

/*
*  isEmpty
*  pre: none
*  post: returns 1 (true) if D is empty, 0 (false) otherwise
*/
int isEmpty(DictionaryRef D);

/*
*  size
*  pre: none
*  post: returns the number of entries in D
*/
int size(DictionaryRef D);

/*
*  lookup
*  pre: none
*  post: returns value associated key k, or UNDEF if no such key exists
*/
int lookup(DictionaryRef D, int k);

/*
*  insert
*  inserts new (key,value) pair into D
*  pre: key k currently does not exist in D, i.e. lookup(D, k)==UNDEF
*  post: !isEmpty(D), size(D) is increased by one
*/
void insert(DictionaryRef D, int k, int v);

/*
*  delete
*  deletes pair with the key k
*  pre: key k currently exists in D, i.e. lookup(D, k)!=UNDEF
*  post: size(D) is decreased by one
*/
void delete(DictionaryRef D, int k);
```